Experimental Evaluation of Cooperative Voltage Scaling (CVS): A Case Study

Hiroshi Kawaguchi, Youngsoo Shin, and Takayasu Sakurai
Center for Collaborative Research, University of Tokyo
4-6-1, Komaba, Meguro-ku, Tokyo 153-8505, JAPAN

Abstract

Power-efficient design of real-time embedded systems becomes more important as the system functionality is increasingly realized through software. This paper presents a dynamic power management method called cooperative voltage scaling (CVS) and its experimental implementation. The implementation includes design of an operating system, applications, and a supporting hardware platform. We study several factors that are important in implementing CVS and discuss efficiency of CVS through experiment.

1. Introduction

Design of modern digital systems is a painstaking process due to various interrelated constraints involving high speed and low power. An increasing amount of system functionality tends to be realized through software there, which is leveraged by high performance processors. Re-programmable processors are frequently used in the systems in the form of off-the-shelf devices or cores. Consequently, *power-conscious design of software components* including an operating system (OS) and applications is important for high power efficiency.

Cooperative voltage scaling (CVS) is a dynamic power management method, which encompasses interaction between an OS and an application to reduce power consumed by a processor [1]. The OS is modified so that it maintains and provides timing information to the application. The application is also modified so that it consists of a sequence of *slices* and some additional *code fragments* are inserted at the head of each slice, which determine clock frequency (*f*) and supply voltage (V_{DD}) of the processor based on the timing information provided by the OS. The rationale of CVS lies in the fact that only the OS knows global information such as dynamic task interaction while each application has better knowledge about its own behavior.

In this paper, we address an experimental implementation of CVS to evaluate feasibility of model, implementation pitfalls, and efficiency. The implementation consists of three components: design of an power-conscious OS based on the Hitachi HI7750 industrial real-time OS (RTOS), design of an application following the concept of *application slicing*, and design of an hardware platform based on the Hitachi SH-4 microprocessor [2]-[3]. The designed system is evaluated by using an example of a task set and the power is reduced to less than 25% compared to the original system.

The remainder of this paper is organized as follows. In the next section, we briefly explain the concept of CVS. In Section 3, implementation details are presented and in Section 4, we discuss experimental results to evaluate CVS. Finally, conclusions with future research direction follows in Section 5.

2. Cooperative Voltage Scaling

As shown in Fig. 1, software architecture of CVS consists of an OS and applications. The applications are designed according to the concept of application slicing that will be shortly explained later on and the power-conscious version of the HI7750 is used. In the model of CVS, a set of applications consists of real-time tasks with its period (the minimum inter-arrival time between successive requests in case of sporadic tasks), deadline, and worst-case execution time (WCET). The real-time tasks are scheduled according to the fixed-priority preemptive scheduling algorithm even though other scheduling algorithms can be used.

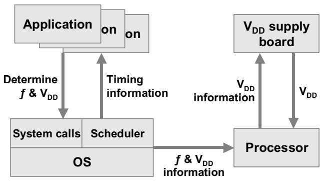

Fig. 1. Structural model of CVS.

The main function of the OS is to predict exact interval during which there is no activity on a processor, to provide *virtual deadline* to each task so that deadlines of all tasks are always guaranteed, and to put a processor into a sleep mode for idle time. This can be done based on status of queues[1].

1 There are two queues: a READY queue and a DORMANT queue. The READY queue holds tasks in order of priority, which are waiting to run. The DORMANT queue holds tasks in ascending numerical order of the time at which their initiation is due, which have already run in their periods and are waiting for their next periods to initiate again. If a task currently occupies a processor, it is called a RUN task. It should be noted that the RUN task is still in the READY queue even though it is in the RUN state.

For instance, first, if a RUN task (say task *A*) is the only task in a READY queue, we set virtual deadline of *A* to earlier time between deadline of *A* and initiation time of the task at the head of a DORMANT queue. Second, if there are more than one tasks in the READY queue, *A* completes its execution within its WCET. It is notable that there is still possibility to decrease *f* and V_{DD} because some slices might complete their execution earlier than their WCET and thereby, time margin to the subsequent slices might be provided. Third, if all tasks are in the DORMANT queue, we put the processor into a sleep mode.

The application-slicing algorithm to adaptively change *f* and V_{DD} according to workload of a processor is also of importance. In case of multimedia applications, since the workload depends strongly on data and execution time of the tasks frequently deviates from its WCET sometimes by a large amount, the control should be dynamic in run-time and should not be static in compile-time [4]. It is too late to notice that past task was an easy task that could be done much less than its WCET because once the task has completed, there is no way to change *f* and V_{DD} to lower the power. On the other hand, it is impossible to predict workload of tasks to be done in future without error. In order to solve this problem, the following algorithm as shown in Fig. 2 is used.

A task is assumed to be chopped into *N* slices, which potentially have different length one another. By checking the current time and time margin to execute the next slice, the optimum *f* is adaptively selected.

1. The following parameters are obtained by static analysis of an application or direct measurement [5].

- T_{WCET}: WCET of a task.
- T_{Wi}: WCET of the *i*-th slice.
- T_{Ri}: WCET from the (*i* + 1)-th to the *N*-th slice.

2. At the *i*-th slice, the target execution time (T_{TARi}) is calculated as $T_{TARi} = T_{Vd} - T_{Ri} - Ttr$ where *Ttr* is the sum of additional code fragment to determine appropriate *f* and V_{DD} and their transition time. Besides, calculation time of the code fragment is negligible because the calculation is easy. T_{Vd} is the time between the current time and virtual deadline, which is a dynamic variable and obtained from an OS via system call. Usually, T_{Vd} means starting point of the next task. The detail of the virtual deadline will be explained in the next section. It should be noted that T_{Vd} might change due to preemptive scheduling.

3. For each candidate $f_j = f_{max} / j$ (j = 1, 2, 3…), the estimated maximum execution time is calculated as $T_{Li,fj} = Ttr + T_{wi} \text{ x } j$. If f_j is equal to one at the (i – 1)-th slice, $T_{Li,fj} = T_{wi} \text{ x } j$.

4. *f* at the *i*-th slice (f_{VARi}) is selected as the minimum f_j whose $T_{Li,fj}$ does not exceed T_{TARi}.

Thus, *f* and V_{DD} are dynamically controlled on a slice-by-slice basis in each task by software. The relationship between *f* and V_{DD} is obtained by measurements.

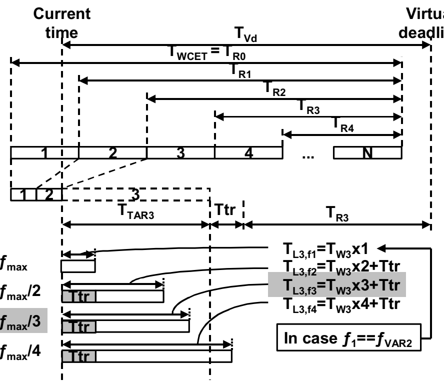

Fig. 2. Application slicing method to determine *f*. f_{max} means that the maximum *f* of a processor.

3. Implementation of CVS

3.1 Hardware platform

Fig. 3 shows a snapshot of the CVS experimental system and the SH-4 embedded system board. Fig. 4 corresponds to a block diagram of the system board. The processor has a clock frequency control register called FRQCR. *f* that is synchronized with the external clock of 33MHz can be instantaneously changed by accessing FRQCR. Since we use *f* of 200MHz and 100MHz that are divisible by the external clock, there is no synchronization problem with external devices at interface of the processor.

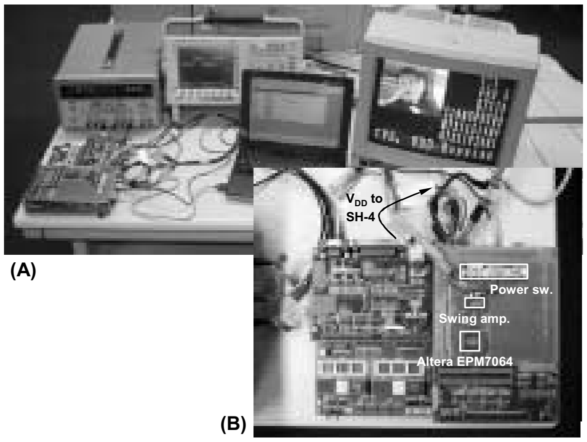

Fig. 3. (A) System. (B) SH-4 embedded system board.

V_{DD} is selected out of 2.0V or 1.2V by power switches located at a V_{DD} supply board. The measured falling and rising time for V_{DD} transition are less than 200µs and 100µs respectively. In order to avoid malfunction, the processor keeps in a sleep mode during the V_{DD} transition. This is carried out by using one of three timers in the processor. Before the V_{DD} transition, the timer is set to expire at the end of the V_{DD} transition and then, the processor is brought to the sleep mode. The timer wakes up the processor by an interrupt when the preset time comes.

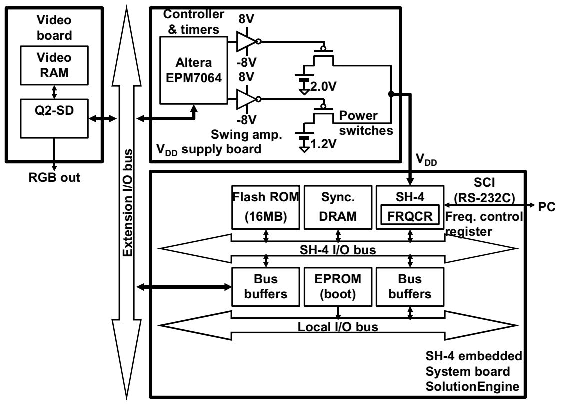

Fig. 4. Block diagram of system board.

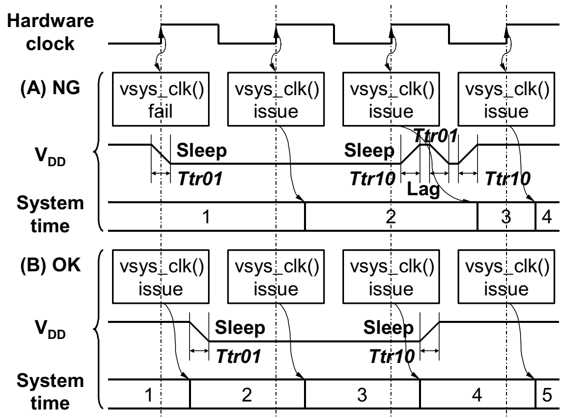

Fig. 5. (A)System clock goes wrong if V_{DD} is changed at will. (B) V_{DD} must be changed just after an timer interrupt service routine (vsys_clk()) has ended.

Since the absolute time called system clock is maintained by the OS whose accuracy is dependent on the time interval between successive interrupts from another timer, care should be taken to change V_{DD}. Specifically, if the V_{DD} transition is allowed without limitation, some of timer interrupts might not be served. If the processor is in a sleep mode due to the V_{DD} transition at the timer interrupts, the consistency of the system clock might be destroyed. Thus, in CVS, the number of the V_{DD} transition is limited to the only one time at every system clock just after the timer interrupt service routine. This ensures the proper system time as shown in Fig. 5.

It should be noted that in the first system clock just after a task is dispatched to a processor, the task could not change V_{DD} at will since V_{DD} in the first system clock stays as it was in the previous system clock. This results from the V_{DD} transition mechanism and task state transition described in the next subsection. The task can change V_{DD} from the next system clock. On the other hand, *f* can be changed anytime unless V_{DD} is 1.2V.

3.2 Software

In order to realize CVS, several modifications are made in the conventional HI7750. First, extended task control block (ETCB) is created to contain specific information while task control block (TCB) contains the conventional information such as priority, start address, and so on. Second, several new system calls are created to support CVS mainly for maintaining and providing virtual deadline, *f*, and V_{DD}. Third, scheduler in the OS is customized to perform necessary action during task state transition. These include managing timing information in the ETCB, computing virtual deadline, and putting the processor into a sleep mode.

3.2.1 ETCB

Each task is associated with the ETCB. Fig. 6 shows a pseudo code of the ETCB.

```
structure ETCB{
    Period;     // Task initiation period
    Tn;         // Next initiation time
    Tsta;       // Time when dispatched
    Texe;       // Time executed already
    Vd;         // Virtual deadline
};
```

Fig. 6. Pseudo code of ETCB structure.

- *Period*; refers to interval in which a task is initiated in principle. *Period* has fixed value and does not changed unlike the other members

- *Tn*; refers to the relative time at which a task should be initiated next time. *Tn* of any task in any state is always decremented by one in every vsys_clk() except for the case where *Tn* has become 0 (*Tn* timeout). In the OS, a new queue called *Tn* queue is adopted to monitor *Tn* timeout where all tasks are sorted in ascending numerical order of *Tn* to easily find *Tn* timeout. If *Tn* timeout happen in the DORMANT queue, the task is initiated.

- *Tsta*; refers to system clock at which a RUN task is dispatched. *Tsta* is valid only when the task is in the RUN state.

- *Texe*; refers to accumulated time that has been already executed. It should be noted that *Texe* is incremented by the difference between system clock and *Tsta* only when the task is preempted. *Texe* is reset when the task is initiated.

- *Vd*; refers to virtual deadline of a RUN task. *Vd* is valid only when the task is in RUN state. *Vd* becomes 0 if there are more than one tasks in the READY queue. On the other hand, if the RUN task is the only in the READY queue, the scheduler takes smaller *Tn* between *Tn* of the RUN task and the smallest *Tn* in the DORMANT queue as *Vd* of the RUN task. Fig. 7 shows how to determine *Vd*. The smallest *Tn* of the tasks in the DORMANT queue can be easily obtained because the tasks are sorted in ascending numerical order of *Tn* as well as the Tn queue. Incidentally, *Vd* is also renewed in every vsys_clk() because *Tn* is renewed.

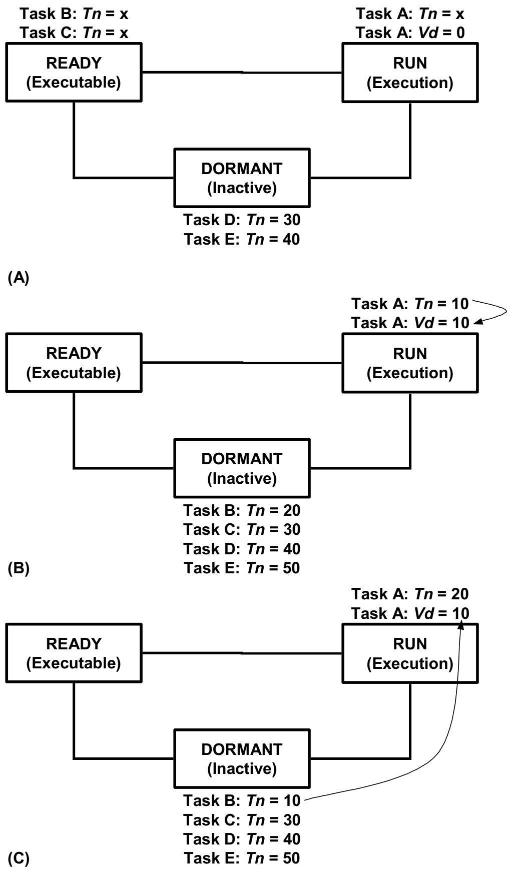

Fig. 7. (A) There are more than one tasks in the READY queue. *Vd* of the RUN task becomes 0 regardless of *Tn* of the tasks in the READY queue. If the RUN task is the only in the READY queue, (B) *Tn* of the RUN task become its own *Vd* in case where *Tn* of the RUN task is smaller than the smallest *Tn* of tasks in the DORMANT queue. (C) In the opposite case, the smallest *Tn* of the tasks in the DORMANT queue becomes *Vd*.

3.2.2 Global symbols

Some global symbols are prepared for the additional system call.

- *G_FRQ_NOW*, *G_FRQ_NEXT*; refer to flags of *f* of the present and the next system clock. *HIGH* (0) means 200MHz and *LOW* (1) means 100MHz.

- *G_VDD_NOW*, *G_VDD_NEXT*; refer to flags of V_{DD} of the present and the next system clock. *HIGH* means 2.0V and *LOW* means 1.2V.

- *G_READY_Q_COUNT*; refers to the number of tasks in the READY queue. This is used when *Vd* is determined. If it is more than one, *Vd* becomes 0.

3.2.3 Additional system calls

There are several system calls to support CVS.

- get_sta_time(); returns *Tsta* of the RUN task as the system clock at which the task has been dispatched.

- get_exe_time(*task_id*); returns *Texe* of the task whose id is *task_id* as the accumulated time that has been already executed before the last preemption.

- get_vd(); returns *Vd* of the RUN task as the virtual deadline.

- get_frq(); returns *G_FRQ_NOW* as a flag of the present *f*.

- get_vdd(); returns *G_VDD_NOW* as a flag of the present V_{DD}.

- set_frq(*Frq*); changes *f* instantaneously by the FRQCR register and set *Frq* to *G_FRQ_NOW* if *Frq* is greater than or equal to *G_VDD_NOW*. In case where *Frq* is less than *G_VDD_NOW*, that is, only in case where *Frq* is *HIGH* (0) and *G_VDD_NOW* is *LOW* (1), *f* cannot be change to 200MHz because the processor cannot run at 200MHz when V_{DD} is 1.2V. set_frq(*Frq*) can be described as shown in Fig. 8 in a pseudo code.

```
set_frq(Frq){
    if(Frq >= G_VDD_NOW){
        if(Frq == HIGH)
            Make f 200MHz by FRQCR register;
        else // if(Frq == LOW)
            Make f 100MHz by FRQCR register;
        G_FRQ_NOW = Frq;
    }
}
```

Fig. 8. Pseudo code of set_frq(*Frq*). This is prepared for flexible power management to decrease only *f* without decreasing V_{DD}.

• set_vdd(*Frq*); just sets *Frq* to *G_FRQ_NEXT* and *G_VDD_NEXT* to change *f* and V_{DD} from the next system clock. This mechanism limits the number of the V_{DD} transition to the only one at every system clock. set_vdd(*Frq*) can be described as shown in Fig. 9 in a pseudo code.

```
set_vdd(Frq){
    G_FRQ_NEXT = Frq;
    G_VDD_NEXT = Frq;
}
```

Fig. 9. Pseudo code of set_vdd(*Frq*). Changing *f* and V_{DD} are actually carried out in the next vsys_clk().

3.2.4 Task states

Tasks are scheduled by the scheduler based on their states as shown in Fig. 10.

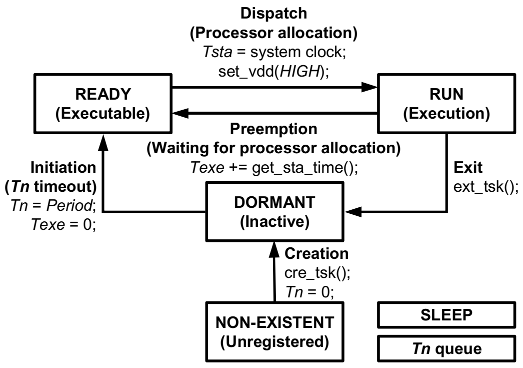

Fig. 10. In CVS, task states are only NON-EXISTENT, DORMANT, READY and RUN. Task transitions are creation, initiation, dispatch, preemption and exit.

• Creation: A task is created in the DORMANT queue by cre_tsk(). The task is then immediately initiated and does not stay in the DORMANT queue because *Tn* is reset.

• Initiation: In vsys_clk(), task initiation is made to monitor *Tn* timeout. If *Tn* has become 0 in the DORMANT queue, the task are initiated. Then, *Period* is set to *Tn* and *Texe* is reset. sta_tsk() is not needed to initiate tasks because tasks are initiated automatically and instantaneously.

• Dispatch: In the READY queue, tasks are sorted in order of priority. The task at the head of the READY queue is dispatched to the processor if there is no RUN task. When the task is dispatched, the system clock is set to *Tsta*. In order to change *f* and V_{DD} from the next system clock, set_vdd(*HIGH*) is called. This is a kind of safety systems keeps the task running at 200MHz to guarantee the real-time feature even if the task is not modified for the application slicing.

• Preemption: *Texe* is incremented by return value of get_sta_time().

• Exit: Tasks are done by ext_tsk().

• Sleep: If there are no tasks to be executed, that is, all tasks are in the DORMANT queue, set_vdd(*LOW*) is called and then processor moves to the sleep mode. This is because lower power can be achieved by decreasing *f* and V_{DD} for sleeping. The processor, however, wakes up in every vsys_clk() to keep the system clock counting and after vsys_clk(), return to the sleep mode again.

3.2.5 vsys_clk()

In CVS, vsys_clk() plays a new role other than the system clock counting. First, *Tn* and *Vd* are renewed and *Tn* timeout is monitored. Then, in order to reflect them, all queues are sorted. After vsys_clk(), the scheduler dispatches the task at the head of the READY queue to the processor.

In case where V_{DD} is changed by accessing one of the V_{DD} switches, there are two conditions, that is, order of changing *f* and V_{DD} is different. We decrease *f* and then V_{DD} in the decreasing case while we increase V_{DD} and then *f* in the increasing case. The processor sleeps masking any other interrupts than the V_{DD} transition timer to eliminate malfunction due to the V_{DD} transition during *Ttr01* or *Ttr10*, which is the falling or rising V_{DD} transition time as shown in Fig. 11. This means that the interrupt level of the V_{DD} transition timer must be the higher than the other interrupt level. vsys_clk() can be described as shown in Fig. 12 in a pseudo code.

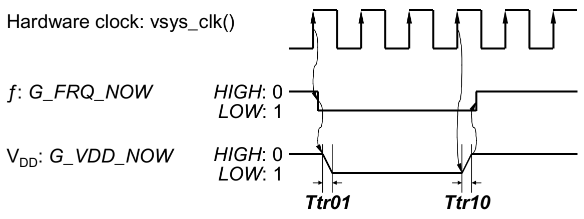

Fig. 11. Order of changing *f* and V_{DD} is different in case of decreasing or increasing them.

```
vsys_clk(){
    Renew system clock, Tn, and Vd
    Sort DORMANT, READY and Tn queues; //Reflect them
    /* Change only f */
    if(G_VDD_NOW == G_VDD_NEXT)
        set_frq(G_FRQ_NEXT);
    else{
        /* Decrease f and then V_DD */
        if(G_VDD_NOW < G_VDD_NEXT){
            /* First, decrease f */
            set_frq(G_FRQ_NEXT);
            /* Then, decrease V_DD */
            Set Ttr01 to V_DD transition timer;
            set_imask(level of V_DD transition timer –1); // Activate timer
            *VDD_SW_ADDRESS = G_VDD_NEXT;
            Sleep;
        }
        /* Increase V_DD and then f */
        else{ // if(G_VDD_NOW > G_VDD_NEXT)
            /* First, increase V_DD */
            Set Ttr10 to V_DD transition timer;
            set_imask(level of V_DD transition timer –1); // Activate timer
            *VDD_SW_ADDRESS = G_VDD_NEXT;
            sleep;
            /* Then, increase f */
            set_frq(G_FRQ_NEXT);
        }
        G_VDD_NOW = G_VDD_NEXT;
        set_imask(0); // Unmask all interrupts
    }
}
```

Fig. 12. Pseudo code of vsys_clk(). set_imask() masks interrupts. V_{DD} is changed by accessing *VDD_SW_ADDRESS*.

4. Experimental Results

In order to demonstrate the feasibility of CVS, we construct a task set that consists of MPEG4 codec and 4096-points fast Fourier transform (FFT). Table 1 summarizes an example of the task set and Table 2 shows characteristics of each slice in the applications. The applications are sliced in the number of the function blocks and loops to be easily able to add code fragments. The period of MPEG4 is set to 114ms to guarantee feasibility of CVS. In the OS, MPEG4 has higher priority.

Fig. 13 shows measured waveforms of V_{DD} and sleep signal of the processor from time 0 up to the lowest common multiple (LCM) of both periods that is equal to 342ms. In the figure, since four falling and four rising V_{DD} transitions are observed, the overhead is less than 1.2ms.

It should be noted that 2.0V is used only during 10.5% of the time on average that gives the average workload of 37.5% (10.5% x 1 + 54% x 0.5 + 35.5% x 0).

TABLE 1. Example of task set.

	MPEG4	FFT
Period	114ms	171ms
Deadline	114ms	171ms
WCET	79ms	35ms
# slices	22	2

TABLE 2. Characteristics of slices.

Applications	# slices	WCET	Function
MPEG4	1	1ms	Initialization
	20	64ms	Macroblock calculation
	1	14ms	Display
FFT	1	2ms	Bit-reversal
	1	33ms	Danielson-Lanczos

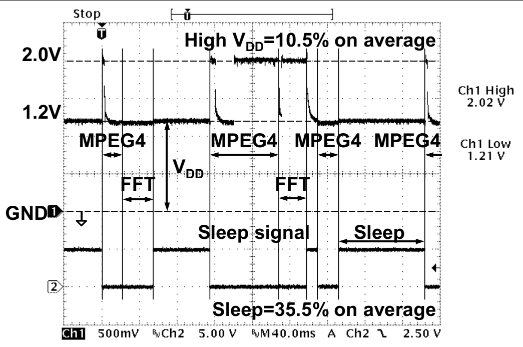

Fig. 13. Measured waveforms of V_{DD} and sleep signal of processor.

The behavior of the measured waveform in Fig. 13 can be explained as follows with the help of Fig. 14. At time 0, MPEG4 is dispatched. *Vd* is set to 0 because FFT is also in the READY queue, which means that MPEG4 should complete its execution within its WCET of 79ms. With low workload for most of slices, MPEG4 completes it at 20ms. At 20ms, FFT occupies the processor while MPEG4 exits to the DORMANT state. *Vd* is set to 114ms that is equal to the next initiation time of MPEG4. Thus, 84ms is allowed to execute FFT of 35ms in the worst case, which means that both slices of FFT can be executed at 1.2V. Upon completion of FFT, the processor goes to the sleep mode until 113ms. At 114ms, MPEG4 is dispatched again with *Vd* of 171ms, which is the next initiation time of FFT. Since the time interval allowed for the execution of MPEG4 (57ms = 171ms – 114ms) is less than its WCET, the advantage of the virtual deadline can not be exploited. The first slice starts its execution at 2.0V and the last slice completes its execution at time 188ms. Since the data for this instance of MPEG4 is close to the worst case, most slices work at 2.0V. The remaining instance of each application can be understood similarly.

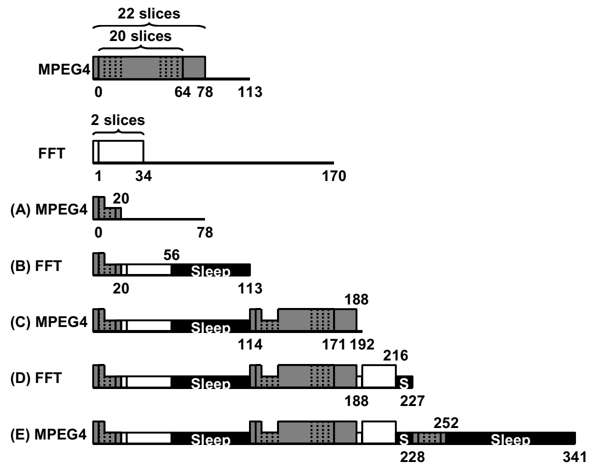

Fig. 14. Example of schedule for MPEG4 and FFT with CVS.

Fig. 15 shows the comparison of the power between CVS and the conventional RMS. In RMS, the processor is assumed to execute NOP when it is not occupied by any task. The processor is measured to consume 0.20W with CVS. Unfortunately, in CVS, I/O buffers of the processor do not work below 1.2V. If the I/O buffers were designed carefully, operation below 0.9V could be achieved instead of 1.2V. In that case, the power would become 0.16W and could be reduce to less than 25% of RMS.

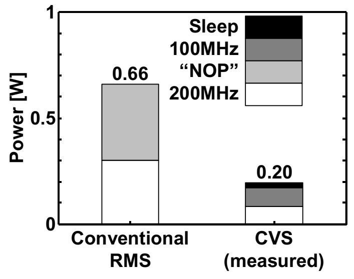

Fig. 15. Comparison of power. The processor consumes 0.8W at 200MHz and 2.0V, 0.16W at 100MHz and 1.2V, and 0.07W in the sleep mode.

Power saving with CVS depends on combination of task periods, which in turn determines how much we can benefit from virtual deadline. It is also dependent on extent of execution time variation. Considering the fact that multimedia applications frequently have many variations due to data-dependent execution and over-estimation of WCET, we can expect significant saving with CVS.

5. Conclusions and Future Work

In this paper, we introduce a cooperative power-optimization method involving design of power conscious OS and development of applications with the concept of application slicing and supporting hardware. CVS obtains power reduction for a processor by exploiting slack times arising from variation of execution time of task instances. We present a run-time mechanism to use the slack times efficiently for the power reduction. The mechanism supports a sleep mode and can change f and V_{DD} dynamically. The experimental results show that CVS obtains significant power reduction across multi-task environment.

As future work, we are studying how to slice multimedia applications for the maximum effect. In addition, influence of unexpected interrupt from a keyboard or Ethernet device in CVS is considered important.

ACKNOWLEDGMENT

The authors would like to thank Dr. K. Sasaki and Dr. K. Ishibashi of Hitachi for fruitful discussion and Mr. H. Yamaki of Hitachi Yonezawa Electronics for the HI7750 modification and helpful advice. This work was supported by Grants from Hitachi and the Japan Society for the Promotion of Science.

REFERENCES

[1] Y. Shin, H. Kawaguchi, and T. Sakurai, "Cooperative Voltage Scaling (CVS) between OS and Applications for Low-Power Real-Time Systems," Proceedings of IEEE Custom Integrated Circuits Conference, pp. 553-556, 2001.

[2] Tron Project Official Home Page, http://www.tron.org.

[3] Hitachi Semiconductor and Integrated Circuits Web Site: http://www.hitachi.co.jp/Sicd/English/Products/micome.htm.

[4] Y. Shin, and K. Choi, "Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems," Proceedings of Design Automation Conference, pp. 134-139, 1999.

[5] S. Lim, Y. Bae, G. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, and C. Kim, "An Accurate Worst Case Timing Analysis for RISC Processors," Proceedings of IEEE Real-Time Systems Symposium, pp. 97-108, 1994.