Runtime Distribution-Aware Dynamic Voltage Scaling

Sungpack Hong, Sungjoo Yoo, Hoonsang Jin,
Kyu-Myung Choi, Jeong-Taek Kong, and Soo-Kwan Eo

CAE Team, System LSI Division, Semiconductor Business,
Samsung Electronics. CO., LTD

{sp7.hong, sungjoo.yoo, hoonsang.jin, kmchoi, jkong, sookwan.eo}@samsung.com

ABSTRACT

We propose a new intra-task dynamic voltage scaling (DVS) method to capture an important fact of 'software runtime distribution' and integrate it into DVS effectively. Specifically, the proposed method finds performance levels, for a given software runtime distribution, i.e. statistical profiling of execution cycles (neither the execution cycle of worst-case execution path nor the worst-case execution cycles of basic blocks), which leads to a minimal energy consumption while satisfying the given deadline constraints. Experimental results report that the proposed method gives 19.2%~33.3% further energy reduction compared with the best-known methods for two industrial multimedia software programs, H.264 decoder and MPEG4 decoder.

1. INTRODUCTION

Dynamic voltage scaling (DVS) is a well-known and effective method for low-power design, especially lowering the processor energy consumption in battery-powered mobile devices. In DVS, the frequency and voltage of a processor are adjusted dynamically, in the middle of execution, to their lowest possible level while meeting the deadline constraints of executing software (SW) program, or task. This performance adjustment is possible when a slack (= deadline – completion time at current performance level) becomes available, i.e. becomes positive in value. Therefore, one of key problems in DVS is to determine a performance level to maximize energy savings without violating the deadline constraints when a slack becomes available.

Most of the previous DVS methods can be classified into two groups in terms of the granularity of performance setting period[1]: inter-task and intra-task DVS. In inter-task DVS, the voltage and the frequency are adjusted only once at the beginning of each task activation assuming the worst-case execution time (WCET) of the task [1-2, 12-13]. On the other hand, intra-task DVS utilizes the slack more aggressively by further adjustments of performance levels in the middle of the execution of the task [3-6]. In this paper, we focus on intra-task DVS.

Existing intra-task DVS methods exploit the fact that as SW execution advances, the remaining workload of SW program may become smaller than observed in the worst case. At each performance setting point in the SW code, the methods determine the minimum processor performance level from the pre-calculated prediction value of remaining workload and the value of current slack while meeting the given deadline constraints. The more accurate the prediction is (i.e. close to the real workload), the more energy gain can be obtained from DVS. Thus, the accurate prediction of workload is required to make the best use of DVS capability**.**

Observation: SW Runtime Distribution

To predict the remaining execution cycles of workload accurately, two factors need to be considered. One is control dependency caused by control statements, e.g. if/else. The variation of execution paths to be taken in the remaining run of SW program may result in huge impact in workload prediction. The other is data and architecture-dependency. To be more specific, we need to take into account the effects of data access patterns (e.g. delay variation due to hit/miss in cache/TLB/write buffer, off-chip memory access delay variation due to the page access locality in the memory controller, etc.) and the effects of dynamic states in the hardware architecture (e.g., branch prediction, out-of-order execution, etc.).

ICCAD'06, November 5-9, 2006, San Jose, CA

[1] **A performance setting point** is a place in the SW code where the performance level of processor is adjusted. The **performance setting period** is the time duration between two performance setting points in the SW code.

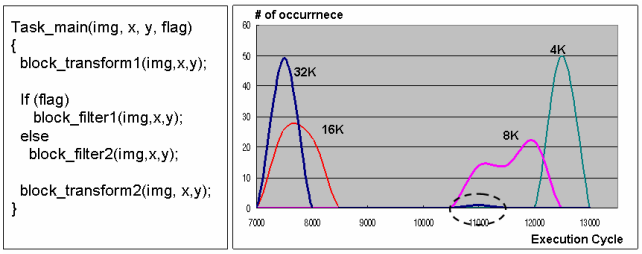


Figure 1. An example of SW runtime distribution

Figure 1 gives an example of SW runtime distribution caused by the data and architectural effects. Figure 1 (a) shows a function consisting of four basic blocks each of which is implemented as a macro[2]. We ran this code on an ARM926 processor while changing the size of I/D cache from 4KB to 32KB. Figure 1 (b) gives the statistics of execution cycle for the case of macro **block_filter1** (with total 60 times of execution[3]). For instance, in the case of 32KB cache, the macro gives about 7500 clock cycles as the execution cycle 50 times among total 60 times of execution.

As shown in Figure 1 (b), the shape of the histogram changes as the cache size changes. It is interesting to see the long right tails (denoted with a dashed oval in the figure) that belong to the cases of cache size 16KB and 32KB. Related with cache behavior, such long tails can be caused by well-known cache miss sources: cold miss (when the first execution of the macro), conflict miss (when the cache lines containing the code/data of the macro are replaced by the intervening code executed between previous and current executions of the macro), and capacity miss (when the macro handles bigger code/data than the cache size). Due to such long tails, the WCEC (worst-case execution cycles) of block_filter1 becomes much greater than the average execution cycle, especially, in the case of cache size 16KB and 32KB.

Limitation of Existing Intra-task DVS Methods

Most of existing intra-task DVS methods [4-6] focus only on control dependency in predicting the remaining workload. However, as shown in Figure 1, the data access patterns and the dynamic states of underlying hardware architecture can be more crucial in the prediction. Even the same execution path taken on the same processor can have significantly different execution cycles (per each execution of the path) depending on the data access patterns and the dynamic states of hardware architecture. Thus, it is required to exploit the data and architecture dependency to better predict the remaining workload thereby maximizing energy reduction.

[2] In the case of multimedia software, it is usual to unroll compute-intensive loops for performance reasons thereby giving big basic blocks. The code details of the macro used in the figure is available at appendix A.

[3] In order to normalize the frequency, we use probability distribution function to be explained in Section 3.

When predicting the remaining workload at a given performance setting point, existing intra-task DVS methods based on execution paths (which we call *path-based methods*) perform the predictions based only on the WCEC's of remaining execution paths (or basic blocks). For instance, in the case of cache size 16/32KB, the WCET of macro block_filter1 may be estimated to be greater than 11,000 cycles while its execution cycle ranges between 7,000 and 8,500 cycles in most cases. Thus, path-based DVS methods may suffer from the error in workload prediction causing less energy reduction than possible otherwise.

Our Contribution

In this paper, we propose a new DVS method which exploits not only execution path-dependent runtime distributions but also data/architecture-dependent ones. Compared to existing path-based intra-task DVS methods which exploit the execution path-dependent runtime distribution only with a worst-case assumption for the data and architecture-dependent runtime distribution, the proposed method gives less energy consumption.

2. RELATED WORK

Lee and Sakurai proposed an intra-task DVS method [3] which exploits workload variation to reduce the energy consumption in multimedia SW program execution. In their work, the workload variation is a slack which is calculated as the difference between expected and real program runtimes of already executed SW code at a performance setting point. For the remaining workload, this work assumes the WCEC.

Azevedo *et al.* proposed a method that adjusts the performance level at performance setting points during SW run [4]. In their work, the predicted remaining workload is the execution cycle of worst-case execution path from a performance setting point to the end of SW program run. Shin *et al.* presented a method of predicting the remaining workload to be the execution cycle of the average-case (i.e. the most frequently taken) execution path [5]. Seo *et al.* presented a concept of virtual execution path to find an optimal performance level for minimum average energy consumption [6]. The method also uses the WCEC's of remaining basic blocks to predict the remaining workload. Some of recent studies apply data flow analysis to path-based intra-task DVS in order to set performance levels more aggressively (i.e. at earlier program points than in previous methods) by identifying the earliest program points where the remaining execution path is determined [14][15].

There has been little work on exploiting the information of runtime variation in DVS. Hua *et al.* presented a method that exploits the information of runtime distribution in task execution times to trade off between energy reduction and timing violation while giving probabilistic guarantees of completion ratio in the case of soft real-time systems [7]. Recently, S. Yaldiz, *et al.* presented an inter-task DVS method of exploiting the information of correlation among the runtime distributions of tasks [8]. Although this work

focuses originally on inter-task DVS with static scheduling, it may also be very effective in the case of strong correlation among big code sections in a task. However, when applied to intra-task DVS, the work may suffer from an exponential complexity while ours has a polynomial complexity. We will give a more detailed comparison between the work in [8] and ours in Section 3.7.

3. Proposed Method

When applying DVS to real variable voltage processors, there are several problems as follows.

- How to select performance setting points?

- How to adjust the performance level at a performance setting point?

- How to take into account the overhead (runtime and energy consumption) of voltage and frequency transition?

- How to handle discrete voltage levels?

In this paper, our main concern is to tackle the second problem, to find a performance setting method. However, when applying the method, we will explain the practical consideration, regarding the other three problems, which we made in our work.

3.1 Terminology and Assumptions

We define our terminology in this subsection.

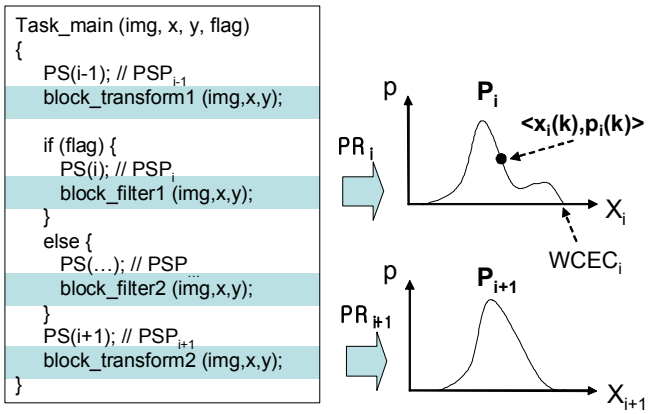


Figure 2. Histograms of performance regions

- **Performance setting point (PSP)**: A code location where the performance level of processor is adjusted. In the left-hand side of Figure 2, each basic block (denoted with a shaded area) has a performance setting point denoted with function PS().

- **Performance region (PR)**: A code section which starts with a single PSP in the beginning and ends before another PSP belonging to another code section. In Figure 2, each basic block corresponds to a PR. However, in reality, a PR will be a complex code section (including control statements such as if/else) bigger than a single basic block. Note that a PSP belongs only to a single PR.

- **Histogram of PR (P)**: The normalized histogram (i.e. probability distribution function) of execution cycles of PR. The right-hand side of Figure 2 shows the histogram, P_i, which corresponds to the histogram of macro block_filter1 denoted with PR_i. In reality, we quantize the histogram into a set of M tuples $<X_i(k),p_i(k)>$, $1 \le k \le M$, where $X_i(k)$ is the k-th quantized value of execution cycle for PR_i and $p_i(k)$ is the probability that the execution of PR_i actually amounts to $X_i(k)$. The maximum value of $X_i(k)$'s is therefore worst-case execution cycle as indicated by $WCEC_i$ in the figure.

- **Leaf, non-leaf and root PR**: A PR is defined as a leaf PR if no other PR is executed after it; otherwise it is defined as a non-leaf PR. The PR that contains program entry point is called as the root PR.

Assumption of Calculating Energy Consumption: $E \sim f^2 * n_{total}$

We assume that the amount of energy consumption E during an execution of one PR is expressed as $E \sim f^2 * n_{total}$ as in [6], where f is the performance level (i.e. clock frequency) set for this PR and n_{total} is the total number of execution cycles of that PR. The justification is as follows: First, from the equation $E \sim V_{dd}^2 * f * t$, we can substitute execution time t as n_{total}/f, since f is constant during a PR. Second, the clock frequency f is known to be related to the supply voltage as $f \sim (V_{dd} - V_T)^{\alpha} / V_{dd} \sim V_{dd}$, where V_{dd} is the supply voltage, V_T is the threshold voltage and α ($1.4 < \alpha < 2$) is the velocity saturation index. Practically, α is considered to be 2 and f is shown to be proportional to V ($f \sim V_{dd}$) [18]. Consequently, $E \sim V_{dd}^2 * f * t \sim f^2 * f * (n_{total}/f) \sim f^2 * (n_{total})$.

3.2 Workload Prediction with the Information of Runtime Distribution: Basic Idea

In this subsection, we explain our basic idea of workload prediction. Figure 3 shows an example of the case of a program with two PR's (denotes as PR_i and PR_{i+1} in Figure 2). Here, the rectangles represent the PRs and the arrow represents the precedence relation between PR's; the execution of PR_{i+1} starts after PR_i finishes its execution. The figure also shows that the two PR's have their own histograms, P_i and P_{i+1}.

Our framework is based on the legacy two-step approach: i) during the design time we determine w_i, the prediction of remaining workload to the end of program, for each PR_i, and ii) during the runtime whenever SW execution reaches PSP_i, the processor's performance level is adjusted using w_i and T_i, the remaining time at that moment: $f_i = w_i / T_i$.[4] However, in this framework our objective is to select w_i values for all PR_i such that average energy consumption for whole program execution is to be minimized, while taking it into

[4] V will be adjusted proportionally to f. For instance, assuming that f_{max} = 500MHz, V_{max} = 1.0V, T=0.4sec, and w_i=100M cycles, then f_i = (100M cycles / 0.4sec) = 250MHz, and V = 0.5V.

consideration the known runtime distribution of each PR.

Before proceeding, note that the workload prediction is trivial for a leaf PR; it must be equal to its WCEC otherwise the deadline is not met for the worst-case. Consequently, when there is only one PR for a program, the performance level will be adjusted to f = WCEC / T at the program entry where T denotes the deadline or timing constraint, which is identical to applying inter-task DVS methods. We can normalize the timing constraint T to unity[5] in following discussions for the brevity of equations without affecting the result.

Now, the problem of workload prediction for the program having two PRs is defined as follows.

Problem of Workload Prediction: *The problem is to predict the remaining workload w_i, of program region PR_i, which gives the minimum total average energy consumption, given w_{i+1}, P_i and P_{i+1}.*

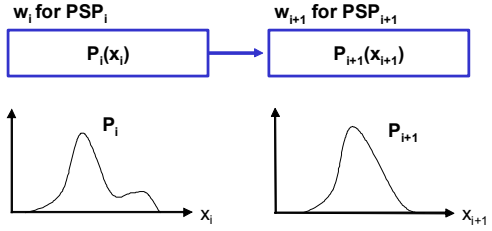


Figure 3. Two performance regions with runtime distributions

First, the total energy consumption is expressed as follows.[6]

$$E(x_i, x_{i+1}, w_i) = f_i^2 x_i + f_{i+1}^2 x_{i+1} = w_i^2 x_i + \frac{w_{i+1}^2 x_{i+1}}{r_i^2} \quad (1)$$

$$r_i = 1 - \frac{x_i}{w_i} \quad (2)$$

where x_i and x_{i+1} are the random variables of distribution functions P_i and P_{i+1}, respectively. As explained earlier, w_{i+1} is $WCEC_{i+1}$ here.

In the previous work of intra-task DVS, w_i is predicted to be the execution cycle of remaining worst-case execution path [4], that of average case execution path [5] ([4] and [5] give the same result in this example since there is only one remaining execution path) or that of virtual execution path [6] (in this case, the prediction gives $WCEC_i + WCEC_{i+1}$).[7]

Assuming that the two histograms are independent of each other, Eqn. (3) gives the average of total energy consumption for P_i and P_{i+1}.

$$\begin{aligned}\overline{E}(w_i) &= \iint E(x_i, x_{i+1}, w_i) p_i p_{i+1} dx_i dx_{i+1} \\ &= w_i^2 \int x_i p_i dx_i + w_{i+1}^2 \int x_{i+1} p_{i+1} dx_{i+1} \int \frac{p_i}{r_i^2} dx_i \\ &= w_i^2 \overline{x_i} + w_{i+1}^2 \overline{x_{i+1}} \int \frac{p_i}{r_i^2} dx_i \\ &= w_i^2 \overline{x_i} + w_{i+1}^2 \overline{x_{i+1}} \sum_{K=1}^{M} \frac{p_i(k)}{(1 - X_i(k)/w_i)^2}\end{aligned} \quad (3)$$

where $\overline{x_i}$ and $\overline{x_{i+1}}$ are the average execution cycles of PR_i and PR_{i+1}, respectively.

What Eqn. (3) means is that the average of total energy consumption is a function of w_i only. Figure 4 illustrates one instance of this function (obtained from the case of H.264 decoder SW in our experiments) Note the optimal prediction of w_i, w_i^{opt} corresponds to the inflection point in the curve of Figure 4.

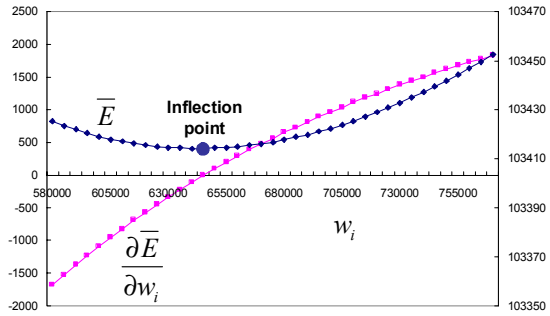


Figure 4. Total average energy consumption w.r.t. w_i

We can calculate w_i^{opt} by solving the following differential equation:

$$\frac{\partial \overline{E}}{w_i} = 0 \quad \Rightarrow \quad \overline{x_i} + w_{i+1}^2 \overline{x_{i+1}} \cdot \left[\sum_{K=1}^{M} \frac{-X_i(k) p_i(k)}{(w_i - X_i(k))^3} \right] = 0 \quad (4)$$

Note that the only variable in Eqn. (4) is w_i. Solving Equation (4) requires a numerical analysis. Since the curve of average energy consumption, as illustrated in Figure 4, is a convex[8], we can apply the Bisection method to solve Eqn. (4). Remind that this numerical analysis is performed at design time. The result of the numerical analysis, i.e. w_i^{opt} is saved in a table and used to set the performance level to the optimal level, f_i^{opt} ($= w_i^{opt} / T$) during runtime.

[5] Also, maximum frequency f_{max} and maximum voltage V_{max} are normalized to unity.

[6] $f_i = w_i / T_i$, $f_{i+1} = w_{i+1} / T_{i+1}$, $T_i = T$, $T_{i+1} = T_i - x_i / f_i$, and T normalized to 1.

[7] If there are no runtime variations for the PRs, all these values become identical.

[8] The derivative of the left-hand side of Eqn. (4) is always positive, meaning that Eqn. (3) is always a convex.

3.3 Workload Prediction with Cascaded PR's

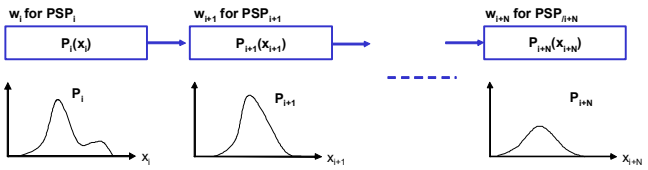


Figure 5. Cascaded PR's

Now, we will extend our discussion to the case of more than two cascaded PR's. This situation is depicted in Figure 5. We use mathematical induction letting our discussions in section 3.2 be the induction basis. And the induction hypothesis is that we already have all the average-energy-optimal workload predictions for the following PRs after PR_i (i.e we have values from w_{i+1} to w_{i+N}). Our goal is again to find w_i that minimizes average energy consumption.

In this case, the total energy consumption is calculated as in Eqn. (5).

$$E(x_i, x_{i+1}, \ldots, x_{i+N}, w_i) = w_i^2 x_i + \frac{w_{i+1}^2 x_{i+1}}{r_i^2} + \frac{w_{i+2}^2 x_{i+2}}{r_i^2 r_{i+1}^2} + \cdots + \frac{w_{i+N}^2 x_{i+N}}{r_i^2 r_{i+1}^2 \cdots r_{i+N}^2} \quad (5)$$

Assuming the independence among w_i's and P_i's, the average energy consumption is calculated as follows.

$$\begin{aligned}\overline{E}(w_i) &= \int\int\cdots\int E(x_i, \cdots, x_{i+N}, w_i) p_i \ldots p_{i+N} dx_i \ldots dx_{i+N} \\ &= w_i^2 \overline{x_i} + w_{i+1}^2 \overline{x_{i+1}} \cdot \int \frac{p_i}{r_i^2} dx_i + w_{i+2}^2 \overline{x_{i+2}} \cdot \int \frac{p_i}{r_i^2} dx_i \cdot \int \frac{p_{i+1}}{r_{i+1}^2} dx_{i+1} \\ &\quad \cdots w_{i+N}^2 \overline{x_{i+N}} \cdot \int \frac{p_i}{r_i^2} dx_i \cdot \int \frac{p_{i+1}}{r_{i+1}^2} dx_{i+1} \cdots \int \frac{p_{i+N}}{r_{i+N}^2} dx_{i+N} \\ &= w_i^2 \overline{x_i} + Z_i \cdot \int \frac{p_i}{r_i^2} dx_i \\ &= w_i^2 \overline{x_i} + Z_i \cdot \sum_{k=1}^{M} \frac{p_i(k)}{(1 - X_i(k)/w_i)^2}\end{aligned} \quad (6)$$

where Z_i is defined as in Eqn. (7)

$$\begin{aligned}Z_i &= w_{i+1}^2 \overline{x_{i+1}} + w_{i+2}^2 \overline{x_{i+2}} \cdot \int \frac{p_{i+1}}{r_{i+1}^2} dx_{i+1} \cdots \\ &\quad + w_{i+N}^2 \overline{x_{i+N}} \cdot \int \frac{p_{i+1}}{r_{i+1}^2} dx_{i+1} \ldots \int \frac{p_{i+N}}{r_{i+N}^2} dx_{i+N}\end{aligned} \quad (7)$$

From the induction hypothesis, Z_i can be calculated. Therefore, we can again set up a differential equation to find w_i which minimizes average energy consumption in the case of cascaded PRs, as Eqn (8). Eqn (8) can be solved using the same technique as in Eqn (4).

$$\frac{\partial \overline{E}}{w_i} = 0 \quad \Rightarrow \quad \overline{x_i} + Z_i \cdot \left[\sum_{K=1}^{M} \frac{-X_i(k) p_i(k)}{(w_i - X_i(k))^3} \right] = 0 \quad (8)$$

Note that this inductive way of proof naturally gives a recursive way of solving, which will be presented in section 3.5.

3.4 Workload Prediction with Control Flow

Conditional statements (e.g. if/else) can also be handled by introducing branch probabilities into Eqn. (4) and (8). Figure 6 exemplifies a case of PR's with two conditional branches.

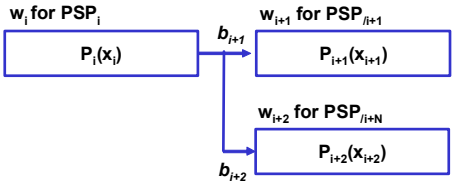


Figure 6. PR's with conditional branches

The two branches (arrows in the figure) are denoted with branch probabilities, b_i and b_{i+1}, respectively. Eqn. (9) gives the total energy consumption in this case. Note that the branch probabilities are already obtained from the profiling data. Thus, they are constant values.

$$E(x_i, x_{i+1}, x_{i+2}, w_i) = w_i^2 x_i + b_{i+1} \frac{w_{i+1}^2 x_{i+1}}{r_i^2} + b_{i+2} \frac{w_{i+2}^2 x_{i+2}}{r_i^2} \quad (9)$$

The predicted workload w_i^{opt} is then obtained by solving the differential equation as follows.

$$\frac{\partial \overline{E}}{w_i} = 0 \quad \Rightarrow \quad \overline{x_i} + [b_{i+1} w_{i+1}^2 \overline{x_{i+1}} + b_{i+2} w_{i+2}^2 \overline{x_{i+2}}] \cdot \left[\sum_{K=1}^{M} \frac{-X_i(k) p_i(k)}{(w_i - X_i(k))^3} \right] = 0 \quad (10)$$

Eqn. (10) can be solved in the same way as (4) and easily extended to the cases that there are multiple (>2) conditional branches. Loops are handled in the same way as in [6]. Thus, we group K iterations of a loop into a single PR or fully expand it.

3.5 Workload Prediction: Summary

We calculate w_i^{opt} of each PR in a recursive way starting from a leaf PR up to the root PR. Figure 7 summarizes the procedure of calculating w_i^{opt}. In Figure 7, $WCEC_{i,total}$ is the clock cycle of the remaining worst-case execution path from PSP_i to the end of program which serves as a boundary for the bisection method.

It is also notable that if we assume all the PRs have no distributions at all, our method gives exactly the same solution as in [6], which is proven to be optimal under that assumption. The detailed proof is omitted here.

• For leaf PR_n:

1) $w_n \leftarrow WCEC_n$

2) $C_n \leftarrow w_n^2 \cdot \overline{x_n}$

• For non-leaf PR_i:

1) $Z_i \leftarrow \sum_{j}^{children} b_j C_j$, where b_j is the branch probability to child PR_j

2) Using bisection, in range ($WCEC_i$ ~ $WCEC_{i,total}$) find w_i such that

$$\overline{x_i} + Z_i \cdot \left(\sum_{k=1}^{M} \frac{-X_i(k)p_i(k)}{(w_i - X_i(k))^3}\right) = 0$$

3) $$C_i \leftarrow w_i^2 \overline{x_i} + Z_i \cdot \sum_{k=1}^{M} \frac{w_i^2 p_i(k)}{(w_i - X_i(k))^2}$$

Figure 7. Calculation of w_i^{opt} in a recursive way

3.6 Satisfying the Given Timing Constraint

When setting the performance level, we need to check the feasibility, that is, to check whether the performance setting will satisfy the given timing constraint T. To do that, first we need to take into account the case that the worst-case behavior occurs, i.e. the WCEC of PR_i happens to occur. In addition to that, we need to include the runtime overhead caused by voltage/frequency transition. Figure 8 illustrates all the factors to be counted in the check, and Inequality (11) shows the feasibility check.[9]

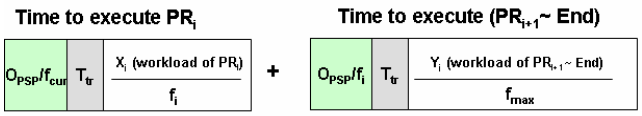


Figure 8. Checking the feasibility of performance setting

$$\left(\frac{O_{PSP}}{f_{cur}} + T_{tr}(f_{cur}, f_i) + \frac{WCEC_i}{f_i}\right) + \left(\frac{O_{PSP}}{f_i} + T_{tr}(f_i, f_{\max}) + \frac{WCEC_{i,total} - WCEC_i}{f_{\max}}\right) \le T_i \quad (11)$$

, where f_{cur} is the clock frequency just before PSP_i. O_{PSP} is the delay overhead (in clock cycles) of performance setting function, PS() as illustrated in Figure 2. The delay overhead of voltage/frequency transition, $T_{tr}(f_{old}, f_{new})$ depends on two frequencies, f_{old} and f_{new}. In reality, O_{PSP} and T_{tr} have also runtime variations. In our experiments, we assumed that they have their worst-case delay values.

After calculating f_i^{opt}, Inequality (11) ($f_i = f_i^{opt}$) checks to see if the timing constraint T is satisfied in the case of setting performance level to f_i^{opt} even when the execution cycle of PR_i happens to be $WCEC_i$. In the cases that Inequality (11) is not satisfied, we set f_i to the minimum value satisfying the Inequality.

3.7 Limitations, Complexity and Future Work

The proposed method has limitations as other existing intra-task DVS methods [3-6] do. To achieve energy-optimality for the multi-task environment, there must be some improvements to our method: i) the integration with inter-task DVS methods as in [16] and ii) accurate estimation of the preemption (especially, cache-related) delay in determining a performance level at a performance setting point [17].

Our method has also the limitation of existing profile-based intra-task DVS methods [5,6,8]: the dependency of efficiency in energy reduction on the profiled information. For our future work, we will work on this issue to improve the efficiency, e.g. by runtime profiling.

In terms of exploiting SW runtime distribution, there is a similar approach in [8]. However, the method in [8] has an exponential complexity, $O(L^P)$ where L is the number of voltage/frequency levels and P is that of performance setting points. Thus, it may be infeasible to be applied to intra-task DVS problems when P is a large number. In our experiments of a H.264 decoder SW, L is 16 and P can be 212 (maximum case). In this case, the method is not likely to find a feasible solution in a reasonable time. The complexity of our method is O(M*P) where M is the number of quantization points in the histogram.

In Eqn (3) and (6), we assume that there is no correlation between two PR's. However, in reality, there can be correlations, especially when cache behavior is concerned. Our future work includes the investigation and exploitation of the correlation among arbitrary code sections, which will be more difficult to identify than in the case of task-level correlation in [8].

3.8 Notes for Practical Application

In spite of aforementioned limitations, our method still has advantages for practical application. Most of all, ours is not only be applicable to basic-block PRs as in [5-6, 14-15], but also can be applied in a coarse-grained way [10] without violating its assumptions. This is a big advantage since real applications can easily have millions of distinguished basic blocks[11].

Also, in a multi-task environment, ours can be orthogonally applied to kernel-level inter-task DVS; kernel adjusts frequency at normal task entries while an intra-DVS-enabled task (IDET) changes frequency autonomously during its execution. If an IDET is interrupted, the frequency gets back to kernel-set level, and when execution is returned back to the IDET, it will adjust the frequency at the next PSP

[9] If you scan for f which satisfies $X_i/f + Y_i/f_{max} = T_i$ under $X_i \le WCEC_i$, $Y_i \le WCEC_{i+1,total}$, and $X_i + Y_i \le WCEC_{i,total}$, the maximum value of f (= minimally required f_i) is found at $X_i = WCEC_i$ and $X_i + Y_i = WCEC_{i,total}$.

[10] PR is not restricted to a single basic block, but can be complex routines. e.g. a series of function calls.

[11] All the activations of basic blocks in a loop or multiply called functions should be distinguished.

regarding updated remaining time.

4. EXPERIMENTS

In our experiments, we build a target architecture consisting of an ARM926 (16K I$/D$), AHB bus, a memory controller and SDRAM (off-chip). The maximum supply voltage of the processor is 1.2v. The frequency ranges between 120 MHz and 480 MHz with 12 discrete voltage/frequency levels. In our experiments, a performance level, which is the lowest but higher than or equal to f_i^{opt} , is selected as a performance level from a given set of discrete frequency.

The delay overhead of performance setting function call, O_{PSP} and that of voltage/frequency transition are assumed to be 1k cycles and 30μs, respectively. We assume also that the energy overhead of voltage/frequency transition is that of processor execution at the higher frequency of the two frequency levels (f_{old} and f_{new}).

For the collection of runtime distribution, i.e. histograms, we set up a transaction-level cycle-accurate simulation model of the target architecture with a commercial environment, MaxSim [9] and built a proprietary C source-level profiler of SW execution cycles on top of the environment. The profiling takes about one hour on a PC (Pentium 4, 2.4GHz, 764MB) in running 100 frames of MPEG4 decoding.

H.264 Decoder Case: We applied the proposed method to an industrial H.264 decoder (QCIF, 30fps)[12]. Figure 9 (a) shows the code structure of H.264 decoder. The rectangles in the figure correspond to the performance regions, PR's. Each PR is denoted with its average runtime at f_{max}. Each of functions, MB-decode and Deblock consists of a loop with 99 iterations. Inside of each of the two functions, we make PR's by grouping K iterations of the loop into a PR. We collected the profiling data of runtime distribution by running the H.264 decoder with the H.264 compliance test pictures [10] as the input[13].

Figure 9 (b) shows the energy consumption as we increase the number of PR's (or PSP's) by changing the K value in the grouping of loop iterations. Since the H.264 code has two loops in the two functions, MB decode and Deblock as shown in Figure 9 (a), we changed the number of PSP's in each loop from 1 to 99 (99 to 1 in terms of K value). The X-axis of Figure 9 (b) shows the number of PSP's in the function MB-decode (L1 as denoted in Figure 9 (a)). In the figure, the two different sets of data correspond to the two different numbers of PR's in function Deblock (L2 as denoted in Figure 9 (a)). In Figure 9 (b), **Inter** represents the case that inter-task DVS is used. Thus, the WCET of the entire program is set to be the performance level in the beginning of the H.264 code and the performance level does not change during the execution. **WT** and **Wn** represent the methods based on remaining worst-case execution path [4][14] and average-energy execution path [6], respectively.

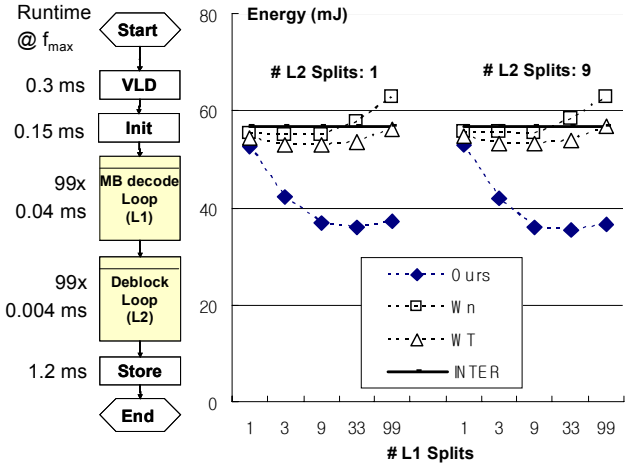


Figure 9. H.264 decoder software case

As shown in Figure 9 (b), our method outperforms existing ones for any combination of K values of L1 and L2. Table 1 summarizes the best cases of ours and existing methods. Ours gives the minimum energy reduction 35.27mJ at (33, 9) which corresponds to 33.3% reduction compared with the best result of existing methods, 52.84mJ at (9,1) with **WT**. The column **Reduction** shows the energy reduction compared with the case that DVS is not applied, i.e. after the completion of one frame, the processor is turned off until the start of the next frame.

Table 1. Comparison of energy consumption: H.264 decoder

	Loop Split (L1, L2)	Energy (mJ)	Reduction (%)
Ours	(33, 9)	35.27	47.5
Inter	(1, 1)	56.77	15.5
WT [4]	(9, 1)	52.84	21.4
Wn [6]	(9, 1)	54.89	18.4

Table 2. Comparison of energy consumption: MPEG4 decoder

	Loop Split (L1, L2)	Energy (mJ)	Reduction (%)
Ours	(11, 9)	26.18	75.8
Inter	(1, 1)	91.36	15.5
WT [4]	(12, 18)	32.41	70.0
Wn [6]	(9, 9)	34.32	68.2

MPEG4 Decoder Case:

We applied the proposed method to an industrial MPEG4 decoder (CIF, 60fps). The code structure of MPEG4 is similar to that of H.264 decoder shown in Figure 9 (a). We collected the profiling data of runtime distribution by running

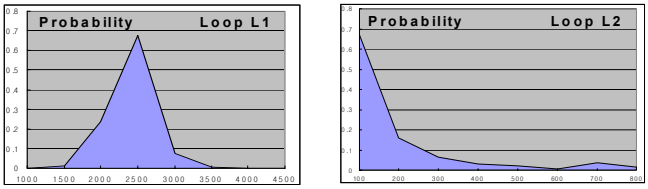
[12] The H.264 and MPEG4 decoder programs are optimized at source/assembly level by designers at Samsung Electronics.

[13] We used four test pictures, NL1_SONY_D, SVA_BA2_D, NLMQ1_JVC_C, and MR1_MW_A.

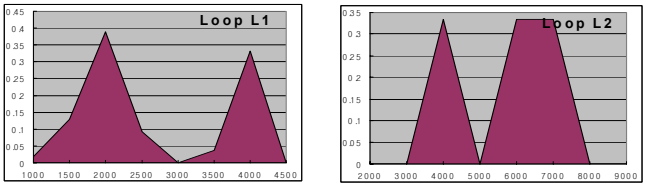
[14] In this example, the average-case execution path [5] corresponds to the worst-case execution path [6] at performance setting points since the H.264 code has two loops whose iterations are fixed to 99 times.

the MPEG4 decoder with 150 frame test pictures. Table 2 gives the comparison of energy consumption. Ours gives the minimum energy reduction 26.18mJ at (11,9) which corresponds to 19.2% reduction compared with the best result of existing methods, 32.41mJ at (12,18) with **WT**.

Figure 10 explains why the proposed method gives less energy reduction in the MPEG4 decoder than in the H.264 decoder. The figure shows the histogram of each of the two loops, L1 and L2, in both programs. As the figure shows, the runtime distribution of H.264 decoder is bigger than that of MPEG4 decoder. Especially, loop L2 of H.264 has a long tail in the runtime distribution, which prevents the existing methods from aggressively reducing the performance level while the proposed method does. Thus, the proposed method showed more relative effectiveness in the H.264 decoder than in the MPEG4 decoder.



(a) H.264 runtime distribution



(b) MPEG4 runtime distribution

Figure 10. Runtime distributions of H.264 and MPEG programs

5. CONCLUSION

In this paper, we proposed a new intra-task DVS method to reduce energy consumption by exploiting the information of software runtime distribution. The key parts of the proposed method are i) the introduction of the concept of 'statistical profiling of execution cycles' into intra-task DVS rather than worst-case execution cycles, and ii) the analytic solution of finding energy optimal performance settings. The experiments show the effectiveness (19.2% ~ 33.3% energy reduction) of the proposed method in two industrial multimedia applications.

6. REFERENCES

[1] D. Kwon and T. Kim, "Optimal Voltage Allocation Techniques for Variable Voltage Processors", DAC, 2003.

[2] R. Jejurikar and R. Gupta, "Dynamic Slack Reclamation with Procrastination Scheduling in Real-time Embedded Systems", DATE, 2005.

[3] S. Lee and T. Sakurai, "Run-time Voltage Hopping for Low-Power Real-Time Systems", DAC, 2000.

[4] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau, "Profile-Based Dynamic Voltage Scheduling Using Program Checkpoints", DATE, 2002.

[5] D. Shin, J. Kim, and S. Lee, "Low-Energy Intra-Task Voltage Scheduling using Static Timing Analysis", DAC, 2001.

[6] J. Seo, T. Kim, and K. Chung, "Profile-Based Optimal Intra-Task Voltage Scheduling for Hard Real-Time Applications", DAC, 2004.

[7] S. Hua, G. Qu, and S. Bhattacharyya, "Energy Efficient Multi-Processor Implementation of Embedded Software", Int'l Workshop on Embedded Software, Oct. 2003.

[8] S. Yaldiz, A. Demir, S.Tasiran, P.Ienne, and Y. Leblebici, "Characterizing and Exploiting Task-Load Variability and Correlation for Energy Management", ESTiMedia, 2005.

[9] MaxSim Technology, ARM Co. Ltd, available at http://www.arm.com/products/DevTools/.

[10] "Conformance specification for H.264 advanced video coding", http://ftp3.itu.ch/av-arch/jvt-ite/draft_conformance/00readme_suz0420.html

[11] ARM IEM Technology, available at www.arm.com

[12] F. Yao, A. Demers, and S. Shenker, "A Scheduling Model for Reduced CPU Energy", Symposium on Foundations of Computer Science, 1995.

[13] G. Guan and X. Hu, "Energy Efficient Fixed-Priority Scheduling for Hard Real-Time Systems", DAC, 2001.

[14] D. Shin and J. Kim, "Optimizing Intra-Task Voltage Scaling Using Data Flow Analysis", ASP-DAC 2005.

[15] D. Shin and J. Kim, "Look-Ahead Intra-Task Voltage Scaling Using Data Flow Information, Int'l SoC Design Conference, 2004.

[16] J. Seo, T. Kim, and N. Dutt, "Optimal Integration of Inter-Task and Intra-Task Dynamic Voltage Scaling Techniques for Hard Real-Time Applications", ICCAD, 2005.

[17] C. Lee, J. Hahn, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim, "Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling", IEEE Transactions on Computers. Vol. 47, No. 6, pp. 700-713 June 1998.

[18] K. Flautner, D. Flynn, D. Roberts, and D. Patel, "IEM926: An Energy Efficient SoC with Dynamic Voltage Scaling", Designers' Forum, DATE, 2004.

Appendix A: Details of code example in Figure 1 (a)

```
#define P(X,Y)          *(img+(y)*XMAX+(x))
__inline void block_filter1(unsigned char* img, int x, int y)  {
   P(x, y) = P(x-2, y)*A[0] +P(x-1, y)*A[1] +P(x+0, y)*A[2] +P(x+1, y)*A[3];
      // this pattern repeated for all pixels in 4x4 block until…
   P(x+3, y+3) = P(x+1, y+3) *A[0] +P(x+2, y+3)*A[1]
                  +P(x+3, y+3)*A[2] +P(x+4, y+3)*A[3]; }
```