Compiler Support for Dynamic Pipeline Scaling

Kuan-Wei Cheng, Tzong-Yen Lin, and Rong-Guey Chang
Department of Computer Science
National Chung Cheng University
Chia-Yi, Taiwan
{why93,lty93,rgchang}@cs.ccu.edu.tw

Abstract

Low power has played an increasingly important role for embedded systems. To save power, lowering voltage and frequency is very straightforward and effective; therefore dynamic voltage scaling (DVS) has become a prevalent low-power technique. However, DVS makes no effect on power saving when the voltage reaches a lower bound. Fortunately, a technique called dynamic pipeline scaling (DPS) can overcome this limitation by switching pipeline modes at low-voltage level. Approaches proposed in previous work on DPS were based on hardware support. From viewpoint of compiler, little has been addressed on this issue. This paper presents a DPS optimization technique at compiler time to reduce power dissipation. The useful information of an application is exploited to devise an analytical model to assess the cost of enabling DPS mechanism. As a consequence we can determine the switching timing between pipeline modes at compiler time without causing significant run-time overhead. The experimental result shows that our approach is effective in reducing energy consumption.

1. Introduction

Since most embedded systems are portable, reducing energy consumption to extend the lifetime of batteries has become a crucial issue. In recent years, many techniques have been proposed to address this issue. DVS is the famous one, which has been demonstrated by much work to be very effective [8, 2, 12]. It adjusts dynamically voltage and frequency to save power, as indicated in Equation 1.

$$E \propto f \times C \times V^2 \tag{1}$$

, where A $\propto$ B means A is in direct ratio to B. However, DVS has no effect on energy saving when the voltage reaches its low bound because it becomes a constant [10]. Fortunately, with reference to Equation 2, energy is in direct ratio not only to the clock frequency and the square of the voltage, but also to instruction-per-cycle (IPC).

$$E \propto f \times V^2 \times t \propto f \times V^2 \times \frac{I_t}{f \times IPC} \propto \frac{V^2}{IPC} \tag{2}$$

Thus, we can reduce energy dissipation at low-voltage level based on IPC. Equation 1 and Equation 2 reveal that IPC is the key to power dissipation at low-voltage level. This fact shows that power will increase in the opposite direction of IPC and motivates our

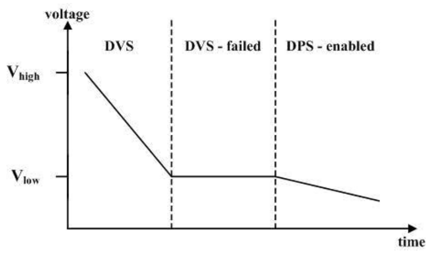


Figure 1. **Voltage characteristic**

low-power idea to devise a DPS technique to evaluate the IPC and determine the switching timing between pipeline modes at compiler time. In DPS, the pipeline consists of deep mode and shallow mode. The deep mode is the default pipeline mode and the shallow mode is designed by dynamically merging adjacent pipeline stages of deep mode, where the latches between pipeline stages are made transparent and the corresponding feedback paths are disabled. In theory IPC is in inverse ratio to the pipeline depth, the IPC of deep mode may be smaller than that of shallow mode [6]. Therefore, executing applications in shallow mode will lead to the reduction of power dissipation. But this statement is not always true. In reality, many factors in deep pipeline mode will influence IPC [15, 13].

Hence, if we want to apply DPS to save power at low-voltage level, we must decide when the pipeline enter deep mode or shallow mode depending upon the IPC. Consider the voltage characteristic shown in Figure 1. In the first stage, DVS is applied to save power at high-voltage level. Although reducing voltage is very effective to low power, DVS fails in the second stage when the voltage reaches its lower bound. At the final stage, we can enable DPS to switch the pipeline modes based on IPC. Since IPC is affected by some factors, we can consider their impact on IPC to determine the switching timing between pipeline modes to save energy.

Previous work on DPS was proposed by architects with architectural support [10, 7, 4, 14, 3]. However, the research about how to solve this issue with compilation techniques remains open. In this paper, we present an optimization technique to enable DPS with respect to IPC at compile time. We first partition an application into many regions and then calculate the IPC of each region to determine the switching timing between pipeline modes based on our evaluating model. Since our work is performed at compiler time, the run-time overhead will be small and the hardware cost and complexity will be as minimal as possible. The experimental results prove that the energy reduction really benefit from our work.

The rest of this paper is organized as follows. Section 2 we study the previous work on DPS. Section 3 gives the overview of our work and then presents our approach in detail. The experimental results are shown in Section 4. Finally, we conclude our paper in Section 5.

2. Related Work

Although many researchers still focus on devising new DVS techniques, other low-power techniques that are irrelevant to DVS are valuable to be exploited. DPS is a typical example. Previous work on DPS has concentrated on hardware techniques, but how to solve this issue using software techniques remains open. For example, Kop-panalil et al. devised a DPS technique by switching the pipeline between deep mode and shallow mode at run time with respect to high and low frequencies respectively [10]. Hiraki et al. proposed a method that skipped several pipeline stages and then used a decoded

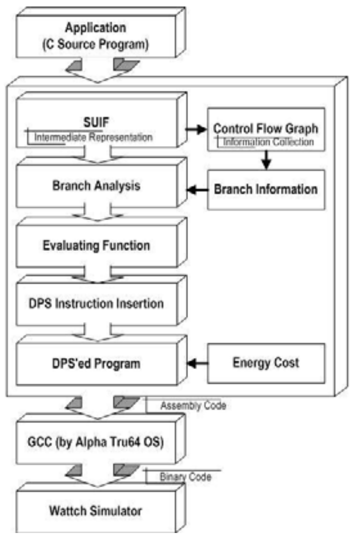


Figure 2. **The proposed DPS compilation system**

buffer to replace the functionality of the original pipelines for low-power support [7]. Ernst et al. speculated on the timing information by monitoring the error rate of pipeline processing to switch pipeline modes at circuit level [4]. Manne et al. applied pipeline gating to reduce energy consumption by stopping wrong-path instructions from entering the pipeline when a branch misses its prediction result [14]. Efthymiou et al. [3] applied a hardware-controlling method to reduce energy dissipation by adapting the depth of pipeline stages.

3. The Proposed Approach

In this section, we focus on how our DPS approach is applied to applications to save energy at compiler time. We first introduce our basic idea in Section 3.1. In section 3.2, we depict the method to partition a code into regions and then present the evaluating function to decide the switching between pipeline modes. The mechanism to enable DPS is given in Section 3.3.

3.1. Basic Idea

Figure 2 shows our compilation system composed of SUIF framework [16], our proposed engine, and Wattch simulator. First, an application is compiled by SUIF as a control flow graph (CFG) and a data flow graph (DFG). Then the CFG and DFG are analyzed to identify the loop regions and collect information for our evaluating model and identify loop regions. In our work, a code will have another type of regions, non-loop regions, except loop regions. Indeed, a region is an union of basic blocks as a unit that our DPS can manipulate. The evaluation model has two goals: one is to partition the remaining part of the loop regions into non-loop regions and the other is enable each region to enter a suitable pipeline mode. The details of partitioning scheme is described in Section 3.2. To activate DPS, for each region we will insert the DPS-enable function in its entrance at compile time so that it can be executed in proper pipeline mode to save energy based on its IPC. Since the switching between pipeline modes best is very hard to decide, we propose an evaluation model to decide the timing during execution. The evaluation model is presented in detail in Section 3.3. In this way, the code will be switched between different pipeline modes at run time. The experiment is performed on the Wattch simulator [1] with DSPstone and Mediabench benchmark suites.

3.2. Evaluation Model for Switching Pipeline Modes

As mentioned in Section 1, the IPC of deep pipeline mode is not always larger than that of shallow mode. As a consequence, the shallow mode may have better power saving than the deep mode according to Equation 2. Thus to reduce power reduction, each region can enter deep mode or shallow mode based on the IPC during execution. To achieve this objective, we conduct an evaluation model to decide the switching timing between deep mode and shallow mode at compiler time. Since the calculation of IPC closely relates to the size of a region, how to partition a code into regions becomes very important to our work. On one hand, if the region size is too large, we may lose the chances to take advantage of switching pipeline modes to save energy. On the other hand, although the small region size can allow us to apply the DPS optimization to a code, it possibly generates severe switching overheads. However what the size of a region is the optimal solution for our approach is very hard to decide, thus we attempt to seek for a principle to guide our selection in this section. With our observations, since the loops usually dominate execution time and power consumption of a code, they are the key to our decision. This result motivates us to use major loop as a guideline to classify a code into regions to perform our code partitioning.

To use the loops to partition a code, they must be identified first and then be referred to divided the remaining part into non-loop regions. Below we present our partitioning approach and evaluation model. Given a code $G = (V, E)$, it is divided into two types of regions, Γ_1 and Γ_2, where $\Gamma_1, \Gamma_2 \subseteq V \times E, G = \Gamma_1 \bigcup \Gamma_2$, and $\Gamma_1 \bigcap \Gamma_2 = \emptyset$. Note $\emptyset$ represents the empty set; that is Γ_1 and Γ_2 are disjoint. We first define Γ_1 in case A and then use Γ_1 to define Γ_2 in case B.

Case A: Γ_1 is the set of regions, which are composed of loops. In other words, each region in Γ_1 only contains a loop.

After defining the loop regions, to classify the non-loop regions, we must present our evaluation model first. Then the evaluation model is applied to loop regions for categorizing the non-loop regions. Below we first give some assumptions and then present how to use them to divided the non-loop part of a code into non-loop regions and finally formalize our evaluation model.

For a code $G = (V, E)$, where $V = \{R_1, R_2, ..., R_n\}$ is the set of regions in G and $E = \{(u, v) \mid u, v \in V \text{and } u \neq v\}$. That is, E is the set of edges between regions. For each region R_i, we assume:

- N_{R_i}: the number of instructions in region R_i, for $i = 1, 2, 3, \cdots$
- N_{b_i}: the number of branches in R_i
- B_{ij}: the jth branch in R_i
- $P_{B_{ij}}$: the probability that B_{ij} is taken
- C_i: the number of clock cycles that does not result from any branch in R_i
- CB_{ij}: the number of clock cycles that results from that B_{ij} is taken
- $\overline{CB}_{ij}$: the number of clock cycles that results from that B_{ij} is untaken

According to Equation 1 and Equation 2, IPC predominate the determination of switching pipeline modes during execution. With the information collected previously and the above terminologies, we can present our evaluating model as follows.

$$\Omega_{R_i} = \sum_{j=1}^{N_{b_i}} [P_{B_{ij}} \times CB_{ij} + (1 - P_{B_{ij}}) \times \overline{CB}_{ij}] + C_i \tag{3}$$

$$\Theta_{R_i} = N_{R_i} / \Omega_{R_i} \tag{4}$$

Equation 3 estimates the clock cycles required for each region of the target code, which is also applied to classify the non-loop regions. Equation 4 calculates the IPC for each region and is the guideline to enable DPS. Since the loop regions very likely dominates power dissipation of a code, we use the following parameter Λ with the aid of the evaluation function of regions in Γ_1 to partition the non-loop part of a code. Λ is defined as the maximum of all Ω_{R_i} in Γ_1. Formally, it can be described as follows.

$$\Lambda = \{\Omega_R \mid \exists R \in \Gamma_1 \text{ and } \Omega_R \geq \Omega_{R_i}, \text{for } i = 1, \cdots, n\} \tag{5}$$

Although the loops usually consume the majority of power dissipation for an application, using λ to partition the non-loop part can be furthermore improved. Instead, we adapt Λ as the new parameter by timing a α to it, where α is a real number. Thus Γ_2 can be defined on the basis of $\alpha\Lambda$ in the following case B.

Case B: For a code we first identify the loop regions in Case A and the remaining part is classified into non-loop regions. This part consists of two types of regions. The first is a set of code segments existing between loops and the second type has two special parts. One is from the beginning of a code to the beginning of the first loop region and the other is from the end of the last loop region to the end of a code. For each non-loop part, on one hand if its evaluation value (Ω value) is smaller than $\alpha\Lambda$, then it is identified as a non-loop region. On the other hand, if Ω value is larger than $\alpha\Lambda$, then it will be categorized into many non-loop regions so that their Ω values are not larger than $\alpha\Lambda$.

To make our partitioning mechanism clear, the above steps are summarized in Figure 3.

3.3. DPS Enabling

After the code partitioning has been done, to enable DPS, we insert a function DPS_enable () into its head of each region to make it executed in deep mode or shallow mode. The DPS_enable () is implemented as follows.

$$DPS\_enable() \begin{cases} Lda & \text{\#SYSCALL\_DEEP2SHAW} \\ Call\_Pal & \#131 \\ Lda & \text{\#SYSCALL\_SHAW2DEEP} \\ Call\_Pal & \#131 \end{cases}$$

DPS_enable() provides two functionalities to switch between pipeline modes with the system call of Alpha 21264 Call_Pal #131. #SYSCALL_DEEP2SHAW switches the pipeline from deep mode to shallow mode and #SYSCALL_SHAW2DEEP switches the pipeline from shallow mode to deep mode. In this way, we are able to determine the timing to switch pipeline modes. The Ω value of each region calculated by Equation 4 is used for DPS_enable() when the code is compiled by our system. Thus the code will dynamically enter the deep mode or shallow mode during execution after the DPS_enable() is inserted into it. Finally the optimized DPSed program is performed on the modified Wattch simulator.

```
For a given code $G = (V, E)$, divide G into two types of
pipeline regions, $\Gamma_1$ and $\Gamma_2$, where $\Gamma_1, \Gamma_2 \subseteq V$,
$V = \Gamma_1 \cup \Gamma_2$, and $\Gamma_1 \cap \Gamma_2 = \emptyset$;
1. Identify the loops of $G$ as the first type of regions, $\Gamma_1$.
   Assume $\Gamma_1 = \{R_{a_1}, R_{a_2}, \cdots, R_{a_n}\}$, where
   $R_{a_i} \cap R_{a_j} = \emptyset$ for $i \neq j$.
2. Define $\Lambda = \{\Omega_{R_{a_i}} \mid \exists\, R_{a_i} \in \Gamma_1$ and $\Omega_{R_{a_i}} \geq \Omega_{R_{a_j}}$ ,
   for $j = 1, ..., n\}$
3. For the following non-loop parts:
   (a) The code segment from the beginning of $G$
         to the beginning of the first loop region.
   (b) The code segment from the end of the last loop
         region to the end of $G$.
   (c) The code segments between loops.

   Partition them into a set of regions $\Gamma_2 =$
   $\{R_{b_1}, R_{b_2}, ..., R_{b_m}\}$, where $R_{b_i} \cap R_{b_j} = \emptyset$ for $i \neq j$
   and $\Omega_{R_{b_i}} \leq \alpha\Lambda$ for $i = 1, 2, \cdots, m$.
```

Figure 3. **Classification of regions**

4 Experimental Results

In Section 4.1, we introduce the system configuration of our work and present the experimental results in Section 4.2.

4.1. System Configuration

The underlying hardware is the Alpha 21264 processor, which contains one fetch buffer, four integer ALUs, two floating-point ALUs, one integer multiplier/divider, and one floating-point multiplier/divider, etc. In instruction window, RUU indicates register update unit and LSQ comprises load queue (LQ) and store queue (SQ). Its main features are summarized in Table 1. To perform our proposed approach, we extend the pipeline mode from one mode to two modes. We assume that the original pipelining mode is shallow mode and the new mode is deep mode by constructed by adding extra four stages to shallow pipeline. It is designed to dynamically disable one of each pair of stages by making the latches between pipeline stages transparent so that the processor can switch between these two pipeline modes. The software configuration is shown in Table 2. The SUIF compiler infrastructure is the front end of our system and generate CFG and branch information. The operating system is Tru64 UNIX for 64-bit instruction set architecture. The Wattch simulator is an architectural simulator that provides cycle-by-cycle simulation and detailed out-of-order issue with multi-level memory system [9]. For keeping consistence with our DPS approach, it has been modified to support shallow pipeline mode and deep pipeline mode.

4.2. Experimental Results

In our experiment, the deep mode is the default pipeline mode and the shallow mode is chosen during execution if necessary. The energy reduction benefits by the switching between deep mode and shallow mode depending on the IPC of a region, which is calculated by equation 4. The experiment is performed on the Wattch simulator with

Processor Core	
Pipeline length	4 cycles (shallow mode) 8 cycles (deep mode)
Fetch buffer	8 entries
Functional units	4 Int ALU, 2 FP ALU, 1 Int mult/div, 1 FP mult/div, 2 mem ports
Instruction window	RUU=80, LSQ=40
Issue width	6 instructions per cycle: 4 Int, 2 FP
Memory Hierarchy	
L1 D-cache size	64KB, 2-way, 32B blocks,
L1 I-cache size	64KB, 2-way, 32B blocks,
L1 latency	1 cycle
L2	Unified, 2MB, 4-way LRU 32B blocks, 11-cycle latency
Memory latency	100 cycles
TLB size	128-entry, fully-associative, 30-cycle miss

Table 1. Hardware configuration

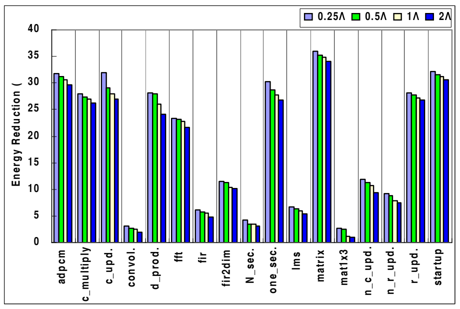
OS and Software Configuration	
Profiler	SUIF
Compiler	MachSUIF
OS	Tru64 UNIX
Simulator	Wattch v.1.02 with DPS

Table 2. Software Configuration

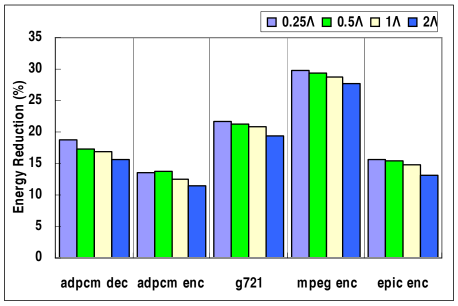
DSPstone and Mediabench. For each program, the baseline is its original energy dissipation and the optimized energy is measured by performing our DPS approach. This benchmark is compiled by the Alpha compiler with default settings and linked with the intrinsic library on Tru64 UNIX operating system.

Figure 4a shows the energy reduction by comparing the baseline energy and the optimized energy for DSPstone. In this experiment, we let $\alpha = 0.25, 0.5, 1.0, \text{and}, 2.0$ to measure the effects of various partitioning sizes of non-loop regions. The energy saving ranges from 2% to 35%, with a mean of reduction 17.8%. As the partitioning size of non-loop region λ becomes larger, the energy saving decreases slowly. With our observation, the large-size region eliminates some chances to switch the pipeline modes based on IPC and thus slightly increases energy consumption. Nine of these programs including adpcm, complex_multiply, complex_update, dot_product, fft, iir_biquad_one_sections, matrix, real_update, and startup, have better energy saving about from 22% to 35%. The reason is that there are many branches in them and thus our DPS approach can take advantage of them to save energy depending upon the contribution of branch penalty to IPC. With our experiences, our approach works better for the codes with many loops and larger loops. Note that many programs have large outer loops such as event-driven programs or programs with GUI, which may include almost the entire programs. In this experiment, the typical example for above discussion is matrix testbench, and it has the best energy saving about 35%.

Figure 4b shows energy reduction by performing our profile-based DPS with Mediabench benchmarks. Since these programs are loop-intensive, they at least contain a



(a)



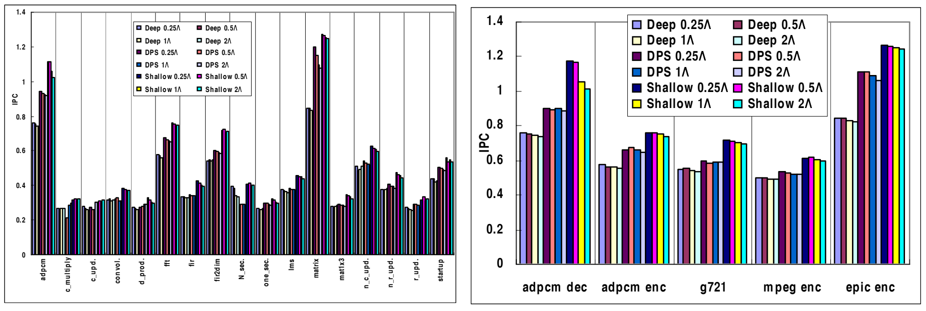
(b)

Figure 4. Energy reduction of various λ of profile-based DPS for DSPstone and Mediabench

nested loop. For each benchmark, the Ω value of its loop region is large and thus the small $\alpha\lambda$ can have better energy reduction than larger ones. From Figure 4b, the bar of 0.25λ has the best energy saving and 2λ is the worst. Compared to the baseline version, our DPS approach can save energy by 12% to 28% and an average of 18.2%. Notice that if a program has a very large loop, it may contain only one non-loop region or two very small non-loop regions. In this case, the value of $\alpha\lambda$ will not influence our experimental result.

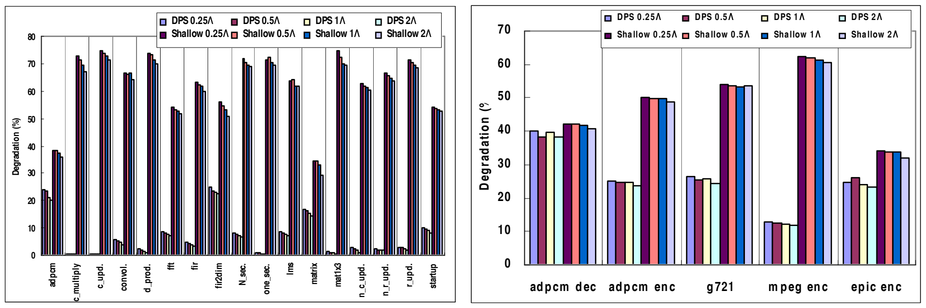
Figure 5 demonstrates the effect on IPC for three cases using DSPstone and Mediabench as the metric. Deep and Shallow represent the results of executing a program in deep and shallow pipeline modes respectively; DPS indicates the result of applying our DPS approach to a program. They are still measured for various partitioning sizes of non-loop regions $0.25\lambda, 0.5\lambda, \lambda, \text{and}, 2\lambda$. For DSPstone, the average IPCs of Deep case and Shallow case are 0.4 and 0.52. In DPS case, the average IPC is 0.45, which is between those of deep mode and shallow mode. For Mediabench, the average IPCs of Deep case, Shallow case, and DPS are 0.63, 0.87, and 0.75. Normally, the IPC of deep mode is larger than that of shallow mode, but this contradicts with the results shown in Figure 5. The reason comes from a key observation that branch penalty is closely related to IPC. Since shallow mode has smaller branch penalty than deep mode, as a result its IPC become larger than that of deep mode. In DSPstone, the IPCs of adpcm, fft, fir2dim, and matrix are larger. This is because since they are loop-intensive applications and the loops in them contribute a lot to the increase of IPC. In Mediabench, the IPCs of adpcm decoder, adpcm encoder, g721, mpeg encoder and epic encoder are larger. This is because the five chosen programs are loop-intensive applications and the loops in them contribute a lot to the increase of IPC.

Figure 6 show the effect of our DPS approach on performance for DSPstone and Mediabench. The latency between pipelining stages is designed to be equivalent to increase performance and achieve resource sharing at each clock. In theory, the performance is in direct ratio to the number of pipelining stages and thus the longer pipeline will lead to the performance. Thus, the processor will result in slowdown when executing in shallow mode. The performance will be degraded if the pipelining stages are merged into shallow mode. In reality, the performance may be degraded due to many factors such as pipelining hazards, branch penalty, or switching overhead between pipeline modes. For each benchmark, the performance of deep mode with λ is the baseline to compare those of deep mode and our DPS method for various λs. For the performance of DSPstone in Figure 6a, on average, our DPS approach leads to 6.23% degradation in performance. By contrast, the performance degradation of shallow pipeline mode is 61.62%, which



(a) (b)

Figure 5. **Chang in IPC for three cases using DSPstone and Mediabench as benchmark**



(a) (b)

Figure 6. **Relative performance of various Λs for DSPstone & Mediabench**

is almost ten times larger than the above one. Although the DPS switches the pipelining modes based on the IPC to save energy, the switching slows down the processor compared to the high-speed execution in deep mode. In addition, larger λ has a better performance than smaller ones since it causes the pipeline to enter the shallow mode more infrequently. For Mediabench, on average, the shallow mode and profile-based DPS have 48.02% and 25.23% performance degradations.

5. Conclusions

DVS has been proven be very effective in low power optimizations, but it cannot further save energy when the voltage reaches its lower bound. Fortunately, DPS can overcome this limitation by adjusting pipeline modes based on IPC. Previous work resolved this issue with hardware techniques and thus increased hardware cost and design complexity. In this paper, we present a DPS technique to reduce power dissipation by proposing an evaluating model so that they can decide the timing of entering the proper pipeline mode. In contrast, our work can eliminate hardware overhead and reduce energy consumption according to the code behavior at compiler time. To investigate the effect of our approach, we perform the experiment with various criteria for DSPstone and Medieabench. In summary, the results show that smaller partitioning sizes of non-loop regions can create optimization space and loop-intensive applications provide more chances to optimize code to save energy.

References

[1] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural level power analysis and optimizations. In *International Symposium on Computer Architecture*, 2000.

[2] T. Burd and R. Brodersen. Design issues for dynamic voltage scaling. In *International Symposium on Low Power Electronics and Design*, 2000.

[3] A. Efthymiou and J. D. Garside. Adaptive pipeline depth control for processor power-management. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2002.

[4] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *the 36th International Symposium on Microarchitecture*, 2003.

[5] M. Gerndt. Automatic parallelization for distributed-memory multiprocessing systems. In *Phd Thesis*, 1989.

[6] A. Hartstein and T. R. Puzak. The optimum pipeline depth for a microprocessor. In *ACM/IEEE International Symposium on Computer Architecture*, 2002.

[7] M. Hiraki, R. S. Bajwa, H. Kojima, D. J. Corny, K. Nitta, A. Shridhar, K. Sasaki, and K. Seki. Stage-skip pipeline: A low power processor architecture using a decoded instruction buffer. In *International Symposium on Low Power Electronics and Design*, 1996.

[8] C. Hsu and U. Kremer. The design, implementation, and evaluation of a com-piler algorithm for cpu power reduction. In *the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 2003.

[9] R. Kessler, E. McLellan, and D. Webb. The alpha 21264 microprocessor architecture. In *Intl Conf. Computer Design*, 1998.

[10] J. Koppanalil, P. Ramrakhyani, S. Desai, A. Vaidyanathan, and E. Rotenberg. A case for dynamic pipeline scaling. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2002.

[11] J. Kornerup. Mapping powerlists onto hypercubes. In *Masters Thesis*, 1994.

[12] C. Krishna and Y.-H. Lee. Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems. In *the 6th Real Time Technology and Applications Symposium*, 2000.

[13] D. Lilia. Reducing the branch penalty in pipelined processors. *IEEE Computer*, 21(7):47–55, 1988.

[14] S. Manne, D. Grunwald, and A. Klauser. Pipeline gating: Speculation control for power reduction. In *ACM/IEEE International Symposium on Computer Architecture*, 1998.

[15] D. Parikh, K. Skadron, Y. Zhang, and M. Stan. Power-aware branch prediction: Characterization and design. In *the IEEE Transactions on Computers*, 2004.

[16] G. SUIF. *Stanford University Intermediate Format*. http://suif.stanford.edu.