

# Power-Aware Compilation with Architectural Support and Instruction Scheduling

Yu-Ting Hung

Department of Computer Science  
and Information Engineering  
National Chung Cheng University

Chia-Yi, Taiwan

hyt91@cs.ccu.edu.tw

Tcong-Yen Lin

Department of Computer Science  
and Information Engineering  
National Chung Cheng University

Chia-Yi, Taiwan

lty93@cs.ccu.edu.tw

Rong-Guey Chang

Department of Computer Science  
and Information Engineering  
National Chung Cheng University

Chia-Yi, Taiwan

rgchang@cs.ccu.edu.tw

## ABSTRACT

This paper is to develop an innovative technique using dynamic voltage scaling (DVS) to reduce the energy dissipation. The objective of our work is to maximize the period of idle functional units and minimize the transitions between power modes. To achieve this goal, our work is performed in the following steps. First, the program will be built as a control flow graph that comprises many regions. Second, we build the power model of the program and collect the power information by profiling the control flow graph. Then the regions with the same functional units will be merged if the merging still preserves the program dependencies. Next, the idle functional units will be turned off and each region will be assigned a power mode based on the power model. Finally, the program is rescheduled to merge the regions to reduce the transitions between power modes. Our work is implemented on the basis of SUIF2 compiler infrastructure and the Wattch simulator and measured by using SPEC2000, DSPstone, and JPEG decoder as benchmarks. On average, the experimental results show that our work can save the energy by 26% with performance degradation between 5% and 18%.

## Categories and Subject Descriptors

E1 [Low-power design]: compilation, scheduling, and partitioning

## General Terms

Experimentation

## Keywords

DVS, scheduling, power mode, energy dissipation.

## 1. INTRODUCTION

With the rapid improvement of semiconductor technology, the capacity of chip has grown very fast and the technology of System-on-Chip has become very critical to integrate hardware, software, and applications. This trend has led to the popularity of embedded systems on multimedia and wireless applications. However, since most of these embedded systems are portable, how to reduce the energy dissipation to extend their lifecycle without resulting in the great performance loss has become one of the important research issues.

The power dissipation is closely related to the voltage and clock frequency [18]. The architectures of modern processors provide various power saving modes to lower energy dissipation by adjusting the voltage and the clock frequency at run time. For

example, there are three power modes (run, idle, and sleep modes) in the StrongARM SA-1100 processor. The switching overhead of returning to the run mode from the sleep mode must take 160ms and consume some energy to wake up on-chip activities [12]. Previous work [1,23] showed the switching overheads of power modes have a great impact on energy dissipation. In this paper, the objective is to reduce the energy dissipation by maximizing the period of idle functional units and minimizing the transitions between power modes. To exploit the maximum potential to achieve this goal, we first perform the interprocedural analysis using the SUIF2 compiler infrastructure and divide the program into a lot of regions, which will contain at least one functional unit, and then we profile the program to build the power model. Next, the regions containing the same functional units will be merged, in which the merging will preserve the program dependencies. Then the idle functional units in each region will be turned off and each region will be assigned one power mode by referring to the power model. Afterward, we reschedule the program so that the regions with the same power modes will be merged if the merging does not violate the program dependencies. Finally, we assign the power modes to basic blocks to further optimize energy and performance. Note the potential of the above mergings is limited by the degree of instruction level parallelism. The experiments will show how much this potential can be exploited. Our implementation is performed on the compilation system on the basis of the SUIF2 [22] compiler infrastructure and the Wattch [5] simulator with SPEC2000, DSPstone, and JPEG decoder benchmarks.

The remainder of this paper is organized as follows. Section 2 presents our main idea with a motivating example. Section 3 describes the system architecture and the algorithm in detail. The experimental results are shown in Section 4. Section 5 is the related work. Finally, section 6 concludes our work briefly.

## 2. Motivating Example

Figure 1 shows a motivating example to explain the basic idea of our work. The left-hand side is the source code and the right-hand side is the optimized code after applying our algorithm. The original code segment consists of two loops that perform additions. For each loop, there is one multiplication following it that will be performed. Obviously, no multipliers are needed in the loops so that we can turn off the multiplier off to save the energy. Now we demonstrate our work in detail. To begin with, we assume that there are four regions in the original code, two loops and two expressions containing the multiplication. These two loops and two expressions can be merged into two regions,

respectively since they contain the same functional unit. Next, we turn off the multiplier in the region containing two loops and turn off the adder in the region containing two expressions. Finally, we assign the region containing two loops the power down mode and the region containing two expressions the normal power mode. In contrast with the original code, the optimized code will have the smaller switching costs.

<pre>int main (void) { ... for ( i=0; i&lt;100; i++) a = a + 1; tmp1 = a * c; for ( j=0; j&lt;200; j++) b = b + 2; tmp2 = b * d; ...}</pre>	<pre>int main (void) { ... /* Insert code to turn off the multiplier for ( i=0; i&lt;100; i++) a = a + 1; for ( j=0; j&lt;200; j++) b = b + 2; /*Insert code to turn on the multiplier tmp1 = a * c; tmp2 = b * d; ...}</pre>
---	---

Figure 1. Motivating example

### 3. The Algorithm

In this section, we first describe the system architecture of our work and then present our DVS algorithm.

#### 3.1 System Architecture

Figure 2 shows our system architecture. Our approach is implemented on the basis of the SUIF2 compiler infrastructure and the Wattch simulator. To begin with, the programs are inputs to the front end of SUIF2 and then are transformed into the SUIF intermediate representations through the optimization, the high SUIF, and the low SUIF steps. SUIF2 can perform the interprocedural analysis on the source programs that helps us do global alias analysis, specialization, and the analysis of data flow. Next, the intermediate representations produced from SUIF2 have become inputs of Machine SUIF (MachSUIF). MachSUIF helps us build the control flow graph (CFG) and the data dependence graph (DDG) for input intermediate representations by providing the analysis libraries and optimization interfaces. Then we perform the low-power optimizations on the program. First, we collect the power information and build a power model by analyzing the control flow graph and the data dependency graph of the input program. Meanwhile, the control flow graphs will be divided into many regions according to the usage of functional units such as adders and multipliers for integer and floating computations. Then each region will be assigned a power saving mode by referring to the power model. Next, the idle functional units in a region will be turned off. Then two regions will be merged to reduce the transitions of power modes, if the functional units in them have the same statuses, their power modes are the same, and the merging still preserves the dependencies of the program. The process will repeat until no further mergings can continue. Finally, the resulting DVS'ed program is compiled and linked with run-time library to produce an executable code running on the Wattch simulator.

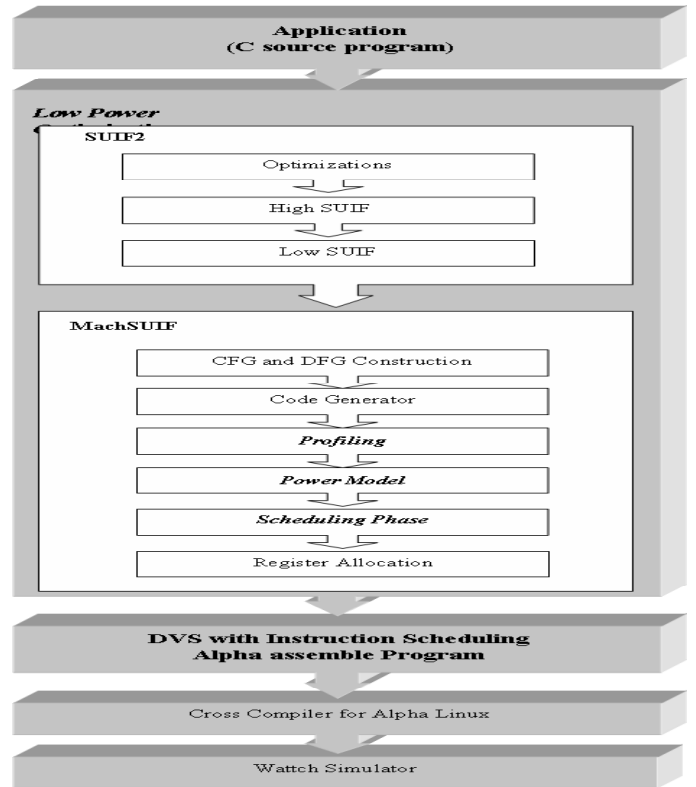


Figure 2. System architecture

#### 3.2 DVS Algorithm

In this section, we first describe the main idea of our work and then present the algorithm. To turn on and off the functional units, we add new instructions into the Alpha instruction set, which are listed below. In this paper, we only consider the adders and the multipliers for integer and floating point.

- alu.on switch ON one integer alu functional unit
- alu.off switch OFF one integer alu functional unit
- mdu.on switch ON one integer multiplier/divide functional unit
- mdu.off switch OFF one integer multiplier/divide functional unit
- alu.s.on switch ON one float alu functional unit
- alu.s.off switch OFF one float alu functional unit
- mdu.s.on switch ON one float multiplier/divide functional unit
- mdu.s.off switch OFF one float multiplier/divide functional unit

In our algorithm, we first perform the interprocedural analysis implemented in SUIF2 on the applications. The interprocedural analysis enables the compiler to analyze the whole program for further optimizations, and results in significant improvements on the performance of applications. Then the applications are translated into the SUIF intermediate formats. In the MachSHIF, these intermediate formats are constructed as CFGs and DFGs, and they will be divided into a lot of regions depending on the usage of functional units. Then we profile the control flow graph of the program and record the following power information of each basic block and each region. For each basic block  $B$ ,  $N(B)$  indicates the number of times  $B$  is executed,  $FU(B)$  indicates the set of functional unit used in the basic block  $B$ , and  $f_{mem}$  indicates the percentage of memory accesses per instruction in  $B$ . For a

region  $R$  in the basic block  $B$ ,  $T_{per}(R)$  is the percentage of its execution time in  $B$ . After the program is partitioned into regions, two regions, say  $R_i$  and  $R_j$ , will be merged if the following conditions are satisfied. (1)  $FU(R_i) = FU(R_j)$ . It implies these two regions contains the same functional units; (2)  $D(R_i, R_j) = \emptyset$ . It means that the merging of these two regions still preserve the dependencies of the program. The process will be repeated until no further merging can be performed. Next, we decide to turn the functional units in each region on or off depending on the threshold  $\gamma$ , which is the default number of clock cycles. In our work, we implemented two power modes in the Wattach simulator. One is normal mode with voltage 1.5V and clock frequency 600MHz, and the other represented by  $f_{down}$  is the power down mode with voltage 0.3V and clock frequency 300MHz. Note that in this paper, the default power mode of a region is the normal mode. In fact, we can implement more than two power modes in simulator and assign these power modes for regions. Moreover, we determine the assignment of power modes in the following two steps. In the first step, we first assign each region a power mode depending on the threshold  $\alpha$  and then perform the merging repeatedly until no further merging can be done. Formally, the regions  $R_i$  and  $R_j$  satisfying  $f(R_i) = f(R_j)$  and  $D(R_i, R_j) = \emptyset$  will be merged across basic blocks. In the second step, the basic block will be assigned a power mode depending on the threshold  $\beta$ .

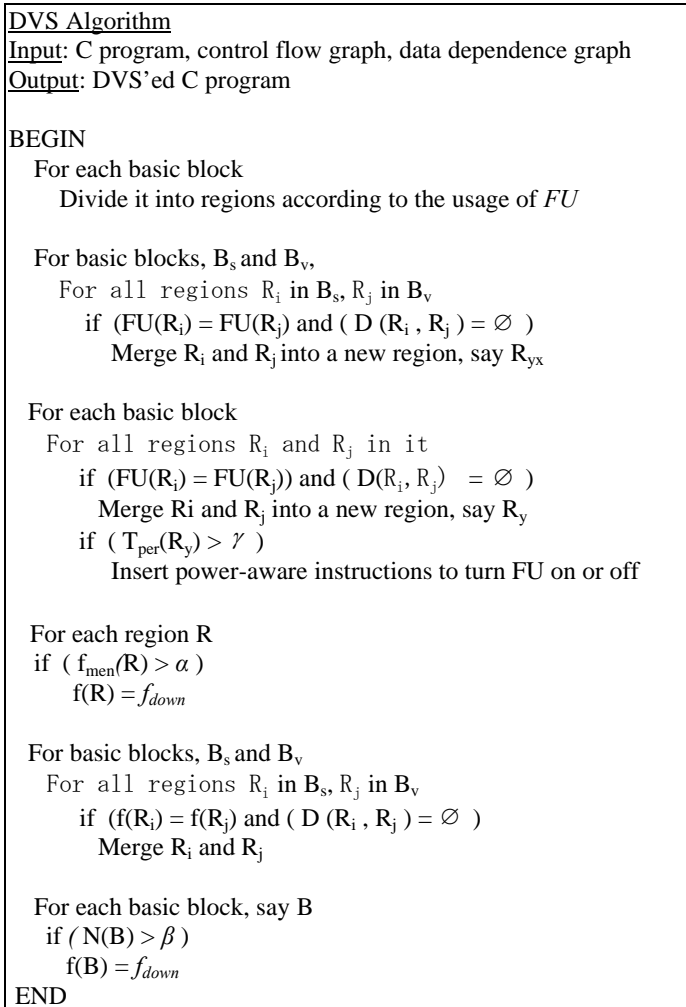


Figure 3. DVS algorithm.

Now we use 8x8 IDCT as an example to demonstrate our algorithm. Figure 4 shows the control flow graph and the power model for one code segment in the 8x8 IDCT.

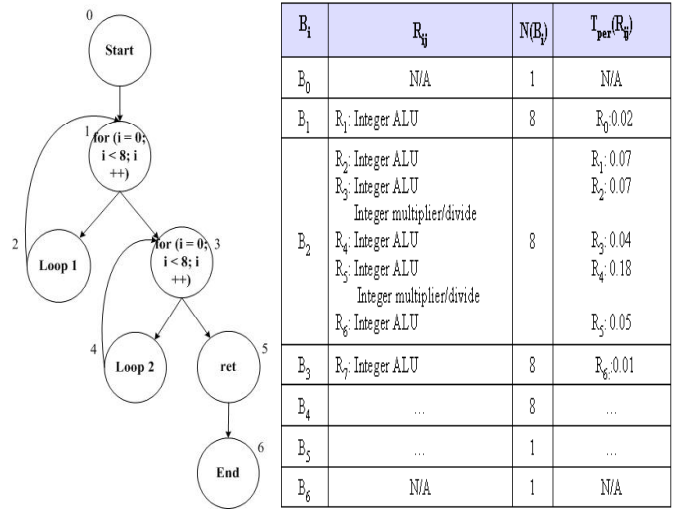


Figure 4 Control flow graph and power model for 8x8 IDCT

Figure 5 shows the dependence flow graph between  $R_2$  and new region  $R_{3-5}$  in the 8x8 IDCT to explain why they are not merged with  $R_1$ .

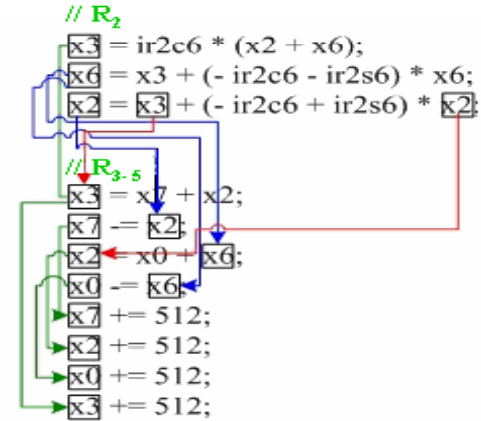


Figure 5. Partial dependence graph for 8x8 IDCT

Figure 6 shows the final result after applying our algorithm to one code segment in 8x8 IDCT. Initially, the adder is turned on since some instructions such as load and store will use it to calculate the target address and the multiplier is turned off. The left-hand side of Figure 5 is the original code fragment. It is divided into five regions by the usage of adder and multiplier. In this example, the profiling information indicates that regions  $R_1$ ,  $R_3$  and  $R_5$  use adder only, and regions  $R_2$  and  $R_4$  use both adder and multiplier. Thus, we can apply our algorithm to reschedule the code so that the regions with the same statuses can be merged. The right-hand side of Figure 5 shows the final result.  $R_1$  cannot be merged with  $R_3$  and  $R_5$  due to the dependences, although they have the same mode. However,  $R_3$  and  $R_5$  can be merged into a new one, because multiplier can be shut down in them and no dependences exist between them. Similarly,  $R_2$  and  $R_4$  can be merged into another new region.

```

Step1: //R1
x8 = x7 + x1;
x1 = x7;
x7 = x0 + x4;
x0 = x4;
x4 = x1 + x5;
x1 = x5;
x5 = x3 + x8;
x8 = x3;

Step2: //R2
x3 = ir2c6 * (x2 + x6);
x6 = x3 + (-ir2c6 - ir2s6) * x6;
x2 = x3 + (-ir2c6 + ir2s6) * x2;

Step3: //R3
x3 = x7 + x2;
x7 = x2;
x2 = x0 + x6;
x0 = x6;

Step4: //R4
x6 = ic3 * (x4 + x5);
x5 = (x6 + (-ic3 - is3) * x5) >> 6;
x4 = (x6 + (-ic3 + is3) * x4) >> 6;
x6 = ic1 * (x1 + x8);
x1 = (x6 + (-ic1 - is1) * x1) >> 6;
x8 = (x6 + (-ic1 + is1) * x8) >> 6;

Step5: //R5
x7 += 512;
x2 += 512;
x0 += 512;
x3 += 512;

Step1: //R1
// turn off multiplier
x8 = x7 + x1;
x1 = x7;
x7 = x0 + x4;
x0 = x4;
x4 = x1 + x5;
x1 = x5;
x5 = x3 + x8;
x8 = x3;
//turn on multiplier

Step2: //R2
x3 = ir2c6 * (x2 + x6);
x6 = x3 + (-ir2c6 - ir2s6) * x6;
x2 = x3 + (-ir2c6 + ir2s6) * x2;

Step3: //R3-5
// turn off multiplier
x3 = x7 + x2;
x7 = x2;
x2 = x0 + x6;
x0 = x6;
x7 += 512;
x2 += 512;
x0 += 512;
x3 += 512;
//turn on multiplier

Step4: //R4
x6 = ic3 * (x4 + x5);
x5 = (x6 + (-ic3 - is3) * x5) >> 6;
x4 = (x6 + (-ic3 + is3) * x4) >> 6;
x6 = ic1 * (x1 + x8);
x1 = (x6 + (-ic1 - is1) * x1) >> 6;
x8 = (x6 + (-ic1 + is1) * x8) >> 6;

```

Figure 6. The optimized code segment in 8x8 IDCT

## 4. Experimental Results

In this section, we present the experimental results of our work. Our work is implemented on the basis of the SUIF2 compiler infrastructure and the Watch simulator. The applications are compiled into the Alpha 21264 instructions by a cross compiler. The Alpha 21264 has four ALUs: one is the multiplier, two are used for addition and address calculation, and the final one is reserved. The experiments are measured using SPEC2000, DSPstone, and JPEG decoder as benchmarks by setting three thresholds  $\alpha = 0.3$ ,  $\beta = 10$ , and  $\gamma = 3$ . With our experiences, these settings are suitable for our work. The impact of the settings on our work will be discussed further in the near future. In this section, we use “with scheduling” to represent all optimizations of our work and “without scheduling” to represent the optimization of turning off the idle functional units.

### 4.1 Energy Evaluation

To demonstrate the effect of our work, we first show the utilization of functional units after applying our approach to the benchmarks in Figure 7 to Figure 10. The utilization of a functional unit is defined as the percentage of the total execution time when it is on and busy executing instructions [3]. In other word, that the utilization of a functional unit is high means this unit is less idle during execution. Figure 7 and Figure 8 show the results performed by with and without scheduling for DSPstone. Figure 9 and Figure 10 show the results after applying our approach with and without scheduling for SPEC2000. In these figures, we are neglectful of the utilization of the integer adder since this unit is always on and busy for some instructions like load and store instructions. The utilization of floating functional units is 100%. For integer multiplier, the utilization with scheduling is higher than that of without scheduling.

benchmark	Utilization (%)			
	Integer		Float	
	Adder	Mult	Adder	Mult
rawcaudio	--	85.8936	99.9	99.9
rawdaudio	--	99.9241	99.9	99.9
Complex multiply	--	99.2441	99.9	99.9
Complex update	--	98.5236	99.9	99.9
Convolution	--	93.2159	99.9	99.9
Dot product	--	99.7512	99.9	99.9
fir	--	92.8177	99.9	99.9
Fir2dim	--	77.5260	99.9	99.9
Irr_biquad_N_sections	--	84.4521	99.9	99.9
Irr_biquad_one_section	--	98.8154	99.9	99.9
lms	--	86.7987	99.9	99.9
matrix	--	34.6866	99.9	99.9
Matrix1 x3	--	94.9786	99.9	99.9
N complex update	--	85.3282	99.9	99.9
N real update	--	90.8177	99.9	99.9

Figure 7: Utilization without scheduling for DSPstone

benchmark	Utilization (%)			
	Integer		Float	
	Adder	Mult	Adder	Mult
rawcaudio	--	92.6696	99.9	99.9
rawdaudio	--	99.9312	99.9	99.9
Complex multiply	--	99.2424	99.9	99.9
Complex update	--	99.7814	99.9	99.9
Convolution	--	95.6945	99.9	99.9
Dot product	--	99.1498	99.9	99.9
fir	--	93.1474	99.9	99.9
Fir2dim	--	83.0473	99.9	99.9
Irr_biquad_N_sections	--	89.4521	99.9	99.9
Irr_biquad_one_section	--	89.9317	99.9	99.9
lms	--	87.4581	99.9	99.9
matrix	--	52.5648	99.9	99.9
Matrix1 x3	--	95.7082	99.9	99.9
N complex update	--	99.7544	99.9	99.9
N real update	--	92.1991	99.9	99.9

Figure 8: Utilization with scheduling for DSPstone

benchmark	Utilization (%)			
	Integer		Float	
	Adder	Mult	Adder	Mult
rawcaudio	--	85.8936	99.9	99.9
rawdaudio	--	99.9241	99.9	99.9
gzip	--	50.3412	99.9	99.9
vpr	--	84.7419	99.9	99.9
bzip	--	34.2197	99.9	99.9
gcc	--	80.4174	99.9	99.9
parser	--	86.2341	99.9	99.9
twolf	--	90.1435	99.9	99.9

Figure 9: Utilization without scheduling for SPEC2000

benchmark	Utilization (%)			
	Integer		Float	
	Adder	Mult	Adder	Mult
rawcaudio	--	92.6696	99.9	99.9
rawdaudio	--	99.9312	99.9	99.9
gzip	--	71.1120	99.9	99.9
vpr	--	85.7419	99.9	99.9
bzip	--	57.3017	99.9	99.9
gcc	--	86.2270	99.9	99.9
parser	--	88.6143	99.9	99.9
twolf	--	92.0714	99.9	99.9

Figure 10: Utilization with scheduling for SPEC2000

Figure 11 shows the energy reductions after applying our approach with and without scheduling for the DSPstone benchmark. As Figure 11 shows, our work can achieve the

average energy reduction of 24% with scheduling and the average energy reduction of 19% without scheduling. For DSPstone, the effect of scheduling is small since the code sizes of applications in DSPstone are smaller. For the matrix, the energy reduction caused by using scheduling can achieve up to 23% since it has a higher instruction level parallelism. Figure 12 shows the experimental results after applying our approach with and without scheduling for the SPEC2000 benchmark. On average, the total energy reduction can achieve up to 27%. The average energy reduction caused by the scheduling is around 11% that is larger than that of DSPstone. The reason is that the code sizes of applications in SPEC2000 are larger in comparison with those of DSPstone and consequently we can exploit more instruction level parallelism to optimize. In fact, with our experiences, our work can acquire the better energy saving if the number of multipliers in the CPU is large. For JPEG decoder, the energy saving can achieve up to 32.8% and 15.7% for using with and without scheduling, respectively.

matrix since the code sizes of applications in it are smaller. By contrast, due to the larger code sizes of applications, the performance degradation of SPEC2000 is around 12.6% and 14.1% on average for the cases with and without scheduling. For JPEG decoder, the performance degradations are 18.1% and 24.9% for using with and without scheduling.

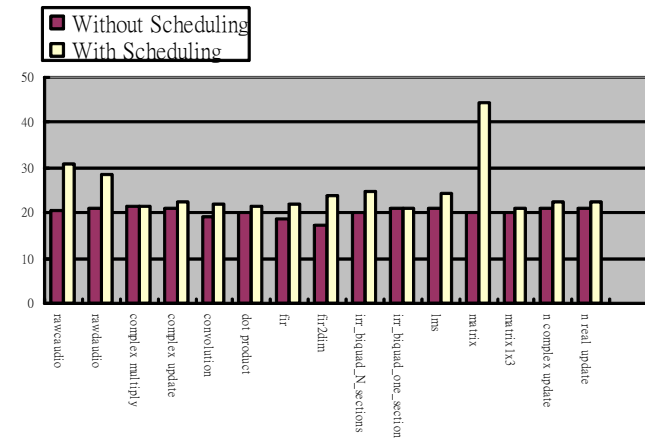


Figure 11: Energy reductions for DSPstone

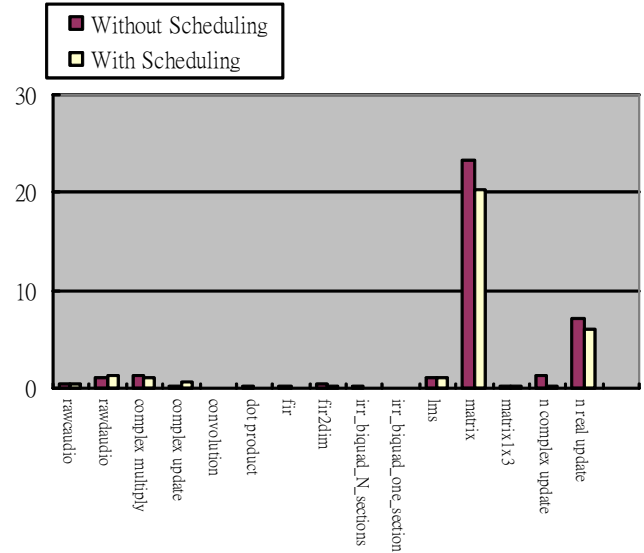


Figure 13: Performance degradation for DSPstone

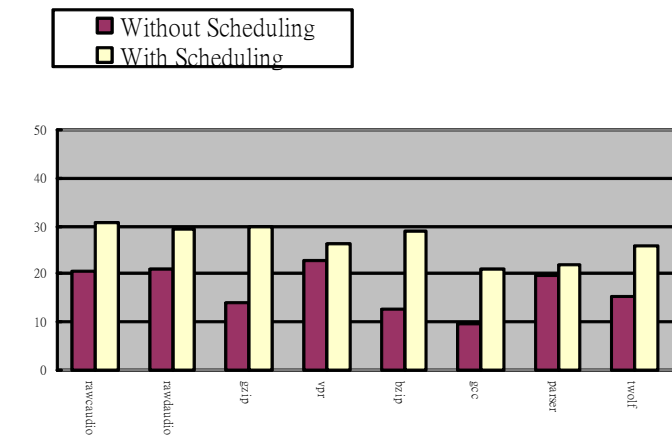


Figure 12: Energy reductions for SPEC2000

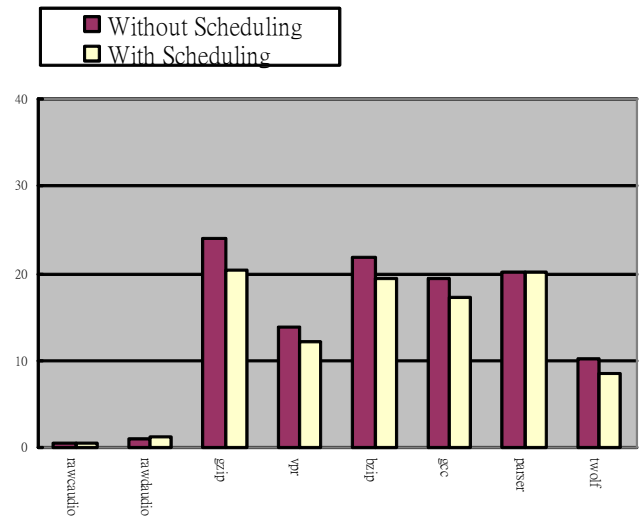


Figure 14: Performance degradation for SPEC2000

## 4.2 Performance Evaluation

Figure 13 and Figure 14 show the performance degradation after applying our work with and without scheduling for DSPstone and SPEC2000 benchmarks. In DSPstone, the performance degradation caused by slowing down the CPU is smaller except

## 5. Related Work

Previous work reduced the energy dissipation by proposing various DVS techniques [1,6,9,10,14,17,20,21,23]. Some work [14,17] focused on scheduling the tasks using DVS to meet the real-time constraint to lower energy consumption. Shin et al. also aimed at the intra-task scheduling under the real-time constraint based on the power information generated by compiler [20]. Previous work addressed the DVS issue by slowing down the frequency with a low voltage in the regions containing many memory accesses [9, 10]. In their work, the issue was modeled as the minimization problem with the performance and the transition constraints. Although they took the transition overheads into account, they did not reschedule the program to exploit the

potential of reducing the transitions between power modes. Rele et al. devised a region-based approach to reduce the energy dissipation by turn off the idle functional units for superscalar processors [18]. Their work only showed the impacts on the utilization of functional units and performance after applying their work to programs, but it did not demonstrate the experimental results about power dissipation. By contrast, on the one hand our work extends the period of idle functional units and on the other hand we performs instruction scheduling on the programs to reduce the number of transitions between power modes.

## 6. Conclusion and Future Work

This paper presents an effective DVS approach at compiler time to reduce power dissipation by attempting to minimize the transitions between power modes. Our work is implemented on the basis of SUIF2 and Wattch tools and performed with DSPstone, SPEC2000, and JPEG decoder benchmarks. On average, the experimental results show that our work can save the energy by around 26% with performance degradation between 5% and 13%. Our future will focus on the settings of three thresholds to see how they influence the optimizations of our work.

## REFERENCES

- [1] N. AbouGhazaleh, D. Moss' e, B. Childers, and R. Melhem. Toward the placement of power management points in real time applications. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, September 2001.
- [2] *Automation of Electronic Systems*, 5:115{192, April 2000.
- [3] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Pro\_le-based dynamic voltage scheduling using program checkpoints in the COPPER framework. In *Proceedings of Design, Automation and Test in Europe Conference*, March 2002.
- [4] L. Benini and G. Micheli. System-level power optimization: Techniques and tools. *ACM Transactions on Design*
- [5] D. Brooks , V. Tiwari , and M. Martonosi. Wattch: A Framework for Architectural Level Power Analysis and Optimizations. In *International Symposium on Computer Architecture (ISCA)* , Vanconver , British Columbia , 2000.
- [6] T. Burd and R. Brodersen. Design issues for dynamic voltage scaling. In *Proceedings of 2000 International Symposium on Low Power Electronics and Design (ISLPED'00)*, July 2000.
- [7] D. Burger and T. Austin. The SimpleScalar tool set v of Wisconsin, June 1997.ersion 2.0. Technical Report 1342, Computer Science Department, University
- [8] C.-H. Hsu and U. Kremer. Compiler-directed dynamic voltage scaling based on program regions. Technical Report DCS-TR-461, Department of Computer Science,Rutgers University, November 2001.
- [9] C.H. Hsu and U. Kremer. Single region vs. multiple regions: A comparison of different compiler-directed dynamic voltage scheduling approaches. In *Workshop on Power-Aware Computer Systems*, 2002.
- [10] C.H. Hsu and U. Kremer. The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, June 2003.
- [11] C.H. Hsu, M. Hsiao, and U. Kremer. Compiler-directed dynamic frequency and voltage scheduling. In *Workshop on Power-Aware Computer Systems*, November 2000.
- [12] Intel, Intel StrongARM SA-1100 Microprocessor Technical Reference Manual, March 1999. Available at <http://www.intel.com>
- [13] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *International Symposium on Low Power Electronics and Design (ISLPED-98)*, August 1998.
- [14] C.M. Krishna and Y.-H. Lee. Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems. In *Proceedings of the 6th Real Time Technology and Applications Symposium (RTAS'00)*, May 2000.
- [15] Lamport, L. *LaTeX User's Guide and Document Reference Manual*. Addison-Wesley, Reading, MA, 1986.
- [16] MachSuif : A Framework built on top of SUIF for building back-ends. <http://www.eecs.harvard.edu/~hube>
- [17] A. Manzak and C. Chakrabarti. Variable voltage task scheduling for minimizing energy or minimizing power. In *Proceeding of the International Conference on Acoustics, Speech and Signal Processing*, June 2000.
- [18] K. Roy. Leakage Power Reduction in Low-Voltage CMOS Design. In *IEEE International Conference on Circuits and Systems* , Pages 167-173, 1998
- [19] Siddharth Rele, Santosh Pande, Soner Onder, and Rajiv Gupta. Optimizing Static Power Dissipation by Functional Units in Superscalar Processors. In *International Conference on Compiler Construction* , LNCS 2304, Springer Verlag, pages 261-275, Grenoble, France, April 2002.
- [20] Sannella, M. J. *Constraint Satisfaction and Debugging for Interactive User Interfaces*. Ph.D. Thesis, University of Washington, Seattle, WA, 1994.
- [21] D. Shin, J. Kim, and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design and Test of Computers*, 18(2), March/April 2001.
- [22] Y. Shin, K. Choi, and T. Sakurai. Power optimization of realtime embedded systems on variable speed processors. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'00)* , pages 365{368, November 2000.
- [23] SUIF. Stanford University Intermediate Format. <http://suif.stanford.edu>
- [24] F. Xie and M. Martonosi and S. Malik. Compile time dynamic voltage scaling settings: Opportunities and limits. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation* , June 2003