Compiler-based vs. Hardware-based Power Gating Techniques for Functional Units

Yen-Hsiang Fang Yuan-Shin Hwang
Dept. of Computer Science & Engineering
National Taiwan Ocean University
Keelung 20224
Taiwan

Yi-Ping You Jenq-Kuen Lee
Department of Computer Science
National Tsing Hua University
Hsinchu 30013
Taiwan

***Abstract*—Reducing leakage power of embedded systems is essential as it constitutes an increasing fraction of the total power consumption in modern embedded processors. Power gating of functional units has been proved to be an effective technique to reduce leakage, and its various implementations can be categorized into compiler-based and hardware-based approaches. Hardware-only designs rely on specific circuits and microarchitectural designs to monitor instruction executions to determine when to power-gate functional units, whereas compiler-based methods attempt to exploit global information of programs and let compilers embed special instructions to turn on and off functional units. This paper compares the efficiencies of hardware and software techniques for power gating of functional units. Experimental results of the DSPstone benchmarks on Wattch show that the hardware-only approach is generally effective in reducing leakage, while the compiler-based approach occasionally performs better as the global knowledge of programs gathered by compilers would avoid incurring excessive power-gating on/off activities. This outcome suggests a better scheme: a hardware-based technique is deployed as the default power gating mechanism, and a compiler would intervene only when its analysis indicates the default method is inferior for certain application programs.**

[0] This research was supported by NSC grant NSC95-2221-E-019-012-MY3 and MOEA project 96-EC-17-A-01-S1-034

I. Introduction

Minimizing power dissipation is critical for embedded systems, and it can be achieved by techniques designed at the algorithmic, architectural, logic, and circuit levels [5]. Various hardware and software techniques have been proposed to reduce dynamic for power dissipation with architecture designs and/or software arrangement at instruction level [1], [6], [11], [19], [20], [24], [25], [26]. For example, several types of code rearrangement have been used to reduce the dynamic power, such as utilizing the value locality of registers [6], swapping operands for Booth multipliers [20], scheduling VLIW instructions to reduce the power consumption on the instruction bus [19], gating the clock to reduce workloads [11], [25], [26], utilizing cache subbanking mechanism [24], and buffering instructions nested within loops in a minicache [1]. Dynamic energy consumption is the main concern of these methods since it is the dominant form of power dissipation then.

As the minimum feature size gets smaller and more transistors are packed densely onto processors, static power dissipation due to leakage takes an increasing fraction of total power in processors. Static power dissipation increases about 5 times each generation since the total leakage current increases about 7.5 times [2], [4]. Consequently, it is estimated that leakage power will be the dominant form of power dissipation soon [8], [13], [15], [17], [18], [23]. In order to minimize the impact, power gating could be used to reduce leakage power [4], [12], [14]. Specifically, a functional unit should be shut down every time it enters a sufficiently long idle period and be turned back on before it is needed. Therefore, the key issue here is how to identify the onsets and conclusions of long idle periods and then perform power gating to turn off and on functional units without incurring significant performance penalties.

Power gating techniques can be categorized into hardware-based and compiler-based approaches. Hardware-based methods rely on special circuits and microarchitectural designs to monitor instruction executions in order to determine when to turn off and on functional units [12], [14]. The advantage of this approach is that programs can be run without any modifications, but its disadvantage is that it cannot exploit global information of programs. By contrast, compiler-based methods attempt to integrate architecture and compiler power-gating mechanisms [9], [21], [27], [29], [30], [31]. This approach involves compilers inserting specific instructions into programs to shut down and wake up components based on data-flow analysis or profiling information. The benefit of this approach is that power gating will be performed based on the insight knowledge of programs gathered by compilers, while its drawback is that additional instructions must be implemented by the hardware to power-gate on/off functional units and they must be explicitly embedded into programs. Furthermore, extra efforts must be taken to carefully merge power-gating instructions to avoid code size explosion if multiple functional units are guarded by power-gating circuits.

This paper compares the efficiencies of hardware- and compiler-based techniques for power gating of functional units. A straightforward implementation of power gating based on a hardware counter [12] is compared with a system that runs programs with compiler-embedded on/off instructions [29]. In addition, the *Sink-N-Hoist* compiler framework is used to merge several power-gating instructions into a single compound instruction, and hence the code size explosion issue is minimized [28]. Experimental results of the DSP-stone benchmarks on Wattch [3] show that the hardware-only approach generally outperforms the compiler-based method, but there are still a couple of programs that compiler-assisted approach works better. This outcome reveals that hardware-only techniques will reduce more leakage power when idle and active phases of programs are distinctive and last relatively long time intervals. However, programs with short active and idle periods would generate excessive on/off activities and hence incur significant overheads for the hardware-only approach. In contrast, global knowledge of programs would help compiler-assisted techniques to avoid such a pitfall.

This observation suggests a better solution. A hardware-based technique can be deployed as the default power gating mechanism, since generally it is very efficient in leakage reduction. However, a compiler would intervene when its analysis identifies that the default method might not be effective for certain application programs. It could either reorganize the code or adjust the parameters of the default hardware mechanism. This study indicates that better leakage reduction can be achieved by the cooperation of hardware-based and compiler-based approaches, and hence further compiler research will be needed in order for a compiler to determine when and how to intervene.

The rest of this paper is organized as follows. Section II portrays the architecture of the target platform. Section III briefly describes the compiler technique to embed and merge power-gating instructions and Section IV outlines the simple hardware-only implementation. Experimental results will be presented in Section V, and Section VI summarizes this paper and discusses the future work.

II. Machine Architecture

The instruction set architecture targeted by compiler-assisted techniques must support power-gating control at the component level. This paper focuses on reducing the power consumption of certain components by invoking power-gating technology. Power gating is analogous to clock gating, except that devices are powered off by switching off their supply voltage rather than the clock. This can be implemented by forcing transistors to be off or using MTCMOS (multi-threshold voltage CMOS technology) to increase the threshold voltage [4], [12], [14], [22].

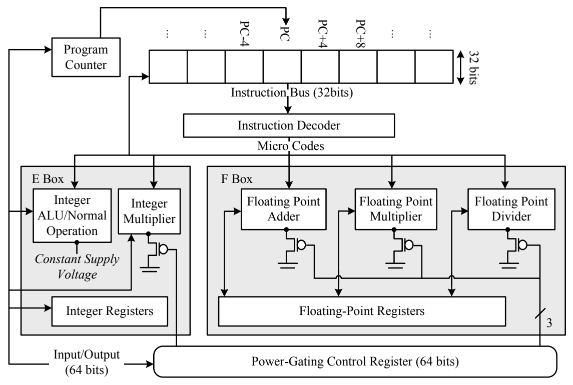

Fig. 1. DEC Alpha 21264 architecture with power-gating support

Figure 1 illustrates an example of the target machine architecture based on a DEC Alpha 21264 processor with an instruction fetch, issue, and retire unit (Ibox), a block of integer functional units (Ebox), a block of floating-point functional units (Fbox), a memory reference unit (Mbox), and an external cache and system interface unit (Cbox) [7].

In the adapted DEC Alpha 21264 architecture model, the Ebox and Fbox were equipped with power-gated functions. The power state of each unit is controlled by the 64-bit integer power-gating control register (PGCR). In this case, one bit is used for the integer multiplier unit and three bits are used for the floating-point functional units. Setting the power-gating bit true powers on the corresponding module, and clearing the bit to zero powers off the corresponding module immediately in the following clock cycle. A new instruction was implemented to control units with the power-gated function by moving the appropriate value from a general-purpose register to the PGCR. The integer ALU unit is always powered on since it takes the responsibility for moving data to the PGCR.

III. Compiler-Assisted Power-Gating Control Placement

This section reviews a compiler approach that statically analyzes the activities of power-gating candidates of the input programs and inserts power-gating instructions at appropriate positions with the consideration of code size issues [29], [30], [28]. The proposed framework, called *Leakage-Power-Reduction Framework*, is operated with three major phases:

(1) *Component-Activity Data-Flow Analysis (CADFA)*,
which estimates the activities of the power-gating candidates within a given program,

(2) *Power-Gating Instruction Scheduling*, which determines whether, where, and when power-gating controls should be employed so as to reduce energy dissipation, and

(3) *Sink-N-Hoist Analysis*, which attempts to sink (postpone) power-off operations and hoist (advance) power-on operations for increasing the opportunity to merge power-gating instructions into compound instructions and thus reduce program code size.

Figure 2 sketches the process scenarios, corresponding to the above three phases, of a motivating example in the view of the compiler approach with the assumption that three floating-point units (an arithmetic logic unit, a multiplier, and a divider) are considered as the power-gating candidates. Each plot of Figure 2

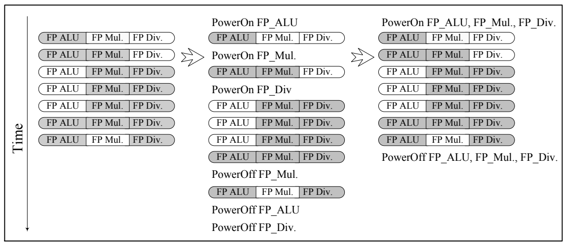

Fig. 2. An example of power-gating controls over floating-point (FP) units (the shaded components are those in use)

shows the activities of the power-gating candidates, represented as boxes, in the timeline and also the placement of power-gating instructions. The leftmost plot shows the activity information produced by CADFA, where a shaded box represents a unit which is in use at that time, and is simply the case without power-gating controls; the middle plot shows the case when Power-Gating Instruction Scheduling is applied; and the rightmost one shows the case when Sink-N-Hoist Analysis is involved.

Basically, the Leakage-Power-Reduction Framework is performed with a set of data-flow equations and the control-flow graph of the input program. In CADFA, *component-activities*, the activities (active or inactive) of components, are propagated with union as the meet operation. A component-activity is generated at a block if a component is required for processing and it is killed if the component is released from the process. Once the activity information of components has been obtained, power-gating instructions can be inserted into programs at the appropriate points (i.e., the beginning and end of an inactive block) to power off and on unused components so as to reduce the leakage power. However, both shut-down and wake-up procedures are associated with an additional penalty, especially the latter due to peak voltage requirements. Power-Gating-Instruction Scheduling is then performed and takes account of the influence of conditional branches in programs — the time required to instigate power-gating controls on components is related to the number and complexity of program branches. The process seems to be done after the phase of Power-Gating-Instruction Scheduling. However, there are concerns about the amount of power-control instructions being added to programs with the increasing amount of power-gating candidates in a system-on-a-chip (SoC) design platform for embedded systems. Therefore, Sink-N-Hoist Analysis was proposed to generate balanced scheduling of power-gating instructions.

The main idea of Sink-N-Hoist analysis is to reduce the problem of too many instructions being added with code-motion techniques. The approach attempts to merge several power-gating instructions into one compound instruction by ‘sinking’ power-off instructions and ‘hoisting’ power-on instructions; that is, postponing the issuing of power-off instructions and bringing forward the issuing of power-on instructions. For instance, a power-off instruction can be postponed for several cycles to be merged with adjacent power-off instructions. This will result mainly in improvements to the code size, but also in performance and energy via grouping effects. Similarly to CADFA, Sink-N-Hoist Analysis is based on a set of data-flow equations to collect the information for the code motion of power-gating instructions, such as the information of possible positions to issue for each power-gating instruction and the information which power-gating instructions should be merged.

IV. Hardware-Only Approach

This section recaps the time-based power gating technique that is commonly deployed by the hardware-only approach [12]. Instead of executing explicit power-gating instructions, hardware-only techniques rely on logic circuits to detect the onsets and conclusions of sufficiently long idle periods and then to power-gate off/on functional units. The easiest way is to turn off a functional unit after it is idle over a certain number of cycles. Similar techniques have been used for reducing leakage of caches [10], [16].

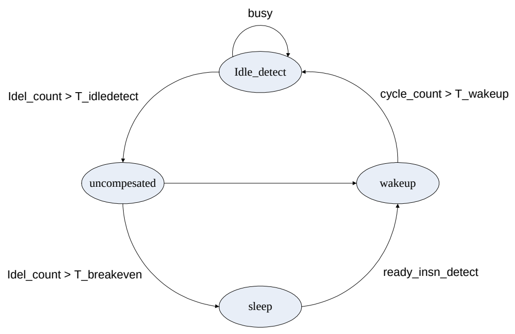

Fig. 3. State Diagram (Adapted from Figure 5 in [12])

In order to model this tactic, a state diagram shown in Figure 3 can be associated with every functional unit. The *idle_detect* state is the normal, active state, when the functional unit is ready for execution. A functional unit will be power-gated off and enter the *uncompensated* state after being idle for more than $T_{idledetect}$ cycles. If it is waked up during this state, it will miss the break-even point as the overhead of power-gating is greater than the leakage reduction. By contrast, power-gating will save leakage once the idle period lasts longer than $T_{breakeven}$, and the functional unit will enter the *compensated* state. The longer the functional unit stays in the *compensated* state, the more leakage power it will save. If an instruction is ready for execution when its corresponding functional unit is in the *compensated* or *uncompensated* state, the unit will move to the *wakeup* state and stay there for T_{wakeup} cycles before entering the active *idle_detect* state. T_{wakeup} is the latency of powering up functional units.

According to the state diagram, the efficiency of the hardware-based power-gating technique is determined by the three parameters, $T_{idledetect}$, $T_{breakeven}$, and T_{wakeup}. The first parameter, $T_{idledetect}$, determines how aggressive the power-gating mechanism would be. It is the only parameter among these three that can be dynamically adjusted, since the other two $T_{breakeven}$ and T_{wakeup} are fixed once the physical circuit design technique is chosen. A small $T_{idledetect}$ would identify more idle periods but it might cause performance degradation if functional units have to be frequently turned back on before the break-even point. On the contrary, a large $T_{idledetect}$ might miss many opportunities to power-gate functional units.

Parameter	Value
Clock	600 MHz
Process parameters	0.10 μm, 1.9 V
Instruction issuing	In-order
Decode width	8 instructions/cycle
Issue width	8 instructions/cycle
Commit width	8 instructions/cycle
RUU size	128
LSQ size	64
Functional units	4 integer ALUs 1 integer multiply/divide unit 4 FP ALUs 1 FP multiply/divide unit
Register files	32 64-bit integer registers 32 64-bit FP registers 1 64-bit power-gating control register (PGCR)

TABLE I
BASELINE PROCESSOR CONFIGURATION

V. Experimental Results

A. Setup

The target platform is a DEC-Alpha-compatible architecture with the power-gating controls and instruction shown in Figure 1, and experiments are conducted on the Wattch simulator with 0.10-μm process parameters and 1.9V V_{DD} [3]. Table I summaries the baseline configuration of the simulator. By default, the simulator performed out-of-order executions. The '-issue:inorder' option is used in the configuration so that instructions would be executed in order to ensure the correctness of power-gating controls. Nevertheless, the software-assisted method can also be applied to out-of-order issue machines if the additional hardware supports proposed in [30] are employed. The benchmarks used in this paper are taken from the floating-point version of the

DSPstone benchmark suite [32]. The average IPC (instructions per cycle) of the benchmarks is 0.36 with the configuration in Table I.

Since Wattch does not model leakage at the component level per se, this paper assumes that leakage power makes up 10% of total power consumption. Furthermore, each wake-up operation is assumed to have a 20-cycle latency (i.e. T_{wakeup} = 20 cycles) and to dissipate ten times of the leakage power. Similarly, every turn-off instruction consumes twice of normal leakage energy. The energy consumption of fetching and decoding a power-gating instruction was assumed to be two times the leakage power. In addition, normalized leakage of the target platform will be computed relative to the leakage measured on Wattch *cc3* with only clock-gating mechanism.

B. Performance Evaluation

This section will evaluate the performance of four hardware-only power-gating configurations, i.e. $T_{idledetect}$ = 16, 48, 64, and 96 cycles, and two software-assisted power-gating designs, namely CADFA and Sink-N-Hoist.

Leakage

Figure 4 illustrates that the normalized leakage of power-gated functional units for the DSPstone benchmarks under the above various software and hardware power-gating configurations. Basically the hardware-only configurations reduce much more leakage than the software-assisted designs. On average the hardware-only configurations lower the leakage power down to about 10%, while the software-assisted designs cut the leakage down to around 30%. However, CADFA and Sink-N-Hoist still manage to outperform the hardware-only configurations for the two benchmarks $fir2dim$ and $matrix1$, and only dissipate only about half of leakage. The main reason is that in some occasions hardware-only configurations might be too eager or too lukewarm due to the lack of global information of the programs, whereas the compiler-based techniques tend to know when to power-gate on/off functional units.

Power gating functional units incur overheads since turning off and on circuits do take energy. The red portions of bars in the figure denote such energy overheads, which are reasonably small for all hardware and software configurations. The hardware-only configurations with small $T_{idledetect}$ cycles introduce smaller overheads than the hardware-only configurations with large $T_{idledetect}$ numbers. The compiler-based approach usually suffers more overheads than the hardware-only method for it has to execute instructions explicitly to power-gate functional units. However, such overheads can be reduced by merging the power-gating instructions, as shown by the result that Sink-N-Hoist incurs much less penalty than CADFA.

Run Time Impact

Turning off/on functional units to reduce leakage power will definitely incur performance penalties since several to tens of cycles are needed to power on/off circuits. Figure 5 shows the performance impact of the various power-gating configurations. Hardware-only implementations commonly suffer higher performance degradation than compiler-based techniques, especially for the configurations with small $T_{idledetect}$ numbers due to excessive power-gating on/off activities. On average they might slow down the execution of the DSPstone benchmarks by 10% to 20%. In contrast, compiler-based designs incur only negligible performance degradation, roughly 2% on average.

Figure 4 and Figure 5 reveal that hardware-only configurations with reasonably large $T_{idledetect}$ cycles seem to be a favorable choice as they can significantly reduce leakage without incur considerable performance penalties.

Wakeup Latency

The hardware-only approach will be more effective if the wakeup latency T_{wakeup} can

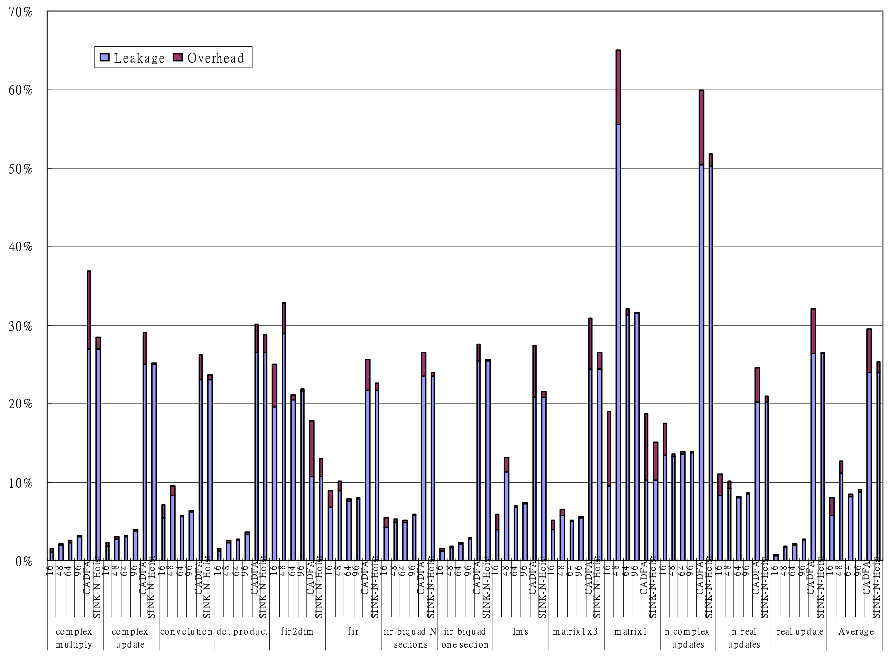

Fig. 4. Normalized Leakage Power

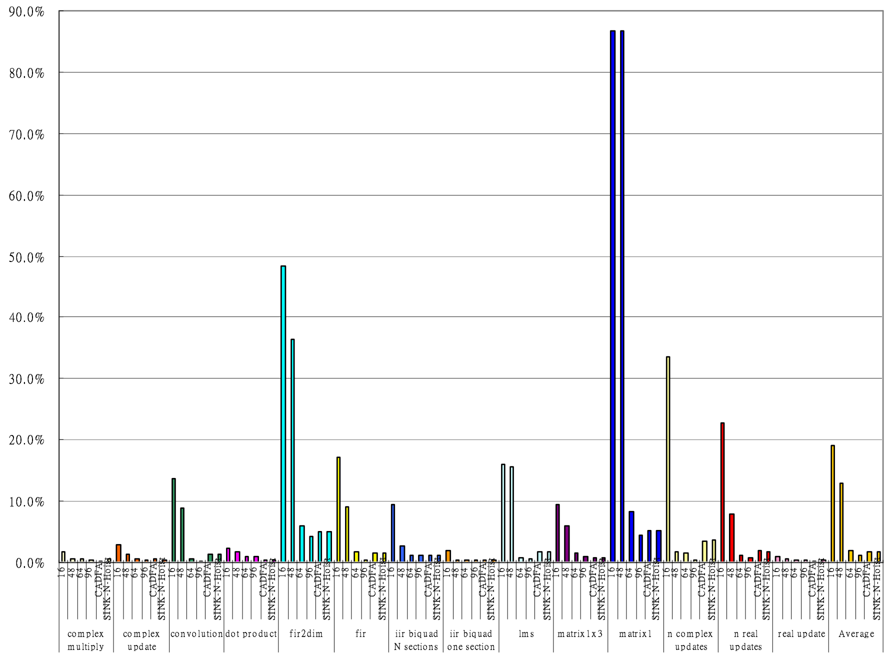

Fig. 5. Average Run Time Impact

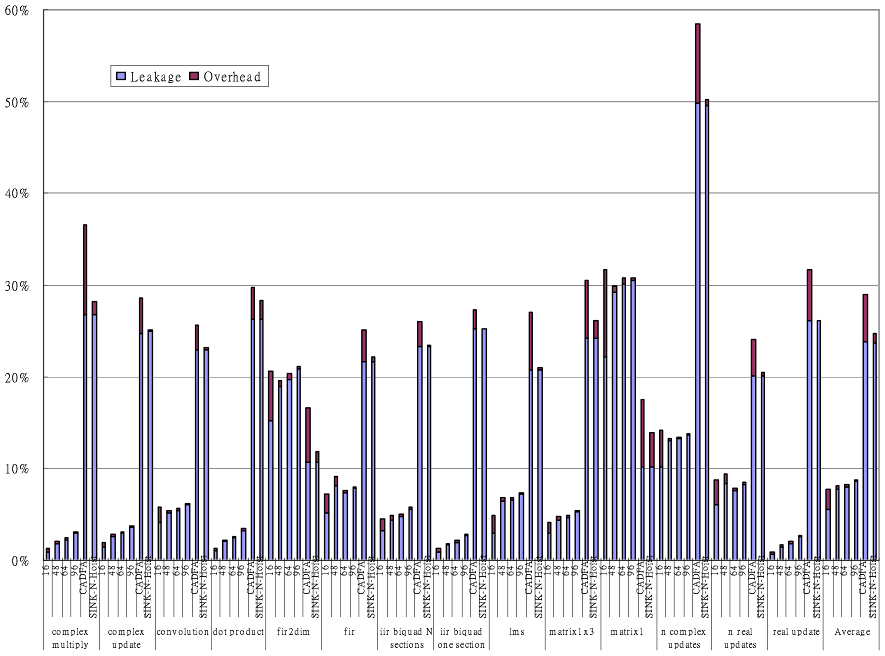

Fig. 6. Normalized Leakage Power (T_{wakeup} = 10 Cycles)

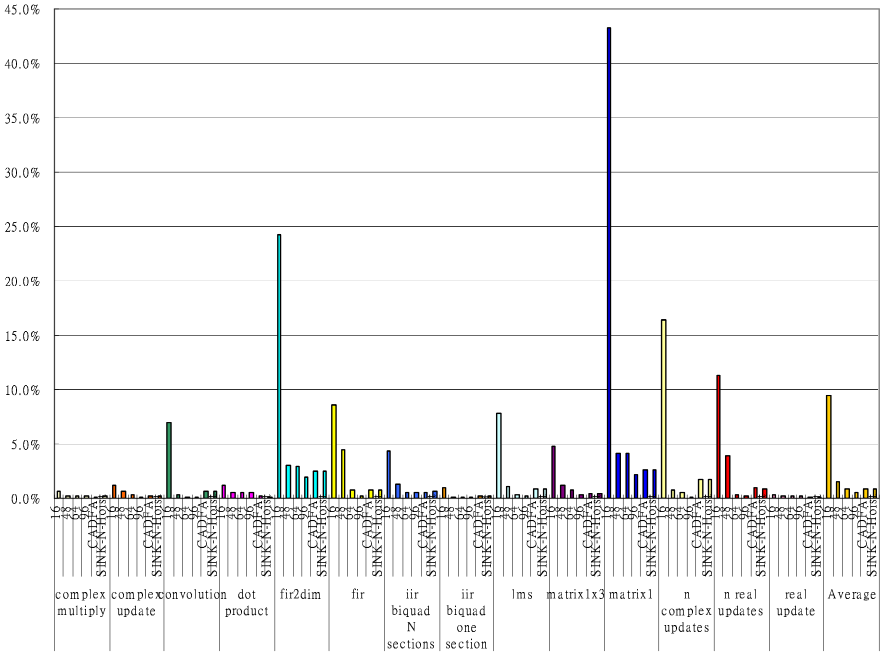

Fig. 7. Average Run Time Impact (T_{wakeup} = 10 Cycles)

be reduced. Figure 6 and Figure 7 show the effects of cutting the wakeup latency by half to 10 cycles. Leakage and runtime impact can be further reduced as functional units sit idle for fewer cycles waiting for their circuits to be powered back on.

VI. Summaries and Future Work

This paper compares the efficiencies of hardware- and compiler-based techniques for power gating of functional units. A straightforward implementation of hardware power gating mechanism is compared with a system that runs programs with compiler-embedded on/off instructions. Experimental results of the DSPstone benchmarks on Wattch show that the hardware-only approach generally performs better than the compiler-based method, but compiler-based approach still manages to outperform in some occasions. This outcome reveals that hardware-only techniques might generate excessive on/off activities when the program executions do not fit the pattern, while global knowledge of programs would help compiler-based techniques to avoid such a pitfall. This observation suggests a combination of hardware-based and compiler-based approaches: a hardware-based technique can be deployed as the default power gating mechanism, and a compiler would intervene only when its analysis indicates the default method is inferior for certain application programs.

A compiler now has to take the responsibility of determining if the default hardware mechanism might inappropriately turn on and off functional units. If could adjust the parameters of the default mechanism, or even reorganize the instructions when adjusting parameters alone would not be enough. There is an on-going project here that studies how to perform analysis on the binary executable codes of application programs in order to decide if the compiler should intervene. In addition, this project also investigates how the object codes can be reorganized to fully exploit the default power gating mechanism.

References

[1] Nikolaos Bellas, Ibrahim N. Hajj, and Constantine D. Polychronopoulos. Architectural and compiler techniques for energy reduction in high-performance microprocessors. *IEEE Transactions on Very Large Scale Integration Systems*, 8(3):317–326, June 2000.

[2] Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the International Symposium on Computer Architecture*, pages 83–94, Vancouver, Canada, June 2000.

[4] J. Adam Butts and Gurindar S. Sohi. A static power model for architects. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, pages 191–201, Monterey, California, December 2000.

[5] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, 1992.

[6] Jui-Ming Chang and Massoud Pedram. Register allocation and binding for low power. In *Proceedings of the Design Automaton Conference*, pages 29–35, San Francisco, California, USA, June 1995.

[7] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*. 1999.

[8] Brian Doyle, Reza Arghavani, Doug Barlage, Suman Datta, Mark Doczy, Jack Kavalieros, Anand Murthy, and Robert Chau. Transistor elements for 30nm physical gate lengths and beyond. *Intel Technology Journal*, 6(2):42–54, May 2002.

[9] Steven Dropsho, Volkan Kursun, David H. Albonesi, Sandhya Dwarkadas, and Eby G. Friedman. Managing static leakage energy in microprocessor functional units. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO'02)*, pages 321–332, Istanbul, Turkey, November 2002.

[10] Krisztian Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 148–157, 2002.

[11] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Proceedings of the IEEE Symposium on Low Power Electronics*, pages 8–11, San Diego, California, USA, October 1994.

[12] Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, and Pradip Bose. Microarchitectural techniques for power gating of execution units. In *Proceedings of the 2004 International Symposium on Low Power Electronics and*

Design (ISLPED'04), pages 32–37, Newport Beach, California, USA, August 2004.

[13] Robert Jones. Modeling and design techniques reduce 90 nm power. *EE Times 08/06/2004*, 2004. Available online at http://www.eetimes.com/showArticle.jhtml?articleID =26806450.

[14] J. T. Kao and A. P. Chandrakasan. Dual-threshold voltage techniques for low-power digital circuits. *IEEE Journal of Solid-State Circuits*, 35(7):1009–1018, 2000.

[15] Tanay Karnik, Shekhar Borkar, and Vivek De. Sub-90nm technologies – challenges and opportunities for CAD. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'02)*, pages 203–206, San Jose, California, USA, November 2002.

[16] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage. In *Proceedings of the 28th annual International Symposium on Computer Architecture*, pages 240–251, 2001.

[17] Nam Sung Kim, Todd Austin, David Blaauw, Trevor Mudge, Krisztian Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage current: Moore's law meets static power. *IEEE Computer*, 36(12):68–75, 2003.

[18] Nam Sung Kim, Krisztian Flautner, David Blaauw, and Trevor Mudge. Circuit and microarchitectural techniques for reducing cache leakage power. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(2):167–184, February 2004.

[19] Chingren Lee, Jenq Kuen Lee, Ting-Ting Hwang, and Shi-Chun Tsai. Compiler optimizations on VLIW instruction scheduling for low power. *ACM Transactions on Design Automation of Electronic Systems*, 8(2):252–268, 2003.

[20] Mike Tien-Chien Lee, Vivek Tiwari, Sharad Malik, and Masahiro Fujita. Power analysis and minimization techniques for embedded DSP software. *IEEE Transactions on Very Large Scale Integration Systems*, 5(1):123–133, March 1997.

[21] Siddharth Rele, Santosh Pande, Soner Onder, and Rajiv Gupta. Optimizing static power dissipation by functional units in superscalar processors. In *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*, pages 261–275, Grenoble, France, April 2002.

[22] K. Roy and S. C. Prasad. SYCLOP: Synthesis of CMOS logic for low power applications. In *Proceedings of the IEEE International Conference on Computer Design*, pages 464–467, Cambridge, Massachusetts, USA, October 1992.

[23] Semiconductor Industry Association. International technology roadmap for semiconductors. 2004.

[24] Ching-Long Su and Alvin M. Despain. Cache designs for energy efficiency. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, pages 306–315, Los Angeles, California, USA, January 1995.

[25] V. Tiwari, R. Donnelly, S. Malik, and R. Gonzalez. Dynamic power management for microprocessors: A case study. In *Proceedings of the International Conference on VLSI Design*, pages 185–192, Hyderabad, India, January 1997.

[26] V. Tiwari, D.Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez. Reducing power in high-performance microprocessors. In *Proceedings of the Design Automaton Conference*, pages 732–737, San Francisco, California, USA, June 1998.

[27] Hongbo Yang, R. Govindarajan, Guang R. Gao, George Cai, and Ziang Hu. Exploiting schedule slacks for rate-optimal power-minimum software pipelining. In *Proceedings of the 3rd workshop on Compilers and Operating Systems for Low Power (COLP'02)*, Charlottesville, Virginia, USA, September 2002.

[28] Yi-Ping You, Chung-Wen Huang, and Jenq Kuen Lee. A sink-n-hoist framework for leakage power reduction. In *Proceedings of the ACM International Conference on Embedded Software (EMSOFT'05)*, pages 124–133, 2005.

[29] Yi-Ping You, Chingren Lee, and Jenq Kuen Lee. Compiler analysis and supports for leakage power reduction on microprocessors. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC'02)*, pages 63–73, Washington, D.C., USA, July 2002. Lecture Notes in Computer Science, Vol. 2481, Springer Verlag.

[30] Yi-Ping You, Chingren Lee, and Jenq Kuen Lee. Compilers for leakage power reduction. *ACM Transactions on Design Automation of Electronic Systems*, 11(1):147–164, January 2006.

[31] W. Zhang, Mahmut T. Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and V. De. Compiler support for reducing leakage energy consumption. In *Proceedings of the 6th Design Automation and Test in Europe Conference (DATE'03)*, pages 1146–1147, Messe Munich, Germany, March 2003.

[32] V. Zivojnovic, J. Martinez, C. Schlager, and H. Meyr. DSPstone: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing and Technology (ICSPAT'94)*, pages 715–720, Dallas, Texas, USA, October 1994.