Saving Register-File Leakage Power by Monitoring Instruction Sequence in ROB

Wann-Yun Shieh* and Hsin-Dar Chen

Department of Computer Science and Information Engineering
Chang Gung University, Taiwan
wyshieh@mail.cgu.edu.tw

Abstract. Modern portable or embedded systems support more and more complex applications. These applications make embedded devices require not only low power-consumption, but also high computing performance. To enhance performance while hold energy constraints, some high-end embedded processors, therefore, adopt conventional features to exploit instruction-level parallelism and increase clock rates. The reorder buffer (ROB) and the register file are the two most critical components to implement these features. The cooperation of them, however, causes serious leakage power, especially for a large register file. In this paper, we propose a pure hardware approach to reduce the leakage power for the register file, such that more complex features (e.g., out-of-order execution, speculation execution, etc) can be applied to high-end embedded processors. In the proposed approach, we design a monitoring scheme in the pipeline datapath to identify the timing of powering up or powering down a register. Simulation results show that our approach saves at least 50% power consumption of the register file, with almost negligible performance lost.

Keywords: register leakage power, dynamic voltage scaling (DVS), reorder buffer, high-end embedded processor.

1 Introduction

Modern portable or embedded systems support more and more complex applications, e.g., multimedia displaying, wireless communications, and pervasive computing etc. These applications make embedded devices require not only low power-consumption, but also high computing performance. To enhance performance while hold energy constraints, some high-end embedded processors, therefore, adopt some conventional features to exploit instruction-level parallelism and increase clock rates. For example, ARMULET3 [1] and Intel Xscale [2] processors both implement multiple functional units in datapath such that they can issue more than one operation in a cycle. Also, both of them support out-of-order-execution for lower CPI.

There are two implicit problems to adopt these features in an embedded processor. First, it may require more hardware resources, such as registers, functional units etc, in a processor to execute a great deal of operations. Second, it may require some smarter techniques to dynamically resolve data dependency. Many efficient

* Corresponding author.

approaches have been proposed to resolve above problems in conventional general-purpose processors. For example, register renaming is used to allow instructions to write their results from functional units into any free registers for resolving name dependency due to out-of-order execution. Another example is speculation execution. It predicts program execution directions after branch instructions, and fetches the instructions speculatively for execution from predicted directions.

The reorder buffer (ROB) and the register file are the two most critical components to implement above features. The ROB maintains a precise state after an interrupt or exception occurs, and flushes the instructions along wrong-execution path. The register file holds the values committed from the ROB, and waits for references by later instructions. A register would become free if its name is redefined in the later instruction.

Unfortunately, such cooperation will make serious leakage power for register file. This is because in order to avoid missing committed values from the ROB, the register file has to stay in full-power mode in any pipeline stages. Some unused registers, therefore, waste energy, including the registers that would store the values has not been produced yet, or store the values never be referenced again in the future. So we can conclude that there are two kinds of register's leakage power occurring during program execution. The first one, we call it the Intra-Pipeline Leakage Power, occurs between the instruction's decode stage and the commit stage. During this interval, the register stores nothing, but has to wait for instruction commitment. The second one, we call it the Seldom-Used Leakage Power, occurs when the register value is no longer to be used, or would be used again after a long period. In this case, all nearby instructions (subsequent consumers) that need that value as source operands have already read it (no matter from the ROB or from the register), or no consumer instructions appear again for a long period. More seriously, some short-lived register-values [9] cause registers never being referenced once (i.e., seldom-used), even after instruction decode stage, but these registers still have to be allocated and wait for those values' commitment.

There are many researches talking about saving ROB's leakage power, but few researches talking about saving register file's leakage power. Moreover, most researches talking about saving register file's leakage power have focused on using compiler-aided approaches [6][7]. In these approaches, the compiler analyzes a program's register-usage with some profiling data, and then inserts voltage-scaling instructions into the program to control supply voltages for registers. Because the compiler cannot exactly know instruction execution sequence, these approaches cannot be applied to out-of-order execution processors.

In this paper, we propose a pure hardware approach to reduce the leakage power of register file, such that more complex features (e.g., out-of-order execution, speculation execution, etc) can be applied to high-end embedded processors. In the proposed approach, we design a monitoring scheme in the pipeline datapath to identify the timing of powering up or powering down a register. The objective of this monitoring scheme is to find out which register-values causing leakage-power (described above), and to maximize the benefits from register supply-voltage changes. The supply-voltages to a register in this paper include the active mode (1 Volts), drowsy mode (0.3 Volts), and destroy mode (0 Volts), as defined in [3].

To monitor a value's usage in pipeline datapath, we modify register file by adding only one bit to each register and modify the ROB structure by adding only two bits to each entry. The bit adding to a register is used to indicate that the value stored in this register is less usage (seldom-used) or not. Two bits adding to a ROB entry indicate if a value would be referenced later or become useless. If a value becomes useless before being written to a register, this value (i.e., short-lived value) will be filtered out from unnecessary commitment to the register.

In order to implement the proposed approach, there are three important issues. First, what information should we monitor to power up a register on time? If a register cannot be powered up to the active mode (1 Volts) before the time of being referenced (register-read) or being committed (register-write), stall cycles would happen. Second, what information should we monitor to power down a register for maximal energy-saving? If we directly power off a register that would be referenced later in destroy mode (0 Volts), a program exception or error will thus occurs. Finally, how to earn extra power-saving for a register by DVS during speculation execution but guarantee processor states can be recovered under branch misprediction situations. If a register is allocated to a speculative instruction, which supply-voltages should be changed to? All of these issues will be considered in designing the proposed approach.

The rest of this paper is organized as follows. Section 2 introduces and discusses related works. Section 3 discusses two primary leakage powers in detail. Section 4 analyzes the timing a register should be powered up or down and implementation. Section 5 shows the simulation environment and simulation results. Section 6 gives conclusion.

2 Related Work

In this section, we discuss previous studies talking about low-power register file design. Also, we discuss the studies about raising the utilization of register file, which could be a guideline to design our proposed approach.

2.1 Compiler-Aided Power Saving Approaches

Several researches save leakage power by using software approaches [3][6][7]. These approaches use some specific registers, which called the power state registers, to control supply voltages to a hardware component. They analyze the profiling of the program to identify which hardware components during which instructions sections are worthy to be powered down, and then insert voltage-change instructions into the program. There are some problems for these approaches. First, different compilers or different data inputs make profiling data varied. Second, inserting extra voltage-change instructions results in additional power and performance penalty. Third, for out-of-order processors, it is impossible to know the exact sequence of instruction execution by software approach. Our goal is to allow more features for out-of-order-execution processors to be applied to embedded systems. Therefore we adopt the pure hardware approach, which is more suitable for out-of-order processors.

2.2 Late Allocation and Early Release of Physical Registers

The main idea of the paper [8] is to delay register allocation until a value actually needs to be written into a register, and to early release a register before its value commits if that register name has been redefined. The objective of this design is to decrease the occupation time for a register in the pipeline. Our design is interested in finding unused registers to save leakage power. The design of the late-allocation approach reminds us that a register would be useless until the commit stage. So it is likely to design a power saving scheme to reduce a register's leakage power before the commit stage. On the other hand, the design of the early-release approach reminds us that we have another opportunity to power down a register if its value will never be referenced again.

3 Leakage Power Analysis

For each register, we define two primary leakage powers, called the intra-pipeline leakage power and the seldom-used leakage power. For the intra-pipeline leakage power, we discuss in which pipeline stages we should power on or power down register to save leakage power. For seldom-used leakage power, we discuss two major situations to cause a register becoming seldom-used. Then we discuss why seldom-used registers result in serious leakage power, even than the intra-pipeline leakage power.

3.1 Intra-pipeline Leakage Power

For a typical pipeline of an out-of-order processor (Here we use Intel P6 processor [10] as example.), it is clearly that the intra-pipeline leakage power occurs between the rename and the commit stage. Also, we found that whether an instruction stays at the rename or the commit stage can be detected by monitoring that instruction in the ROB. So, it is useful to modify the ROB to power down a register at the rename stage, and power on it at the commit stage to save the intra-pipeline leakage power. Note that, due to the time delay of powering on a register, there is no enough time to perform a register's voltage change as well as the value's commitment at the same cycle. So, to avoid performance penalty, we should actually power-on a register earlier than the commit stage, i.e., the writeback stage.

3.2 Intra-pipeline Leakage Power

There are two primary situations to cause a register becoming seldom-used. First, because register pressure is not always high through whole program sections, some registers are active but seldom-used during some sections. Second, some other registers may have special purposes, not for storing operation results. For instance, a register may be used to store function return value. However, if a callee function is large or its execution time is long due to intensive memory stalls, that register will be used again after a long period; it becomes seldom-used thereby. There are other specific registers, such as the stack pointer register, the global pointer register, and the return address register, etc. All of them may possibly become seldom-used due to the characteristics in a program.

4 Approach

In this section, we first show the approach to save the intra-pipeline leakage power, and then show the approach to reduce the seldom-used leakage power.

4.1 Approach for Intra-pipeline Leakage Power

To save the intra-pipeline leakage power, we have to clarity what power controlling decisions we should make at different stages. As described in Section 3, we can power down a register to the destroy mode (0 Volts) at the rename stage. The exception is when we consider the speculation execution, using the destroy mode will loss a register's previous value, which may need to be restored after branch misprediction. So we use the drowsy mode (0.3 Volts) to keep the register value alive instead in this case. On the other hand, we should power on a register to the active mode (1 Volts) at the writeback stage. But when we consider filtering unnecessary commits, some registers, which has been redefined by the subsequent instructions in the ROB, do not need stay in the active mode at the writeback stage. That is, the registers, which have been allocated to those short-lived-values, can still stay in low-power mode (the destroy or drowsy mode), without being waked up at the writeback stage. Note that if a destination register has been redefined but this redefined instruction is fetched along a branch-prediction path, we cannot make sure the value which will be stored in this destination register is short-lived or not. So, this destination register still has to be waked up at the writeback stage.

After the commit stage, there are three cases for a register that can be considered to save leakage power further. First, for the registers committed with a speculative redefined instruction in the ROB, we can power down them in the drowsy mode to save more power. Second, for the registers committed without being redefined, we should keep them in the active mode for later references by incoming consumers. Finally, for unnecessary-committed registers we can power off this registers in the destroy mode to save the largest leakage power. Note that when a branch misprediction occurs, we should power on all drowsy registers to restore the precise state for the pipeline. If all branch instructions have been committed, we can power off all of the drowsy mode registers into the destroy mode.

To implement the approach discussed above for saving the intra-pipeline leakage power, we need to modify the architecture of the ROB to monitor instruction sequence, and add a voltage-scaling controller for each register. For the ROB, we add two bits to each ROB entry. One bit is used to indicate that the destination register of this entry has been redefined or not. Another bit is used to identify whether a branch instruction exists between this entry and the later redefined entry (i.e., speculative redefined instruction). Because the ROB is a CAM-like (Content Addressable Memory) structure, both of these two bits can be updated by each incoming instruction. For each register, we use a two-bit power-state-register to record its supply voltage. The value of the power-state register will be determined by a voltage-scaling controller which performs voltage scaling strategies described above according to the ROB's monitoring results.

By such a pure hardware approach, we found that there is no performance penalty in writing a value into a register with DVS controls because we power on the destination register before an instruction commits. Also, there is no performance penalty in reading an operand from a register because we power off only the registers that has been redefined in the ROB, and power on all registers that are in the drowsy mode when a branch misprediction occurs.

4.2 Approach for the Seldom-Used Register Leakage Power

To save the seldom-used-register's leakage power, we use a decay approach [11] to identify which registers are seldom-used. We setup a period (decay cycle) to observe whether a register has been accessed or not during this period. If a register has not been accessed during this period, we can suppose that it is a seldom-used register and then power down this register in the drowsy mode to save leakage power. (It cannot be powered down to the destroy mode further because no redefined instructions occur in this period.) It is very easy to implement the decay approach by using a counter to setup the decay period and adding an extra bit for each register to indicate whether it has been accessed or not in the period.

The problem of this approach is that the register we have power-down after decayed may need to be read again in the future. In this case, we need one cycle delay to power on this register in the active mode. The decay period we select may impact such performance penalty. A large decay period can avoid a register from being powered-down before next reference, but waste much leakage power to wait for decay. On the other hand, a small decay period may save more leakage power, but results in performance penalty described above. Therefore, the setup of the decay period would be a trade-off, and could be selected by individual program.

5 Simulation Results

In this section we first present the evaluation configuration, including evaluation environment and evaluation parameters. Then we present power-saving results and performance penalty.

5.1 Experimental Framework

We carry out our experiments and analysis by using the Simplescalar [12] simulator, and obtain our power model by Wattch [13]. We assume the proposed approach is

Table 1. Architectural Configuration of Simulated Processor

Machine width	4wide fetch, 4-wide issue, 4-wide commit
Window size	32 entry fetch queue, 32/64/128 entry ROB, 32 entry load/store queue
L1 I-cache	32KB, 4-way set-associative, 32byte line, 2 cycles hit time
L1 D-cache	32KB, 4-way set-associative, 32byte line, 2 cycles hit time
L2 Cache combined	512KB, 4-way set-associative, 64byte line, 4 cycles hit time
BTB	1024 entry, 2-way set-associative
Memory	128 bit wide, 12 cycles first chunk, 2 cycles interchunk
TLB	64 entry (I), 128 entry (D), 4-way set-associative, 30 cycles miss latency
Function Units and Latency (total/issue)	4 Int Add(1/1), 1 Int Mult(3/1) / Div (20/19), 2 Load/Store(2/1), 4 FP Add(2), 1 FP Mult(4/1) / Div(12/12) / Sqrt(24/24)

implemented in a 12-stage pipeline, like Intel P6. Table 1 shows our architectural configurations of the simulator. In addition, eleven benchmarks from SPEC2000 suite are used: seven for integer (gzip, bzip2, gcc, mcf, parser, twolf, vortex) and four for floating point (ammp, art, equake, mesa).

5.2 Results

5.2.1 Power Saving

Figure 1 shows register file power consumption for integer benchmarks and floating point benchmarks. In integer benchmarks, we found the leakage power of intra-pipeline save 7%~40% power. In floating benchmarks, we save 42~69% intra-pipeline leakage power. Our Intra-pipeline approach performs well in floating point benchmarks than integer benchmarks due to longer execution. When intra-pipeline leakage power co-works with the decay approach of the 4-cycle decay period, we earn more power-saving, about 40%~60%. It shows that the seldom-used registers would be the primary source of leakage power.

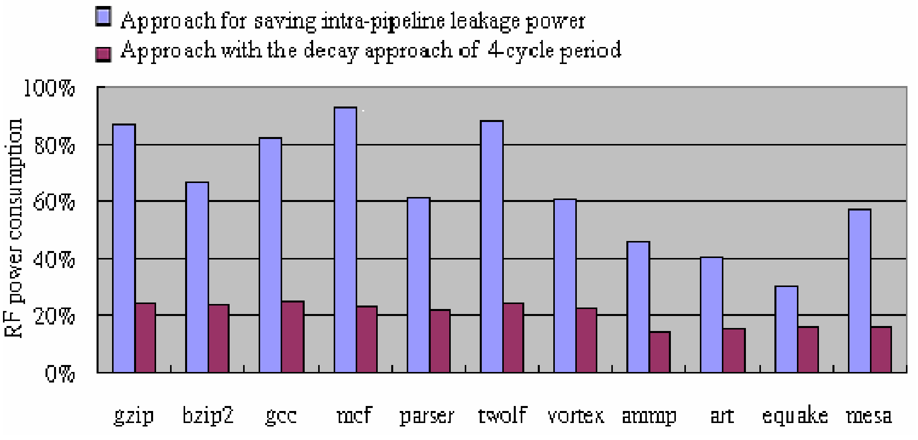

Fig. 1. Register file power consumption for eleven benchmarks

Figure 2 shows the register file power consumption for different decay periods. Even with 128-cycle period, we can save at least about 50% leakage power. In Figure 2, we found that power consumption increases when the period length become large. This result is consistent with the discussion in Section 4.2, in which a large decay period saves less power than a small period because large periods delay some seldom-used registers entering drowsy mode, and vice versa.

Figure 3 shows the saving intra-pipeline leakage power under different ROB sizes (i.e., 32-entry vs. 64-entry). Although a large ROB supplies more view to monitor the

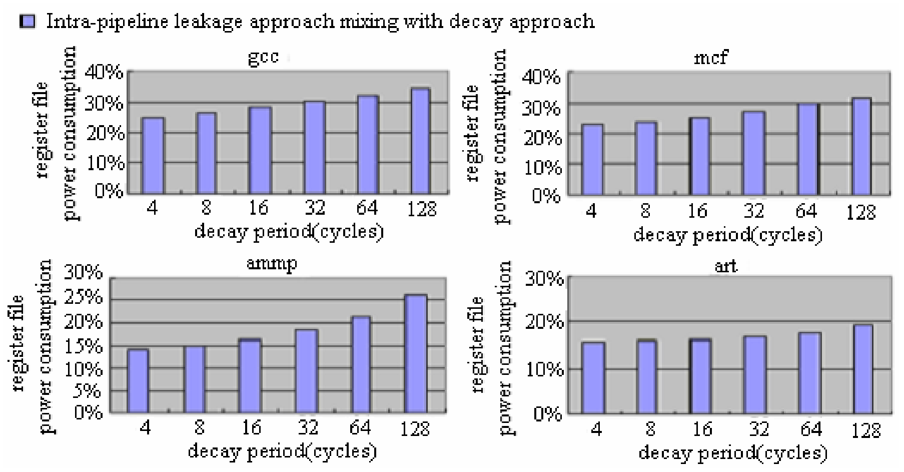

Fig. 2. Register file power consumption with different decay periods

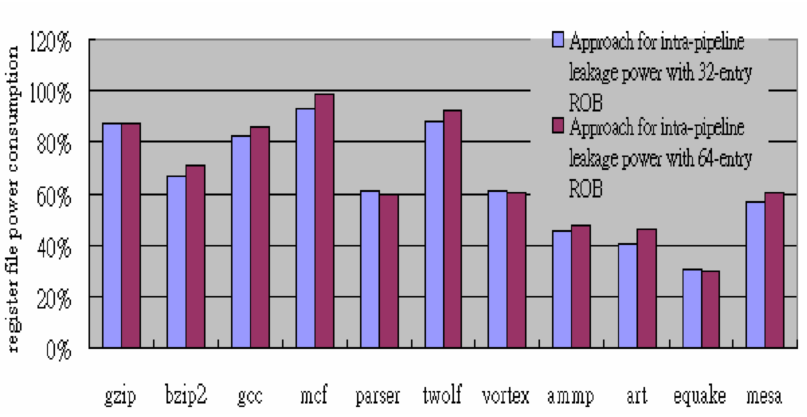

Fig. 3. Register file power consumption under different ROB sizes

instruction execution sequence, it require more hardware cost and thus results in, averagely, more leakage power consumption.

5.2.2 Performance Penalty

Figure 4 shows the performance penalty of the decay approach for the 32-entry ROB and the 64-entry ROB. For all benchmark, the performance penalty decreases proportionally with the increases of decay period lengths; when the period is 128 cycles long, the performance penalty is only 1% or almost negligible. This result is also consistent with the discussion in Section 4.2, in which a large decay period filter non-seldom-used registers not entering drowsy mode. So we have less performance penalty when the period becomes large.

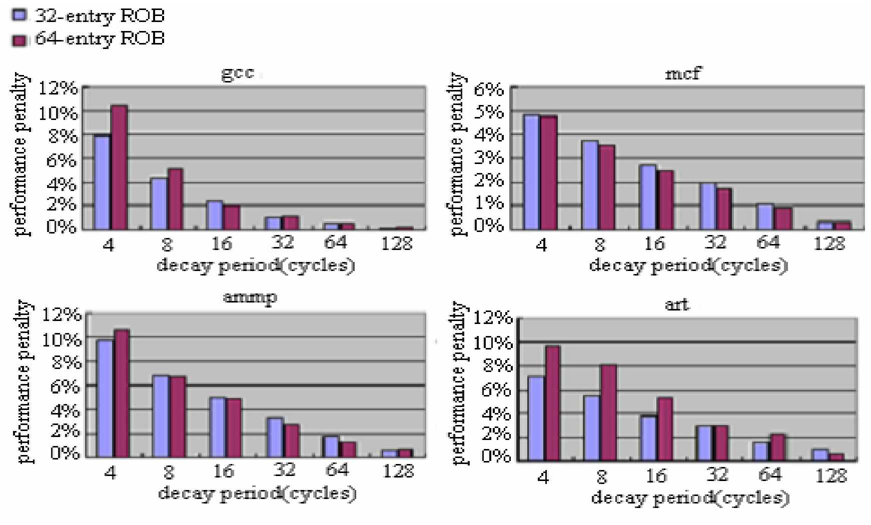

Fig. 4. Performance penalty with different decay period under different ROB sizes

6 Conclusion

In this paper we proposed a pure hardware-monitor approach to save the intra-pipeline leakage power, and applied a decay approach to save the seldom-used register leakage power. For saving the intra-pipeline leakage power, we power off the registers, in which the values will not be used again, to the destroy mode (0 Volts), and power down the registers, in which the value may be accessed again after branch misprediction, to the drowsy mode (0.3 Volts). For saving the seldom-used-register leakage power, we power down the registers, which has not been accessed since a period of time. Simulation results show that our approach can save significant power consumption for the register file with negligible performance penalty, especially for long-depth pipelines or floating-point operations. This means that through our approach, more performance-oriented features can be applied to a high-end embedded processor.

References

1. Furber, S.B., Garside, J.D., Gilbert, D.A.: AMULET-3: a high-performance self-timed ARM microprocessor. Proc. International Conference on Computer Design: VLSI in Computers and Processors (1998) 247-252
2. Intel XscaleTM Technology, http://www.intel.com/design/intelxscale/
3. W.Zhang, et al.: Reducing Instruction Cache Energy Consumption Using a Compiler-based Strategy. ACM Transactions on Architecture and Code Optimization, Vol 1, No. 1 (2004) 3-33
4. D. Ponomarev, et al.: Energy-Efficient Design of the Reorder Buffer. Proc. the International Workshop on Power and Timing Modeling, Optimization and Simulation (2002) 289-299

5. G. Kucuk, et al.: Complexity-Effective Reorder Buffer Designs for Superscalar Processor. IEEE Transactions on Computers, Vol 53, Issue 6 (2004) 653 – 665
6. Jose L. Ayala.: Power-Aware Compilation for Register File Energy Reduction. International Journal of Parallel Progreamming, Vol, 31, No. 6 (2003) 451-467
7. Wann-Yun Shieh, Chien-Chen Chen.: Exploiting Register-usage for Saving Register-file Energy in Embedded Processors. Lecture Notes in Computer Science, Vol 3824 (2005) 37-46
8. Monreal, T., Vinals, V., Gonzalez, J., Gonzalez, A., Valero, M.: Late Allocation and Early Release of Physical Registers. IEEE Transactions on Computers, Vol 53, Issue 10 (2004) 1244 – 1259
9. Dmitry Ponomarev, Gurhan Kucuk, Oguz Ergin, Kanad Ghose.: Isolating Short-Lived Operands for Energy Reduction. Computers, IEEE Transactions on Vol 53, Issue 6 (2004) 697-709
10. Intel Corp.: The Intel Architecture Software Developers Manual. (1999)
11. Kaxiras, S., Zhigang Hu., Martonosi, M.: Cache decay: exploiting generational behavior to reduce cache leakage power. Proc. 28th Annual Interna-tional Symposium on Computer Architecture (2001) 240-151
12. Doug Burger, Todd M. Austin.: The SimpleScalar Tool Set, Version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison (1997)
13. D. Brooks, et al.: Wattch : A Framework for Architectural-Level Power Analysis and Optimiza-tions. Proc. The 27th International Symposium on Computer Architecture (2000) 83-94