Uncriticality-directed Low-power Instruction Scheduling

Watanabe, Shingo
Kyushu Institute of Technology

Sato, Toshinori
System LSI Research Center, Kyushu University

出版情報：IEEE Computer Society Annual Symposium on VLSI. 2008, pp.69-74, 2008-04-07
バージョン：
権利関係：

Uncriticality-directed Low-power Instruction Scheduling

Shingo Watanabe
Kyushu Institute of Technology
s-watanabe@klab.ai.kyutech.ac.jp

Toshinori Sato
Kyushu University
toshinori.sato@computer.org

Abstract

Intelligent mobile information devices require low-power and high-performance processors. In order to reduce energy consumption with maintaining computing performance, we proposed to utilize information regarding instruction criticality. Every functional unit in our processor has different latency and energy consumption. Only critical instructions are executed in power-hungry units, and the total energy consumption can be reduced. While we have studied several techniques to identify critical instructions for years, we have not found any technique that achieves both simplicity and high accuracy to identify. In this paper, we propose to exploit uncriticality rather than criticality. Only uncritical instructions are executed in power-efficient units, and energy consumption can be reduced. Our simulation results show approximately 10% energy reduction.

1. Introduction

Smart mobile devices and embedded systems require high computing capability, thus employing high performance microprocessors. In addition, however, they require low power consumption as well as high performance. While there is a trade-off between power and performance, power is the primary design constraint in embedded applications. The dynamic power P_{active} and gate delay t_{pd} of a CMOS circuit are given by

$$P_{active} \propto f \cdot C_{load} \cdot V_{dd}^2 \quad (1)$$

$$t_{pd} \propto \frac{V_{dd}}{(V_{dd} - V_{th})^a} \quad (2)$$

where f is clock frequency, C_{load} is load capacitance, V_{dd} is supply voltage, and V_{th} is the threshold voltage of the device. a is a factor depending upon the carrier velocity saturation and is about 1.3 in advanced MOSFETs [11, 12]. Eq.(1) clearly shows that power supply reduction is the most effective way to lower power consumption. On the other hand, Eq.(2) tells us that supply voltage reduction increases gate delay, that means a slower clock frequency and hence lower processor performance. Therefore, it is required that the threshold voltage is proportionally scaled down with the supply voltage in order to maintain high speed in transistor switching.

On the other hand, leakage power can be given by

$$P_{off} = I_{off} \cdot V_{dd} \quad (3)$$

where I_{off} is the leakage current. The subthreshold leakage current I_{off} is dominated by threshold voltage V_{th} in the following equation:

$$I_{off} \propto 10^{-\frac{V_{th}}{S}} \quad (4)$$

where S is the subthreshold swing parameter and is around 85mV/decade [17]. As you can easily find, lower threshold voltage leads to increase subthreshold leakage current and thus leakage power. Maintaining high speed in transistor switching via low threshold voltage gives rise to a significant amount of leakage power consumption. In order to solve the problems, we decided to exploit information regarding critical path in executing a program [14]. Actually, a microprocessor has dual-power functional units. Instructions that are on critical path, which determines the execution time of the program, are executed in fast and power-hungry units and instructions that are not on critical path are executed in slow and power-efficient units. It is expected that the scheduling reduces microprocessor power consumption with maintaining its performance.

This paper is organized as follows. Section 2 introduces the criticality-directed low-power scheduling, which we have studied for years. Related works are also summarized in the section. Section 3 proposes alternative scheduling directed by instruction uncriticality and describes a mechanism to identify uncritical instructions. Section 4 explains our evaluation methodology and Section 5 presents experimental results. Finally, Section 6 concludes.

2. Criticality-directed Scheduling

Even embedded processors currently execute instructions in an out-of-order fashion to attain high performance [10, 13]. The execution time is determined by the processor's computing capability and by dependences between instructions executed on the processor. The critical path is the longest path in a

data flow graph (DFG), where each node represents an instruction and each arc represents a dependency between instructions, and it limits performance on processors with instruction level parallelism [15]. Figure 1 shows a DFG. In this example, its critical path consists of instructions I0 -> I3 -> I4 -> I6 -> I7 if every instruction's latency is one cycle.

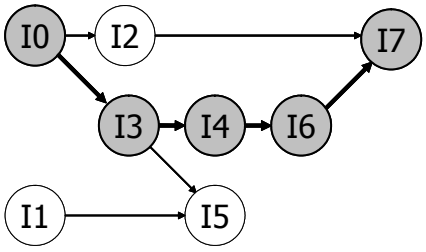

Figure 1. DFG and Critical Path.

We proposed to provide a microprocessor with slow and power-efficient functional units as well as fast and power-hungry ones [14]. Instructions on critical path, which determine the execution time of the program, are executed in the fast and power-hungry units and the otherwise are executed in the slow and power-efficient ones, as shown in Figure 2. Using this scheduling strategy, it is expected that microprocessor energy consumption is reduced while its performance is maintained. In order to benefit from the energy-aware scheduling, there are two key components, the power-efficient functional unit and some mechanisms to identify critical path.

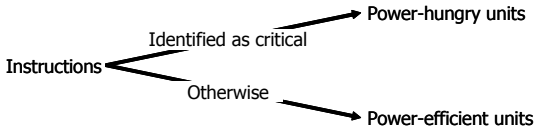

Figure 2. Criticality-directed Scheduling.

Utilizing slow and power-efficient functional units for executing probably uncritical instructions is the key idea behind the proposed low-power architecture. Power-efficient units are realized by the following circuit designs. Delivering lower supply voltage to the slow unit improves its energy efficiency. Increasing threshold voltage diminishes circuit speed performance but decreases power due to leakage current.

2.1. Previous Work

The critical path is a chain of dependent instructions, which determines the number of cycles executing the program. And thus, the performance of the processor is limited by the speed at which it executes the instructions along the critical path. If we can identify which instructions are critical, we can accelerate their execution by any means.

Critical path predictors (CPP's) [18] are a mechanism to dynamically identify critical instructions. Exploiting information regarding instruction criticality is effective not only for improving processor performance but also for reducing power consumption [3, 14, 16]. Seng et al. [16] proposed to use the CPP to reduce power consumption on high-performance microprocessors. The critical instructions are executed in fast and power-hungry functional units while the non-critical ones are executed in slow and power-efficient units. They focus on power consumption rather than energy consumption, which is a primary concern in portable embedded devices. Chin et al. [3] utilize slack [5] for dynamic voltage scaling (DVF). The slack of a dynamic instruction is the number of cycles it can be delayed with no effect on the execution.

While CPP's can be utilized for identifying critical path, none of them achieves both simplicity in circuit and accuracy in identification [4]. Complex circuit consumes additional power. Low accuracy seriously diminishes processor performance, resulting in the increase in energy consumption.

3. Uncriticality-directed Scheduling

3.1. Overview

While we have studied several techniques to identify critical instructions for years, we have not yet found any technique that achieves both simplicity in circuit and high accuracy in identification [4]. In order to reduce the performance loss due to low identification accuracy, we propose to exploit instruction uncriticality rather than instruction criticality. Only uncritical instructions are executed in the slow units, as shown in Figure 3.

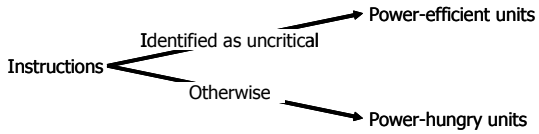

Figure 3. Unriticality-directed Scheduling.

While we have several CPP's to identify critical instructions, there are not any mechanisms to identify uncritical instructions. Since the mechanism for identifying critical instructions does not achieve both simplicity and accuracy, it seems difficult to construct such a mechanism for identifying uncritical instructions that achieves the both. However, it is not correct. We can easily identify uncritical instructions.

Out-of-order execution processors have the instruction scheduling window, where instructions wait for their input operands. Each instruction can be issued to a functional unit where it is executed, only when its operands become available. Here, we call such an instruction *ready* instruction. In the instruction window, there are two types of ready instructions. One is instructions that have their dependent instructions in the instruction window. The other is instructions that do not have any dependent instructions. Here, we call the latter instructions *solitary* instructions. It is not necessary to execute solitary instructions in hurry, since their execution results will not be immediately used. They can be executed in the slow units. In other words, solitary instructions are uncritical.

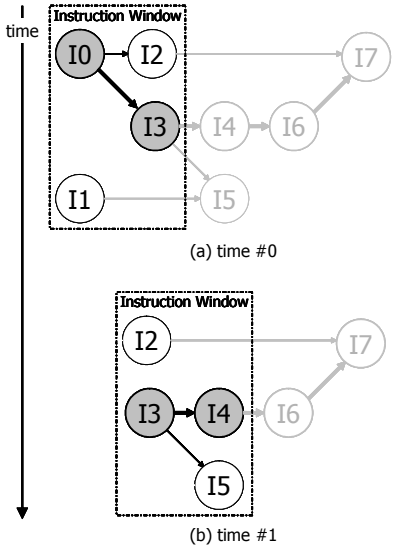

Figure 4. Scheduling Example.

Figure 4 explains how uncriticality-directed instruction scheduling works. We use the DFG shown in Figure 1. At time #0, it is supposed that four instructions, I0 - I3, are in the instruction window. The dashed box indicates the instruction window and gray nodes indicate the future instructions, which have not been dispatched yet. You can see that instructions I0 and I1 are ready instructions. I0 has two dependent instructions, I2 and I3, while I1 does not have any. That is, I1 is a solitary instruction and is issued to the slow and power-efficient unit. In contrast, I0 is issued to the fast and power-hungry unit. At time #1, I0 and I1 have already left the window, and I4 and I5 are dispatched into the window. Now, I2 and I3 are ready instructions and only I2 is a solitary instruction. Hence, I3 is issued to the fast unit and I2 is issued to the slow one. From this example, you can understand how uncriticality-directed scheduling works.

3.2. Identifying Solitary Instructions

Next, we propose a mechanism to identify solitary instructions. Before describing its details, we explain register renaming mechanism.

In order to eliminate anti- and output- dependences, out-of-order execution processors perform register renaming before they dispatch instructions into the instruction window. There are two common ways to implement register renaming. One is using a separated renaming registers which are usually constructed by reorder buffer. The other combines the renaming registers with architected registers in a single register file [19]. We focus on the latter one. The register renaming mechanism requires a register mapping hardware, which mainly consists of three structures; map table, active list, and free list. By means of the map table, every logical register is mapped into a physical register. The destination register is mapped to a free physical register which is supplied by the free list, while operand registers are translated into the last mapping assigned to them. The old destination register is kept in the active list. When an instruction is retired, the old destination register that is allocated by the previous instruction with the same logical register is freed and placed in the free list. We utilize the map table in order to identify solitary instructions.

Figure 5 shows the mechanism to identify solitary instructions. A small table is attached to the map table. We call the table *solitary* table. The solitary table is 1-bit wide and its entry size is equal to the number of physical registers. Since conventional processors have only tens of registers, its hardware budget is very small. In a different view, every register file entry has an additional 1-bit field. The solitary table works as follows. (1) When a new physical register is allocated as a destination, its associated entry in the solitary table is set. (2) When every instruction refers the map table by its logical source register number (Ln in the figure) and obtains the corresponding physical register number (Pm in the figure), its associated entry in the solitary table is reset. (3) Whenever an instruction is issued, it refers the solitary table by its physical destination register number. If its associated entry is still set, it is a solitary instruction.

This mechanism is 100% accurate in identifying solitary instructions, because all instructions in the instruction window have updated the solitary table when they are dispatched into the window. From these observations, we can see that the solitary table achieves both simplicity in circuit and accuracy in identification, when it is utilized for uncriticality-directed instruction scheduling.

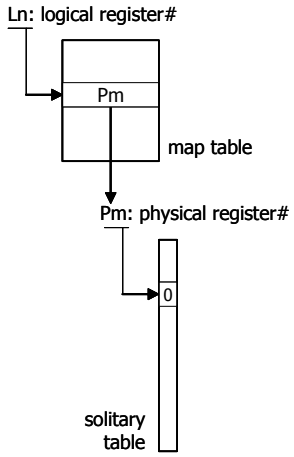

Figure 5. Solitary Table.

4. Evaluation Methodology

We implemented our simulator using SimpleScalar/ PISA tool set [1]. Table 1 summarizes processor configurations. Six programs from SPEC2000 CINT and eight programs from MediaBench [8] are used. For SPEC programs, 200 million instructions are skipped before actual simulation begins. After that, each program is executed for 100 million instructions. For MediaBench, each program is executed from beginning to end. We do not count NOP instructions.

Table 1. Processor Configurations

Fetch width	8 instructions
L1 instruction cache	16K, 2 way, 1 cycle
Branch predictor	gshare + bimodal
gshare predictor	4K entries, 12 histories
bimodal predictor	4K entries
Branch target buffer	1K sets, 4 way
Dispatch width	4 instructions
Instruction window size	32 entries
Issue width	4 instructions
Integer ALU's	4 fast units (+ 1 slow)
Integer multipliers	2 units
Floating-point ALU's	1 unit
Floating-point	1 unit
L1 data cache ports	2 ports
L1 data cache	16K, 4 way, 2 cycles
Unified L2 cache	8M, 8 way, 10 cycles
Memory	Infinite, 100 cycles
Commit width	8 instructions

We will compare two processor models; the baseline model and the proposed one. We focus on integer ALU (iALU). The baseline model has four fast and power-hungry iALU's and the proposed one has an additional slow iALU. The fast iALU executes most integer operations in one cycle, while the slow iALU executes operations in two cycles. In this evaluation, we count the activities of each iALU and estimate its dynamic power consumption based on Pentium M's frequency-voltage configurations [6] for the fast (1.6GHz - 1.484V) and slow (800MHz - 1.036V) iALU's. While it was predicted that leakage power would be comparable to dynamic power in future process technologies [2], we assume that leakage power is equal to 10% of dynamic power since Intel's 45nm process technology will allow chips with more than five times less leakage power than those made today [7]. We apply this assumption to the fast iALU's but do not consider leakage power for the slow iALU, since supply voltage is reduced and threshold voltage is increased in the slow iALU.

5. Results

This section presents simulation results. First, the impact on computing performance is shown. After that, we show how frequently slow iALU is utilized. And last, the impact on energy efficiency is presented.

5.1. Impact on Performance

Figure 6 shows the percentage increase in execution cycles. For SPEC programs, the impact on performance is little; the percentage increase is up to 4.1% and an average of 2.3%. Some programs from MediaBench suffer slightly larger performance degradation than SPEC programs do. For two programs (unepic and mpeg2encode), performance degradation is more than 5%. However, the average increase in execution cycles among all MediaBench programs is only 3.3%. From these observations, we confirm that solitary instructions do not have serious impact on performance even if their execution latency is increased. Considering all programs evaluated, the average increase in execution cycles is only 2.9%.

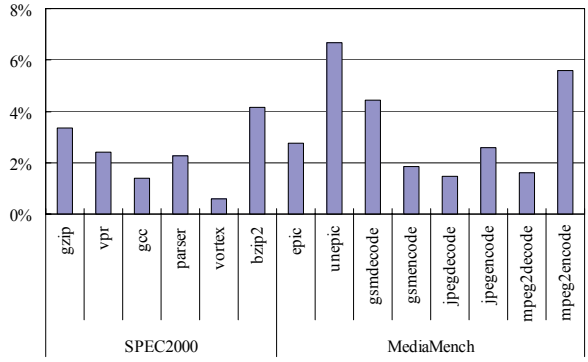

Figure 6. Increase in Execution Cycles.

5.2. Solitude Breakdown

Figure 7 presents the breakdown of integer ALU instructions. It explains how frequently iALU instructions are executed in the slow iALU. In other words, it explains how many instructions are solitary. Since we have already seen that performance impact is little, the many the solitary instructions are, the larger the power reduction is. Each bar is divided into two parts. The bottom part indicates the percentage of instructions that are executed in the fast iALU's. The top one indicates the percentage of instructions that are executed in the slow ALU.

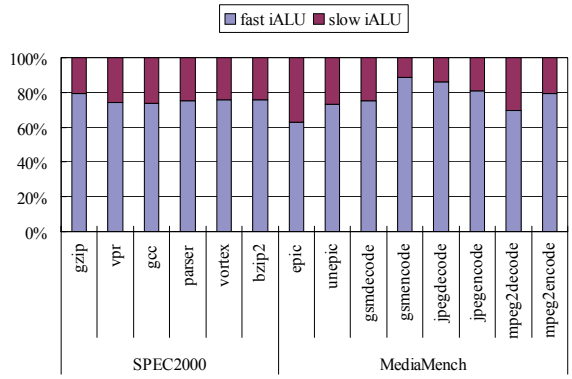

Figure 7. Breakdown of iALU Instructions.

For SPEC programs, the percentage of solitary instructions is almost the same among the programs and is 24.2% on average. One forth of iALU instructions are executed in the slow iALU. This is a good number, since the issue width of the model processor is four and there is one slow iALU. In the case of MediaBench, every program has a different characteristic. The percentage of solitary instructions varies between 11.4% and 37.1%. Programs that have a large number of solitary instructions will loose the chance of power reduction, since the model processor does not have enough number of the slow iALU's for such a large percentage. In contrast, programs that have only a small number of solitary instructions will achieve small power reduction. On average, 22.9% of instructions are executed in the slow iALU. The average number is smaller than that of SPEC programs. When we consider all programs evaluated, 23.5% of instructions are executed in the slow iALU. Hence, the ratio of iALU's, that is four fast iALU's and one slow iALU, is a good trade-off point.

5.3. Impact on Energy Efficiency

Figure 8 shows energy consumed by integer ALU's. For each program, the number is normalized by that for the baseline model. Energy reduction can be easily estimated from Figure 7. Actually, the shape of the graphs in Figures 7 and 8 are very similar. The difference comes from leakage power. In the programs where severe performance loss occurs, leakage power is increased due to longer execution time. For SPEC programs, the average energy reduction is 9.3%. In the case of MediaBench, energy reduction varies between 2.9% and 16.1%. The average power reduction among all programs is 9.0%. While this number looks small, it is achieved by a very small additional hardware budget; the small solitary table and an integer ALU.

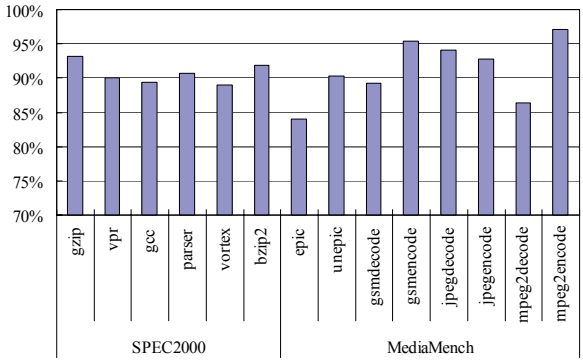

Figure 8. Relative Energy Consumption.

Figure 9 show the relative energy-delay product (EDP). Smaller EDP values are better in energy efficiency. While the impact on performance is little, one program (mpeg2encode) diminishes energy efficiency by 2.5%. Referring back to Figure 6, it can be seen that mpeg2encode is one of the programs that has larger impact on performance. Nonetheless, an average of 6.3% reduction in EDP is achieved. Especially, in the case of epic, 13.7% improvement in energy efficiency can be seen.

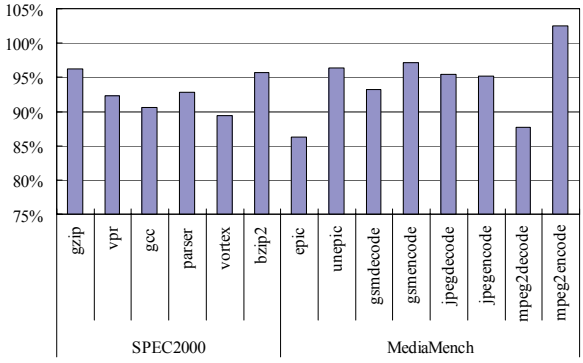

Figure 9. Relative Energy-Delay Product.

Figure 10 presents the relative energy-delay-square product (ED^2P). ED^2P is another good metric for evaluating energy efficiency under DVS [9]. While we do not utilize DVS, we use multiple supply voltages

for iALU's and hence we evaluate energy efficiency by ED^2P. Smaller ED^2P values are better in energy efficiency. Since the impact on performance is further considered in ED^2P, two programs (unepic and mpeg2encode) show degradation in energy efficiency. Referring back to Figure 6, it can be seen that unepic and mpeg2encode are the programs that we are afraid of larger performance degradation. While ED^2P is a metric for high-performance processors and it might not be good for evaluating embedded processors, we still achieve an average improvement of 3.6% in ED^2P.

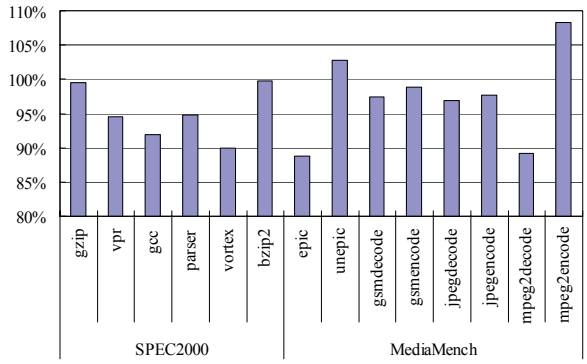

Figure 10. Relative Energy-Delay^2 Product.

6. Conclusions

Modern smart mobile and embedded systems require low-power and high-performance processors. In order to reduce energy consumption with maintaining computing performance, we are studying to utilize information regarding critical path in the program. However, unfortunately, we have not found any technique that achieves both simplicity and high accuracy to identify. This paper proposed not to utilize information regarding instruction criticality but to utilize information regarding instruction uncriticality. Uncritical instructions are executed in power-efficient units and critical ones are executed in power-hungry units, and hence the total energy consumption can be reduced. We proposed a very simple mechanism to identify uncritical instructions. It only requires a 1-bit table where the number of entries is equal to that of physical registers. Our simulation results showed that approximately 10% reduction in energy consumption is achieved. The mechanism proposed in this paper is very simple so that we expect that its one of the promising power reduction techniques for embedded systems.

Acknowledgements

This work is partially supported by the CREST program of Japan Science and Technology Agency.

References

[1] T. Austin et al.: SimpleScalar: an infrastructure for computer system modeling, IEEE Computer, 35(2), 2002.

[2] S. Borkar: Microarchitecture and design challenges for gigascale integration, 37th Int. Symp. on Microarchitecture, Keynote, 2004.

[3] Y. Chin et al.: Evaluating techniques for exploiting instruction slack, Int. Conf. on Computer Design, 2004.

[4] A. Chiyonobu et al.: Evaluating the critical path predictors using critical path detection criteria, IPSJ SIG Technical Reports, 2006-ARC-169-11, 2006.

[5] B. Fields et al.: Slack: maximizing performance under technological constraints", 29th Int. Symp. on Computer Architecture, 2002.

[6] Intel Corporation: Intel Pentium M Processor, Datasheet, 2004.

[7] Intel Corporation: Introducing the 45nm next-generation Intel Core microarchitecture, white paper, 2007.

[8] C. Lee et al.: MediaBench: a tool for evaluating and synthesizing multimedia and communications systems, 30th Int. Symp. on Microarchitecture, 1997.

[9] M. Martonosi et al.: Modeling and analyzing CPU power and performance: metrics, methods, and abstractions, Int. Conf. on Measurement and Modeling of Computer Systems, 2001.

[10] V. Rajagopalan: New area and power-efficient MIPS processors achieve high performance, Microprocessor Forum, 2007.

[11] T. Sakurai: Alpha power-law MOS model, IEEE Solid-State Circuits Society Newsletter, 9(4), 2004.

[12] T. Sakurai et al.: Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas", IEEE JSSC, 25(2), 1990.

[13] T. Sartorius: The Scorpion mobile application microprocessor, Microprocessor Forum, 2007.

[14] T. Sato et al.: Power and performance fitting in nanometer design", 5th Int. Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems, 2002.

[15] M. Schlansker et al.: Critical path reduction for scalar programs, 28th Int. Symp. on Microarchitecture, 1995.

[16] J. S. Seng et al.: Reducing power with dynamic critical path information, 34th Int. Symp. on Microarchitecture, 2001.

[17] D. Sylvester et al.: Power-driven challenges in nanometer design", IEEE D&TC, 18(6), 2001.

[18] E. Tune et al.: Dynamic prediction of critical path instructions 7th Int. Symp. on High Performance Computer Architecture, 2001.

[19] K. C. Yeager: The MIPS R10000 superscalar microprocessor", IEEE Micro, 16(2), 1996.