Instruction Scheduling for Variation-originated Variable Latencies

Sato, Toshinori
System LSI Research Center, Kyushu University

Watanabe, Shingo
Kyushu Institute of Technology

Instruction Scheduling for Variation-originated Variable Latencies

Toshinori Sato
Kyushu University
toshinori.sato@computer.org

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

Abstract

The advance in semiconductor technologies presents the serious problem of parameter variations. They affect threshold voltage of transistors and thus circuit delay also has variations. Recently, variable latency adders and long latency adders are proposed to manage the variation problem. Unfortunately, replacing a variation-affected adder with the long latency one has severe impact on processor performance. In order to maintain performance, the present paper proposes an instruction scheduling technique considering instruction criticality. By issuing and executing only uncritical instructions in the long latency ALU, we can maintain processor performance. From detailed simulations, we find that the proposed scheduling technique improves processor performance by 12.5% on average over the conventional scheduling and that performance degradation from a variation-free processor is only 4.0% on average, when 2 of 4 ALU's are affected by variations.

1. Introduction

The advanced semiconductor technologies increase parameter variations [1]. Parameter variations affect threshold voltage of transistors and therefore circuit delay also has variations. This reduces parameter yield and thus is serious for profitability of semiconductor companies. Fortunately, all circuits on a single chip are not affected by variations. Hence, by replacing the variation-affected circuits with variation-resilience ones, the variation-affected chips can be shipped. Examples of the variation resilient circuits based on such the strategy are variable latency and long latency adders [2, 3]. They are simple and clever techniques to manage parameter variations. Unfortunately, our evaluations show severe performance degradation. Considering the problem, we propose an instruction scheduling technique that is aware of variable latencies. There are a lot of related works regarding instruction criticality [4, 5, 6]. However, none of them achieves both high accuracy for identifying critical instructions and simple and small hardware [7]. This paper proposes the uncriticality-directed scheduling and the solitary table in order to achieve the two requirements.

The rest of the present paper is organized as follows. Section2 summarizes the criticality-directed instruction scheduling technique, which is previously proposed. Section 3 introduces an instruction scheduling technique directed by instruction uncriticality. Section 4 introduces evaluation methodology. Section 5 presents simulation results. Section 6 concludes the paper.

2. Criticality-directed Scheduling

The execution time is determined by the processor's computing capability and by dependences between instructions

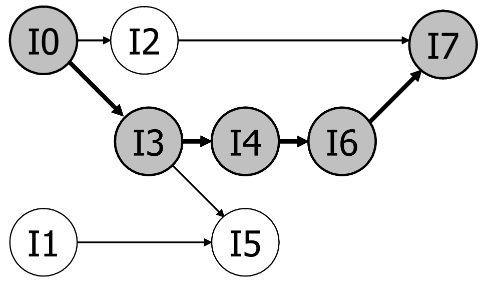

Figure 1. DFG and Critical Path

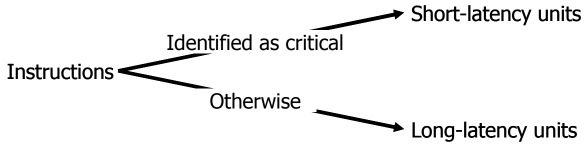

Figure 2. Criticality-directed Scheduling

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 of processors with instruction level parallelism. Figure 1 shows an example of the DFG, where its critical path consists of instructions I0 -> I3 -> I4 -> I6 -> I7 if every instruction's latency is one cycle.

We proposed that a microprocessor has fast functional units with short latency and slow ones with long latency [6]. Instructions on critical path, which determine the execution time of the program, are executed in the fast units and the otherwise are executed in the slow ones, as shown in Figure 2. Using this scheduling strategy, it is expected that microprocessor performance is maintained. Since even embedded processors currently execute instructions in an out-of-order fashion to attain high performance [8, 9], they can enjoy the strategy. In order to benefit from the criticality-directed instruction scheduling, some mechanisms to identify critical path is required.

2.1. Previous work

The critical path is a chain of dependent instructions, which determines the number of cycles executing the program. Processor performance 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 predictor (CPP) [4, 5, 6] is a technique for identifying critical instructions dynamically. Exploiting information regarding instruction criticality is effective for improving processor performance. While CPP's can be utilized for identifying critical path, none of them achieves both

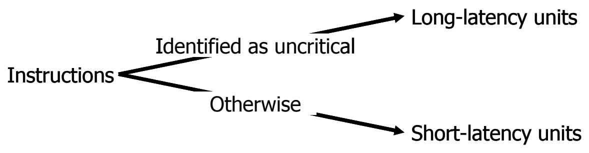

Figure 3. Uncriticality-directed Scheduling

simplicity in circuit and accuracy in identification [7]. Complex circuit consumes additional power and low accuracy seriously diminishes processor performance.

3. Uncriticality-directed Scheduling

3.1. Architecture 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 [7]. In order to reduce the performance loss due to poor accuracy in identification, we propose to exploit instruction uncriticality rather than instruction criticality. Only uncritical instructions are executed in the slow units, as shown in Figure 3.

While we have 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 [7], 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. Every instruction can be issued to a functional unit where it is executed, only when its operands become available. Here, we call such instructions *ready* instructions. 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 there. Here, we call the latter ones *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 on the slow units. In other words, solitary instructions are uncritical.

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 unit. In contrast, I0 is issued to the fast 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-based scheduling works.

3.2. Identifying solitary instructions

Next, we propose a mechanism to identify solitary

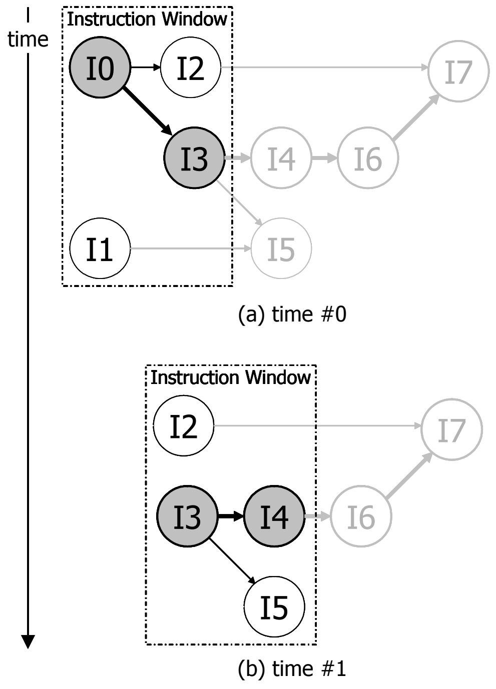

Figure 4. Example of Uncriticality-directed Scheduling

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. 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 (L_n in the figure) and obtains the corresponding physical register number (P_m in the figure), its associated entry in the

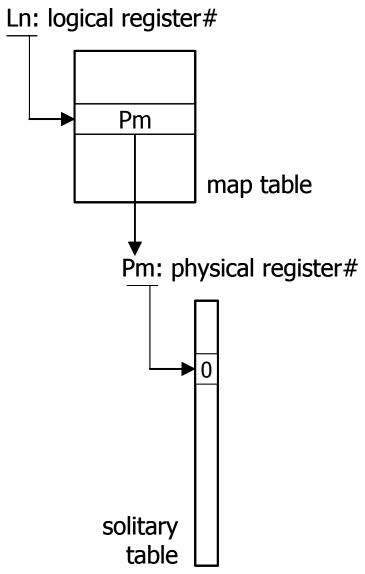

Figure 5. Solitary Table

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-based instruction scheduling.

3.3. Uncriticality-directed Scheduling for Variation-aware Units

In order to maintain processor performance under process variations, we propose to combine the variable latency or long latency ALU with the uncriticality-directed instruction scheduling. Only uncritical instructions are issued into and executed on the variable latency or long latency ALU's.

4. Evaluation Methodology

We implemented our simulator using SimpleScalar/PISA tool set [10]. Table 1 summarizes processor configurations. In the rest of this paper, functional unit means integer ALU (iALU).

Six programs from SPEC2000 CINT and six programs from MediaBench [11] 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.

We consider three cases, where 1, 2, and 3 iALU's are affected by variations, respectively, and thus they are replaced with 2-latency iALU's. We compare the proposed scheduling with the conventional one.

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 ALUs	4 units
Integer multipliers	2 units
Floating ALU	1 unit
Floating multiplier	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

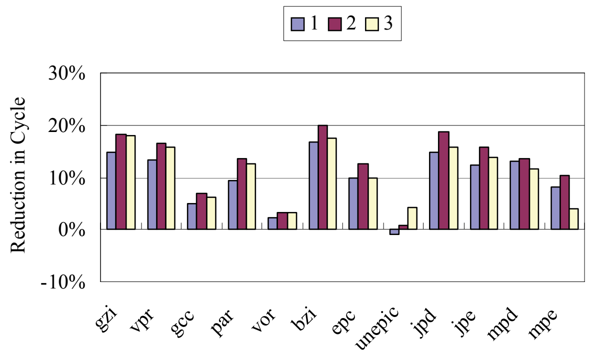

Figure 6. Performance Improvement

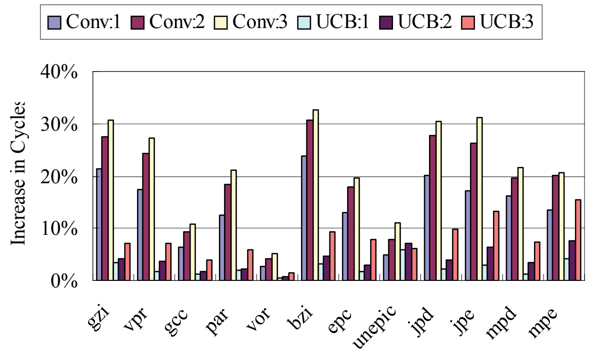

Figure 7. Increase in Execution Cycles

5. Results

Figure 6 presents how the uncriticality-directed scheduling improves processor performance over the conventional one. We use the percentage reduction in execution cycles as a metric. For each group of three bars, the left one indicates the percentage reduction for the case where one iALU is affected by variations and is replaced by a 2-latency iALU. The middle and the right bars are for the cases where 2 and 3 iALU's are replaced by the

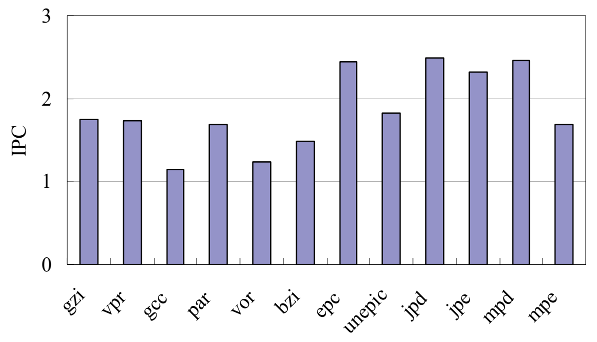

Figure 8. Baseline Performance

2-latency iALU's, respectively. Only in the case of epic with one 2-latenct iALU, the uncriticality-directed scheduling degrades performance. It reduce the execution cycles by 9.9%, 12.5%, and 11.1% for one, two, and three 2-latency iALU situations, respectively, on average. An interesting observation is that the effectiveness is largest for two 2-latency iALU case.

Figure 7 explains how the uncriticality-directed instruction scheduling maintains processor performance under parameter variations. We use the percentage increase in execution cycles as a metric. For each group of six bars, the left three bars (Conv:X) are for the conventional scheduling, and the right three bars (UCB:X) are for the uncriticality-directed scheduling. For each group of three bars, the left one indicates the percentage reduction for the case where one iALU is affected by variations and is replaced by the 2-latency iALU. The middle and the right bars are for the cases where 2 and 3 iALU's are replaced by the 2-latency iALU's, respectively.

As you can see, the conventional scheduling degrades processor performance seriously, especially for the case where the impact of variations is large. When 3 iALU's have to be replaced with the 2-latency iALU's, processor performance is decreased by as much as 32.6% with an average of 21.8%. In contrast, the uncriticality-directed scheduling efficiently maintains performance. It reduces performance only by 7.9% on average. Especially in the case of the replacement with one iALU, performance degradation is negligible and is 2.5% on average.

From these observations, we found that the uncriticality-directed instruction scheduling effectively exploit variable latency iALU's and thus has variation resilience.

Figure 8 presents the baseline processor performance. We use instructions per cycle (IPC) as a metric. As you can see, the shape of the graphs follows the performance improvement shown in Figure 6. The noticeable exceptions are vortex and epic. This explains that the more the baseline performance is the larger the gain from the uncriticality-directed scheduling is.

6. Conclusions

The advanced semiconductor technologies increase parameter variations, which seriously affect circuit delay. Recent proposals of variable latency iALU and long latency one are one solution for managing variations, however, the impact on processor performance is severe. This paper proposed the uncriticality-directed instruction scheduling. Only uncritical instructions are issued to and executed on the long-latency iALU's. When 2 of 4 iALU's are affected by variations, it improves processor performance by 12.5% on average over the conventional scheduling, and performance degradation from a variation-free processor is only 4.0% on average. The uncriticality-directed instruction scheduling effectively exploit variable latency iALU's and thus has variation resilience.

Acknowledgements

This work is partially supported by Grant-in-Aid for Scientific Research (KAKENHI) (A) # 19200004 from Japan Society for the Promotion of Science, and by the CREST program of Japan Science and Technology Agency.

References

[1] S. Borker, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation," IEEE Micro, Vol. 25, No. 6, 2005.

[2] Y. Chen, H. Li, J. Li, and C.-K. Koh, "Variable-latency Adder (VL-adder): New Arithmetic Circuit Design Practice to Overcome NBTI," International Symposium on Low Power Electronics and Design, 2007.

[3] D. Mohapatra, G. Karakonstantis, and K. Roy, "Low-power Process-variation Tolerant Arithmetic Units Using Input-based Elastic Clocking," International Symposium on Low Power Electronics and Design, 2007.

[4] E. Tune, D. Liang, D. M. Tullsen, and B. Calder, "Dynamic Prediction of Critical Path Instructions," 7th International Symposium on High Performance Computer Architecture, 2001.

[5] B. Fields, S. Rubin, and R. Bodik, "Focusing Processor Policies via Critical-Path Prediction," 28th International Symposium on Computer Architecture, 2001.

[6] A. Chiyonobu, T. Sato, and I. Arita, "Correlation-based Critical Path Predictors for Low Power Microprocessors", 6th International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems, 2003.

[7] A. Chiyonobu and T. Sato, "Evaluating the Critical Path Predictors Using Critical Path Detection Criteria", IPSJ SIG Technical Report, 2006-ARC-169, Vol. 2006, No. 88, 2006 (in Japanese).

[8] V. Rajagopalan, "New Area and Power - Efficient MIPS Processors Achieve High Performance", Microprocessor Forum, 2007.

[9] T. Sartorius, "The Scorpion Mobile Application Microprocessor", Microprocessor Forum, 2007.

[10] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: an Infrastructure for Computer System Modeling", IEEE Computer, Vol. 35, No. 2, 2002.

[11] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a Tool for Evaluating and Synthesizing Multimedia and Communications Systems", 30th International Symposium on Microarchitecture, 1997.