A Preliminary Evaluation of Timing-Speculative Instruction Collapsing

Toshinori Sato
Kyushu University
toshinori.sato@computer.org

Akihiro Chiyonobu
Kyushu Institute of Technology
chiyo@mickey.ai.kyutech.ac.jp

Abstract

The deep submicron semiconductor technologies will make the worst-case design impossible, since they can not provide design margins that it requires. We are investigating a typical-case design methodology, which we call the Constructive Timing Violation (CTV). This paper extends the CTV concept to collapse dependent instructions, resulting in performance improvement and power reduction. Based on detailed simulations, we find the proposed mechanism effectively collapses dependent instructions.

1. Introduction

In the deep submicron (DSM) semiconductor technologies, a conservative approach called "worst-case design" will not work very soon. The DSM increases noise and process variations and requires supply voltage reduction, and thus reduces design margins that the worst-case design methodologies require. We have to design microprocessors by considering typical case rather than worst case.

The Constructive Timing Violation (CTV) paradigm [15] is such a design methodology, where designers are focusing on typical cases rather than worrying about very rare worst cases. The CTV exploits an observation that the longest path for an individual operation of every logic circuit is generally much shorter than its critical path. The CTV also utilizes the fact that input signals activating the critical path are limited to a few variations. In other words, timing violations rarely occur even if the timing constraints on the critical path are not satisfied. For example, it has been reported that nearly 80% of paths have delays of half the critical time [21]. The CTV relies on circuit-level speculation, and thus sometimes timing violations occur, resulting in logic errors. Some fault tolerance mechanisms are provided for timing violations [10, 15, 19, 23]. When a violation is detected, a recovery mechanism used in modern microprocessors for speculative execution is utilized in order to reverse the processor state to a safe point. The philosophy behind the CTV can be applied for improving energy efficiency [15] as well as for boosting clock frequency [13], and we have evaluated it on a carry select adder [19].

It is expected that variations in dynamic circuit delay is larger in data path than in control logic in microprocessors. It is also expected that the difference between critical path delay and a typical delay in every circuit is larger in data path than in control logic. From the observations, we believe that the CTV is more beneficial for data path than control logic and propose an aggressive combination of the CTV with collapsing ALUs [22]. This makes it possible to execute multiple dependent operations in a single cycle. Its potential in performance gain is significant [16], and it will be translated into energy reduction when we prefer improvement in energy efficiency rather than in performance. In this paper, we propose a practical mechanism to detect a chain of instructions and to collapse them.

This paper is organized as follows. Section 2 reviews related work. Section 3 describes the mechanism which dynamically detects a chain of instructions and collapses them into a macro-instruction. Section 4 details evaluation methodology. Section 5 presents simulation results. Finally, Section 6 concludes.

2. Related Work

2.1. Typical-case design techniques

Kondo et al. [5] proposed Variable Latency Pipeline (VLP) structure for integer ALUs. Using properly two kinds of circuits according to the longest path of the circuits for each operation, the effective execution latency can be almost one cycle while its critical path is longer than one cycle. Our proposal is strongly influenced by the VLP. However, Kondo et al. does not mention the power issue, while we exploit the CTV to improve energy efficiency. In addition, our proposal is applicable not only to ALUs but also to any combinational logics.

Matsuo et al. [9] adopted the CTV for every pipeline stages in a microprocessor, which they call Dependable Pipelining. In Dependable Pipelining, clock frequency is adjusted for the CTV to be effective for performance improvement. When timing violations

frequently occur, clock frequency is decreased. Otherwise, it is increased. They evaluated how frequently timing violations occur on a carry look-ahead adder using Verilog-HDL and logic synthesis. We evaluated it on a carry select adder [19].

Razor [1, 3] permits to violate timing constraints to improve energy efficiency. Similarly with the CTV, Razor works at higher clock frequency that determines critical path. In order to detect timing violation, a Razor flip-flop (FF) is proposed. Each timing-critical FF (main FF) has its shadow FF, where a delayed clock is delivered to meet timing constrains. If the values latched in the main and shadow FFs do not match, a timing violation is detected. After that, the pipeline is recovered using a mechanism based on counterflow pipelining. One of the difficulties on Razor is how it is guaranteed that the shadow FF could always latch correct values. The delayed clock has to be carefully designed under the worst case constraints.

The CTV also shares the concept of the typical-case design with approximation circuits [7, 8], algorithmic noise tolerance (ANT) [18], and TEAtime [20].

2.2. Instruction collapsing techniques

Vassiliadis el al. [22] investigated to collapse dependent instructions dynamically. Consecutive two instructions are collapsed and executed in a single cycle using the interlock collapsing ALU (ICALU). Sazeidas et al. [17] have extended the idea of the ICALU. Non-consecutive and up-to-three instructions can be collapsed. Unfortunately, the assumed mechanism to detect collapsible instructions requires complex logic, and thus it might have serious impact on the cycle time of the instruction window.

Sassone et al. [12] proposed to dynamically detect instruction strands and execute them speculatively. A strand is a chain of dependent integer instructions without intermediate fan-out. Non-consecutive instructions can form a strand. We borrow the definition of the strand to realize our proposal. Values with only one consumer are called transient operands, and will be connected to form a strand. The operand table is utilized to detect transient operands. Unfortunately, however, any mechanisms for connecting transient operands to form a strand are not described in detail. In order to check if every transient operand connects to an existing strand, a content-addressable memory (CAM) might be utilized. The strand cache is also a CAM and stores strands. Every strand is handled as a macro-instruction. For example, it occupies only one instruction issue queue entry while it consists of multiple instructions. The closed-loop ALU is an integer ALU with a self-forwarding path, and can compute two operations in a single cycle since it does not require any long forwarding wires.

Dynamic Instruction Cascading (DIC) [11] is another technique to execute two dependent instructions in a single cycle. While renaming, every instruction is checked if it has a producer in the instruction issue queue. If it does, two instructions are cascaded and then simultaneously issued and executed in a single cycle. Non-consecutive instructions can be cascaded and the producer can have multiple consumers. In other words, there are not any restrictions in fan-out. There are not any descriptions on the mechanism to detect collapsible instructions, and thus we guess it resembles the one proposed for the ICALU.

3. Instruction Collapsing Boosted by CTV

3.1. Transient operands detection

A transient operand is a value that has only one consumer instruction, and a strand is a chain of dependent integer instructions that are joined by transient operands [12]. Each strand is speculatively executed in a single cycle, and hence dependent instructions are collapsed.

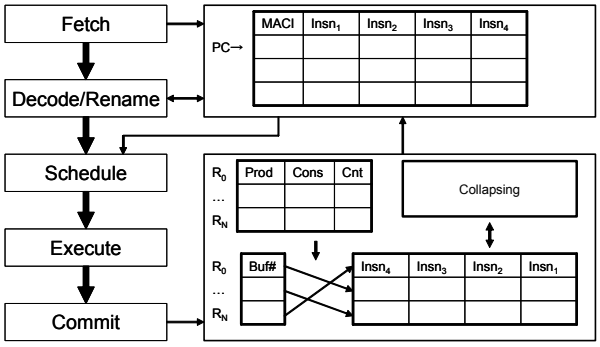

Figure 1. Strands detection, formation, and dispatch

In order to detect transient operands, we utilize the operand table [12] shown in the middle of Figure 1. The operand table has one entry for every architectural register, and each entry keeps the program counter (PC) of its producer instruction, the PC of last consumer instruction, and the number of consumers. The information is registered in the producer field (Prod in Figure 1), the consumer field (Cons in Figure 1), and the consumer counter (Cnt in Figure 1), respectively. When an instruction is committed, its PC is registered in its corresponding entry indexed by its destination register identifier. After that, when an instruction consumes the register value, its PC is registered in its corresponding entry indexed by its

source register identifier and the consumer counter in the entry is incremented. When the producer field is overwritten, the consumer counter is checked. If the value is one, a transient operand is detected.

3.2. Strands formation

When a transient operand is detected, it is checked if it connects to an existing strand. A straightforward implementation of this strand formation requires CAMs. Each entry of a CAM consumes large power by discharging when it does not find a match. In the strand formation, at most one entry finds a match, and hence large power is always consumed. To solve the problem, we propose a direct-mapped structure to connect a transient operand to an existing strand. The mechanism is shown in the bottom of Figure 1.

It consists of two tables. Each table has one entry for every architectural register. The former one is called the strand buffer, and the latter one is called the tail pointer. The strand buffer stores strands, which are under formation. In Figure 1, instructions in a strand are denoted as Insn's. The tail pointer is indexed by a register identifier of a transient operand and its content (Buf# in Figure 1) shows which strand in the strand buffer is connected with the transient operands via the register identifier. If a new transient operand is connected to an existing strand, the content in the tail pointer indexed by the register identifier of the transient operand is updated and it points the entry where the transient operand is stored. Therefore, it does not require any CAMs but requires direct-mapped tables and a simple switch box.

3.3. Strands dispatch

After a strand is formed, it is registered in the strand cache shown in the top of Figure 1. The strand cache also has a direct-mapped memory structure. The corresponding entry of the coming strand is determined by indexing it using the PC of its top instruction ($Insn_1$ in Figure 1). In order to identify the strand as a macro-instruction, it has a strand identifier (MACI in Figure 1).

At instruction fetch, the strand cache is referred in parallel with the instruction cache (or a trace cache). If an existing entry is found for the PC, the strand in the entry is provided to the following stages rather than the instruction from the instruction cache.

Every strand is handled as a single macro-instruction and is dispatched into a single entry in the instruction queue and in the reorder buffer[1]. When all source operands except transient operands produced by the strand are ready, it is issued into a collapsing ALU. As shown in Figure 2, the collapsing ALU consists of multiple conventional ALUs by cascaded each other. While it looks similar to the ALU pipeline used in Dataflow Mini-graph [2], it has the restrictions in operand distribution. One of the inputs for each ALU except the top one has to come from the preceding ALU. Bypassing any ALUs is prohibited. Only one operand from register files can be provided to each ALU in the collapsing ALU. The other operand has to be provided to the top one. Only value produced by the tail instruction in the strand is written into register files and all transient operands are removed[2].

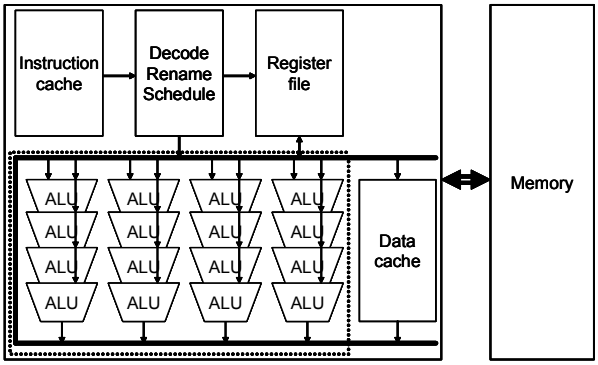

Figure 2. Instruction collapsing

3.4. Adoption of CTV

If a strand is executed in less than the sum of the worst delay in ALUs, a timing violation might occur in the collapsing ALU. To handle this, we adopt the CTV for the collapsing ALUs. We will not adopt the CTV for any other blocks in the processor. Since the typical delay is much smaller than the critical path delay in the ALU, it is expected that, in most cases, multiple instructions are executed in the collapsing ALU without a timing violation. The more instructions are packed into a strand, the less variance there should be from a typical circuit timing path. It is very unlikely that more than a few of the instructions in a strand will exercise their long paths in the same dynamic instance of the strand. As a result, where a single instruction exercising its longest path may cause a timing violation, with a strand that includes such an instruction the total path length may still fit within the allocated time.

Figure 3 explains why such a combination is good. In the middle of the figure, a specific execution delay

[1] This feature has not been implemented in a simulator as described in Section 4.

[2] This feature has not been implemented in the simulator, and all transient operands are written into register files as well.

in each ALU is shown, when a strand is issued in the collapsing ALU. The delays differ with each other, since the ALUs execute with different operands. If the sum of the delays is smaller than the allocated time, the strand can be executed without timing violation. In this paper, we assume that the allocated time is one clock cycle.

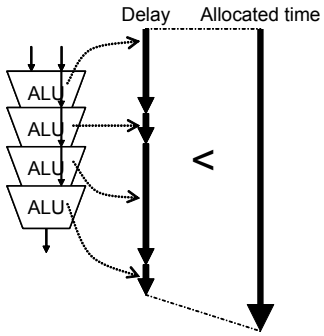

Figure 3. Combining Instruction collapsing with CTV

3.5. Power reduction

Speculatively collapsing instructions improves processor performance. If we prefer power reduction rather than performance improvement, we exploit Dynamic Voltage Scaling (DVS) to reduce power consumption. Clock frequency can be reduced to the point where processor performance matches to the original performance. The target of DVS is whole processor; both control logic and data paths. Since this is the conventional DVF, any timing violations in the control logic do not occur after the supply voltage is reduced. Similarly, the frequency of timing violation in the data path is maintained before and after supply voltage reduction.

3.6. Misspeculations detection and recovery

There are two types of misspeculations. One is due to timing violation. We assume every timing violation can be detected by comparing speculative value with its correct one. This is possible by using Razor FFs or the previously studied redundant techniques [10, 15]. Timing violation of strand execution is easily handled by presenting recovery mechanism for speculative execution. There are two solutions. Processors revert to a safe point using rollback mechanism. Or, misspeculated strands are reissued after decomposed into instructions. Deadlock situation does not occur, since processor state is recovered using correct values provided by the error detection mechanism.

The other type of misspeculation is due to misidentification of transient values. It could occur due to changes in control flow, and is detected by the decoder. If any one of transient values in a strand must be consumed by a following instruction, some recovery action is required since the transient value is lost. Any check-pointing mechanisms might not work for handling this misspeculation. This is the same issue with early physical register release, and thus some shadow storage might be inevitable. We have not found any good recovery mechanism yet. The recovery mechanism that can efficiently handle both types of misspeculations is an important future study.

4. Methodology

We evaluate our proposal on an execution-driven Alpha ISA simulator that models aggressive superscalar, out-of-order execution processor. Table 1 summarizes the configuration. The strand cache is direct-mapped and has 1024 entries.

Table 1. Processor configuration

Fetch width	4 instructions
L1 instruction cache	32KB, 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
Branch penalty	7 cycles
Dispatch width	4 instructions
Instruction issue queue	128 entries
Reorder buffer	128 entries
Load store queue	64 entries
Issue width	4 instructions/strands
Integer ALUs	4 units x collapsing depth
Integer multiplier	1 unit
Floating adders	4 units
Floating multiplier	1 unit
L1 data cache	32KB, 2 way, 1 cycle
L1 data cache ports	2 ports
Unified L2 cache	1MB, 4 way, 10 cycles
Memory	Infinite, 230 cycles
Commit width	4 instructions

Currently, our simulator has the following restrictions. First, multiple entries in the instruction issue queue and in the reorder buffer is occupied by every strand, since dynamic translation into macro-instructions have not been implemented yet. This limits the benefit from the improved efficiency of the instruction queue and reorder buffer. Second, the oracle scheduler is assumed and only correctly-speculated strands are dispatched and executed, since any recovery mechanisms for misspeculated dispatch of strands have not been working yet. Hence,

evaluations do not suffer from any misspeculation penalties and the results will be slightly optimistic. Third, due to the same reason, transient operands are written into register files. This limits the benefit from the improved efficiency of the commit bandwidth.

We use the following assumption to estimate timing violation. Since we do not perform detailed circuit simulations, some assumption on timing violation is necessary. In this evaluation, we assume that any timing violations does not occur when all operands requested by instructions in a strand is smaller than 16-bit. The operand includes transient operands as well as those provided by register files. This is a practical assumption as follows. As modern processors spend half of the execution cycle on ALU execution and half on full bypass [12] and as Intel Pentium 4 perform two 16-bit ALU operations in a single cycle [4], tightly-connected ALUs can compute three 16-bit operations in a single cycle. This is the worst case scenario, and it is expected that more instructions are executed in a single cycle if every operand requested by a strand is much smaller than 16-bit.

MediaBench [6] is used for this study. It is developed for use in the context of embedded, multimedia, and communications applications, and contains image processing, communications, and DSP applications. Table 2 lists the programs and their input sets. Each program is executed to completion.

Table 2. Benchmark programs

adpcm-decode	clinton.adpcm
adpcm-encode	clinton.pcm
epic-decode	test.image.pgm.E
epic-encode	test.image.pgm
g721-decode	clinton.g721
g721-encode	clinton.pcm
ghostscript	tiger.ps
gsm-decode	clinton.pcm.run.gsm
gsm-encode	clinton.pcm

5. Results

In this evaluation, the maximum number of instructions forming a strand is selected from 3, 4, 5, and 16. We call the maximum number "collapsing depth". Proportionally with the collapsing depth, the number of integer ALUs is increased, while the sum of instructions and strands, which are simultaneously issued, is up to four.

The average number of instructions included in each strand is shown in Figure 4. The horizontal line indicates benchmark program names, and the vertical line indicates the average number of instructions in a strand. For each group of four bars, the bars from left to right are for the cases where the collapsing depth is 3, 4, 5, and 16, respectively. Even when a strand can include 16 instructions, the average number of instructions in a strand is less than three in most programs. It is also found that the collapsing depth of 3 captures almost same number of instructions that the collapsing depth of 16 does. It is 2.5 instructions per strand. The increase in the collapsing depth requires a large strand cache and results in inefficient utilization of instruction slots. Hence, the collapsing depth of 3 is a good tradeoff point.

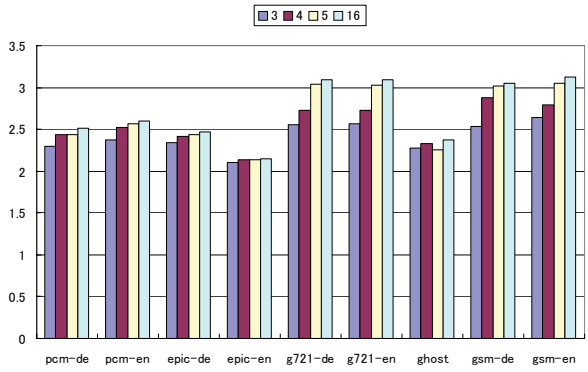

Figure 4. The number of instruction per strand

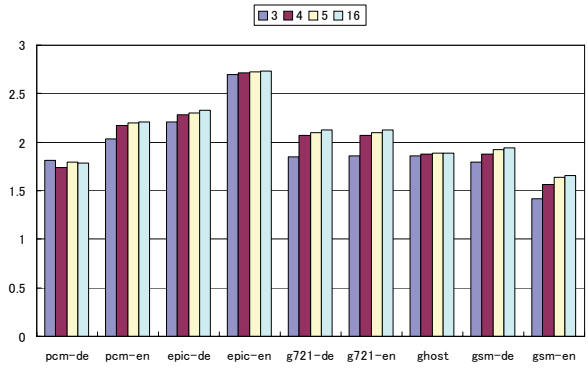

Figure 5. The number of operands per strand

The average number of operands requested by every strand is shown in Figure 5. If the number is high, the pressure on the register ports is increased. As can be seen, strands need less than three operands on average in all programs. If we limit the number of instructions per strand as 3, only two read ports are enough for most programs. This means 2.5 times more instructions are executed without the increase in register port requirement. This is very good news, since register files with large number of ports increase their access latency, resulting in slow clock frequency.

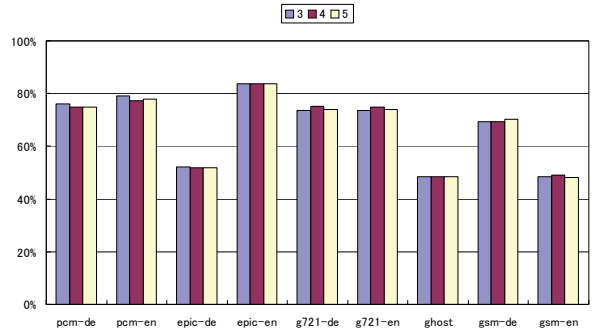

Figure 6. Percent of instructions with less-than 16b operands

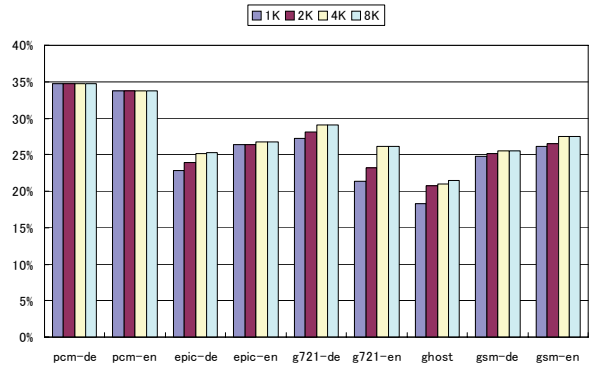

Figure 7. Strand cache hit rate

Also from the results, we see that the collapsing depth of 3 instructions per strand is a good tradeoff point.

If a strand can not be executed in a single cycle, it causes a rollback to the top instruction in the strand and the instructions in the strand must be individually executed. This misspeculation penalty diminishes processor performance. Next, we evaluate how frequently strands are executed in a single cycle. Figure 6 shows the percentage of instructions whose operands are less than 16-bit. Only instructions included in strands are considered. Since executing 16 instructions in a single cycle is unrealistic, we only consider the collapsing depth of 3, 4, and 5. As you can see, in most programs, more than 70% of instructions require values that are smaller than 16-bit. The collapsing depth does not significantly affect the percentage. The percentage of strand that can be executed in a single cycle will be larger than those shown in the figure. For example, a strand can be easily executed in a single cycle when an operand is larger than 16-bit while the remaining all operands are zero. This will happen frequently, since programs generally have a characteristic of frequent value

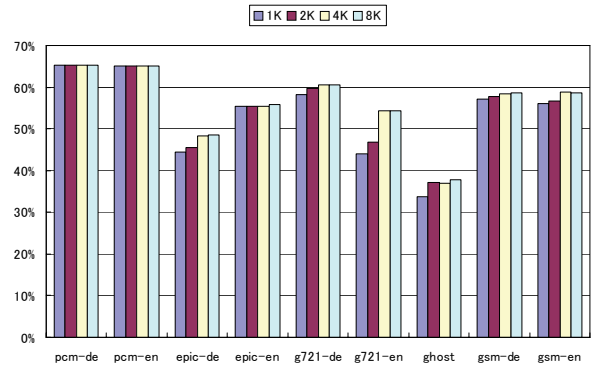

Figure 8. Percent of instructions dispatched as strands

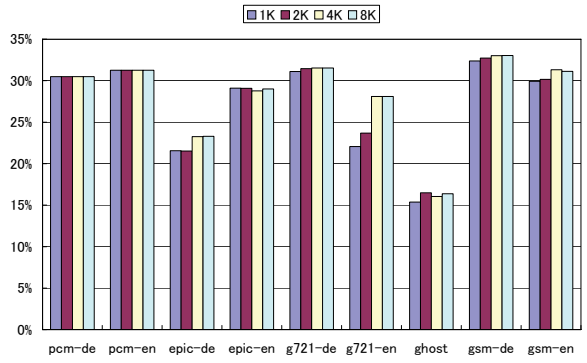

Figure 9. Percent of removed instructions

locality [14, 24] and the most frequent value is zero.

Figure 7 shows the hit ratio of the direct-mapped strand cache. In this evaluation, its number of entries is varied between 1K and 8K, and each entry stores up to three instructions. In other words, every strand consists of less than three instructions. For each group of four bars, the bars from left to right are for the cases where the strand cache has 1K, 2K, 4K, and 8K entries, respectively. It is easily observed that a large cache is not required. The 1K-entry strand cache captures almost all the chances that the 8K-entry cache does. Since it is direct-mapped, it is expected that it works at higher clock frequency and consumes less power than the originally proposed one [12] does while the former has larger cache size than the latter does.

Considering the all observations above, Figure 8 shows the percentage of dynamic instructions that are replaced by strands. The strand cache size is varied between 1K and 8K. For most programs, more than 40% of instructions are replaced by strands even when the strand cache has only 1K entries. This is that interesting since the percentage is much larger than the strand cache hit rate. If the dynamic translation into macro instruction is implemented, some instructions

Figure 10. IPC speedup

are removed and the issue queue, reorder buffer, and register file utilization is improved. Figure 9 shows the percentage of instructions removed by the translation. We can expect the improvement of around 28% in the utilization.

Figure 10 shows the IPC speedup. In this evaluation, only 1K-entry strand cache is considered and successfully speculated strands only require operands less than 16-bit. For each program, the left group of three bars is for the cases where the number of operands provided for each strand is not limited, and the right one is for the cases where the number is limited to two. In other words, in the right group, the number of register file ports is not changed from that of the baseline processor. For each group of three bars, the bars from left to right are for the cases where the collapsing depth is 3, 4, and 5, respectively.

When the increase in register file ports is possible, the IPC speedup is approximately 7.7% on average, regardless of the collapsing depth. In the case of gsm-encode, the IPC speedup is nearly 15% when the collapsing depth is five instructions. The restriction in register file ports seriously affects performance. The average IPC speedup falls down to 5%. Especially, in the case of gsm-decode, the benefit from instruction collapsing is reduced by half.

A possible reason why the IPC speedup is rather small is as follows. By instruction collapsing, processors will benefit from the increase in effective capacities of the instruction issue queue and reorder buffer. The reduction in the pressure on register file write ports will be also beneficial for performance. However, these benefits are not included in the current evaluation. Only the reduction in the pressure on register file read ports are considered. Therefore, the instruction issue queue, the reorder buffer, and the register files become the bottlenecks, even if the data path performance is improves. In some cases, the data path starves for instructions and strands. In other cases, it stalls due to the insufficient writeback and commit bandwidth. The consideration of the benefits above is an important future study.

Since the IPC speedup is much smaller than expected, we can not expect power reduction using DVS. We can reduce clock frequency at most 5% on average. Intel Pentium M reduces power supply voltage from 1.484V to 1.420V while clock frequency is changed from 1.6GHz to 1.4GHz. We can guess that dynamic power consumption is reduced by 8.2% if we decrease clock frequency by 5%. However, it should be noted that the additional circuits, such as the operand table, the strand cache, and the collapsing ALUs, will easily consume power larger than the power saving.

6. Conclusion

The DSM semiconductor technologies will make the worst-case design impossible, since they can not provide design margins that it requires. We have been investigating a typical-case design methodology, which we call the Constructive Timing Violation (CTV). Utilizing the CTV, processors work at higher clock frequency than that the critical path delay determines. In this paper, we proposed to combine the CTV with the instruction collapsing technique. By aggressively and speculatively collapsing instructions, multiple integer ALU operations are executed in a single cycle, resulting in possible improvement in performance and power efficiency.

We proposed simple mechanism to connect an instruction into a strand. It does not rely on any CAMs, and thus both access latency and power consumption are small. From cycle-by-cycle simulations, we found the mechanism effectively captures strands. However, unfortunately, performance improvement was limited and thus we could not expect any power reduction. This is because there are possible bottlenecks in the instruction issue queue, the reorder buffer, and register files. The removal of these bottlenecks is an important future study.

Acknowledgements

This work was supported by PRESTO, Japan Science and Technology Agency, and is partially supported by Grants-in-Aid for Scientific Research #16300019 and #176549 from Japan Society for the Promotion of Science.

References

[1] T. Austin, D. Blaauw, T. Mudge, and K. Flautner, "Making Typical Silicon Matter with Razor", IEEE Computer, vol.37, no.3, March 2004.

[2] A. Bracy, P. Prahlad, and A. Roth, "Dataflow Mini-mraphs: Amplifying Superscalar Capacity and Bandwidth", 37th International Symposium on Microarchitecture, December 2004.

[3] D. Ernst, N. S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, T. Mudge, and K. Flautner, "Razor: a Low-power Pipeline based on Circuit-level Timing Speculation", 36th International Symposium on Microarchitecture, December 2003.

[4] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium 4 Processor", Intel Technology Journal, vol.5, no.1, February 2001.

[5] Y. Kondo, N. Ikumi, K. Ueno, J. Mori, and M. Hirano, "An Early-completion-detecting ALU for a 1GHz 64b Datapath", International Solid State Circuit Conference, February 1997.

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

[7] T. Liu and S-L. Lu, "Performance Improvement with Circuit-level Speculation", 33rd International Symposium on Microarchitecture, December 2000.

[8] S-L. Lu, "Speeding up Processing with Approximation Circuits", IEEE Computer, vol.37, no.3, March 2004.

[9] T. Matsuo, T. Fujikawa, K. Metsugu, and K. Murakami, "Dependable Pipelining: Micro-architecture for the Multi-GHz Generation", Technical Report of IEICE, vol.102, no.262, August 2002 (in Japanese).

[10] K. Mima and T. Sato, "Hardware Cost Reduction in Fault Detection Mechanism for Constructive Timing Violation Technique", 10th International Symposium on Integrated Circuits, Devices and Systems, September 2004.

[11] H. Sasaki, M. Kondo, and H. Nakamura, "Dynamic Instruction Cascading on GALS Microprocessors", IPSJ SIG Technical Reports, vol.2005, no.80, August 2005 (in Japanese).

[12] P. G. Sassone and D. S. Wills, "Dynamic Strands: Collapsing Speculative Dependence Chains for Reducing Pipeline Communication", 37th International Symposium on Microarchitecture, December 2004.

[13] T. Sato and I. Arita, "Give up Meeting Timing Constraints, but Tolerate Violations", COOL Chips IV, April 2001.

[14] T. Sato and I. Arita, "Low-Cost Value Predictors Using Frequent Value Locality", 4th International Symposium on High Performance Computing, May 2002.

[15] T. Sato and I. Arita, "Constructive Timing Violation for Improving Energy Efficiency", in Luca Benini, Mahmut Kandemir, J. Ramanujam, "Compilers and Operating Systems for Low Power", Kluwer Academic Publishers, September 2003.

[16] T. Sato and D. Morishita, "A Field-Customizable and Runtime-Adaptable Microarchitecture", 2nd International Conference on Field-Programmable Technology, December 2003.

[17] Y. Sazeides, S. Vassiliadis and J. E. Smith, "The Performance Potential of Data Dependence Speculation and Collapsing", 29th International Symposium on Microarchitecture, November 1996.

[18] N. R. Shanbhag, "Reliable and Efficient System-on-chip Design", IEEE Computer, vol.37, no.3, March 2004.

[19] A. Tanino and T. Sato, "Evaluating the Potential of an Energy Reduction Technique Based on Timing Constraint Speculation", 4th Workshop on Compilers and Operating Systems for Low Power, September 2003.

[20] A. K. Uht, "Going beyond Worst-case Specs with TEAtime", IEEE Computer, vol.37, no.3, March 2004.

[21] K. Usami, M. Igarashi, F. Minami, T. Ishikawa, M. Kanazawa, M. Ichida, and K. Nogami, "Automated Low-power Technique Exploiting Multiple Supply Voltages Applied to a Media Processor", IEEE Journal of Solid-State Circuits, vol.33, no.3, March 1998.

[22] S. Vassiliadis, J. Phillips, and B. Blaner, "Interlock Collapsing ALU's", IEEE Transactions on Computers, vol.42, no.7, July 1993.

[23] M. Yamahara, K. Mima, and T. Sato, "Fast Fault Detection Circuit for Constructive Timing Violation", IPSJ Kyushu Chapter Symposium, March 2005 (in Japanese).

[24] Y. Zhang, J. Yang, and R. Gupta, "Frequent Value Locality and Value-centric Data Cache Design", 9th International Conference on Architectural Support for Programming Languages and Operating Systems, November 2000.