九州大学学術情報リポジトリ
Kyushu University Institutional Repository

A Simple Mechanism for Collapsing Instructions under Timing Speculation

佐藤, 寿倫
福岡大学

https://hdl.handle.net/2324/6794510

出版情報：IEICE Transactions on Electronics. E91-C (9), pp.1394-1401, 2008-09. IEICE
バージョン：
権利関係：(c) 2008 IEICE

IEICE TRANS. ELECTRON., VOL.E91–C, NO.9 SEPTEMBER 2008

PAPER Special Section on Advanced Processors Based on Novel Concepts in Computation

A Simple Mechanism for Collapsing Instructions under Timing Speculation

Toshinori SATO[†], *Member*

SUMMARY 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. Based on detailed simulations, we find the proposed mechanism effectively collapses dependent instructions.

key words: *Typical-case design, constructive timing violation, instruction collapsing*

1. Introduction

As the complexity of the semiconductor manufacturing process increases, it is likely that process variations will be more difficult to control[2]. The demand for low power leads supply voltage reduction and hence makes voltage variations a serious problem. Higher and higher clock frequency increases temperature variations in a chip. Under these situations, the deep submicron (DSM) semiconductor technologies will make the "worst-case design" impossible, since they can not provide design margins that it requires. In order to realize robust designs, designers have to be aware of design for manufacturing (DFM).

One of the promising solutions is typical-case design methodology, where LSIs should be designed with typical case considerations rather than with worst case considerations. The Constructive Timing Violation (CTV) paradigm[18] is such a design methodology. 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[9]. 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[25]. The CTV relies on timing speculation, and thus sometimes timing violations occur, resulting in logic errors. Some fault tolerance mechanisms are provided for timing violations[14], [18], [23], [28]. 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 as well as for boosting clock frequency[18], and we have evaluated it on carry select adders[8], [23].

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 therefore propose an aggressive combination of the CTV with collapsing ALUs[26]. This makes it possible to execute multiple dependent operations in a single cycle. Its potential in performance gain is significant[20]. 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 introduce the CTV. Section 4 describes the mechanism which dynamically detects a chain of instructions and collapses them into a macro-instruction. Section 5 details evaluation methodology. Section 6 presents simulation results. Finally, Section 7 concludes.

2. Related Work

2.1 Typical-case design techniques

Kondo et al.[7] 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.[13] 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 tim-

Manuscript received 00, 2000.
Manuscript revised 00, 2000.
Final manuscript received 00, 2000.
[†]The author is with Fukuoka University.

ing violations occur on a carry look-ahead adder using Verilog-HDL and logic synthesis. We evaluated it on carry select adders[8], [23].

The characteristics of the critical path can be exploited for robust design. Critical path isolation[5] is a design paradigm, which achieves robustness with respect to parametric delay failures. Critical paths of synthesized design are isolated as a known logic block. This makes it possible to predict the occurrence of any delay violations and thus allows us to aggressively reduce the supply voltage. In input-based elastic clocking[15], the number of clock cycles required for computation is changed depending upon input data patterns. When delay failures are predicted in advance, the clock cycles are increased.

Razor[1], [4] 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[11], [12], algorithmic noise tolerance (ANT)[22], and TEAtime[24]. In the approximation circuits[11], [12], instead of implementing the complete circuit necessary to realize a desired functionality, a simplified circuit is implemented to approximate it. The approximation circuit works at higher frequency than the complete circuit does, and usually produces correct results. If it fails, the system utilizing the approximation circuit has to recover to a safe point. In ANT[22], information theoretic technique is employed to determine the lower bounds on energy and performance. In order to approach these bounds, circuit- and algorithmic-level techniques are evolved. TEAtime[24] uses a tracking circuit to mimic the worst-case delay. As long as the tracking circuit works correctly, clock frequency can be increased and supply voltage can be decreased. Usually, a 1-bit-wise critical path is used for the tracking circuit.

2.2 Instruction collapsing techniques

Vassiliadis el al.[26] 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.[21] 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.[17] 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)[16] is another technique to execute two dependent instructions in a single cycle. During register 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. Constructive Timing Violation Technique

The DSM technologies increase variations, and hence design margins that the conventional worst-case design methodology requires, are reduced. The conservative approach will not work. Considering the situation, we have to change design methodology. Typical-case design methodology is a promising one. It exploits an observation that worst cases are rare. We should focus on typical cases rather than worst cases. Since we do not have to consider worst cases, design constraints are relaxed, resulting in easy designs.

In the typical-case design methodology, we adopt two methods to a design at a time. One is performance-oriented design, where only typical cases are considered. Since worst cases are not considered, design constraints are relaxed, resulting in easy designs. The other is function-guaranteed design. While it requires worst case considerations, designers do not have to consider

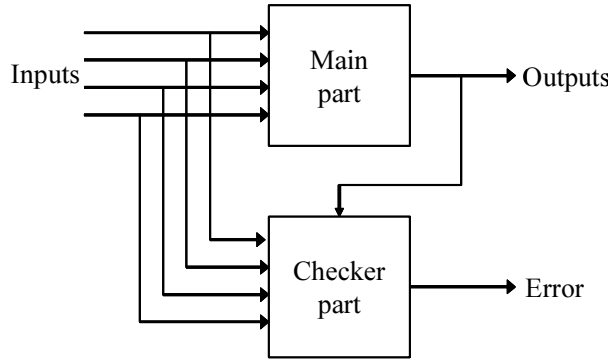


Fig. 1 Constructive timing violation

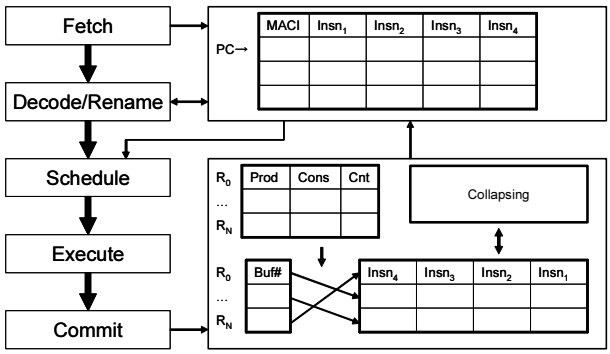


Fig. 2 Strands detection, formation, and dispatch

Insn#0	PC_i	$R_{10} \leftarrow R_1 + R_2$
Insn#1	PC_{i+1}	$R_{11} \leftarrow R_{10} + R_3$
Insn#2	PC_{i+2}	$R_{12} \leftarrow R_{10} + R_{11}$
Insn#3	PC_{i+3}	$R_{10} \leftarrow R_4 + R_5$
Insn#4	PC_{i+4}	$R_{11} \leftarrow R_6 + R_7$

Fig. 3 Instruction sequence example

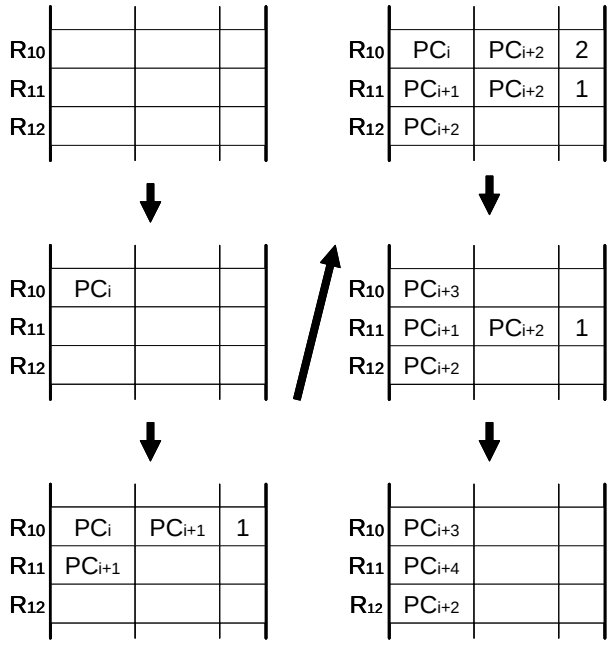


Fig. 4 Strands detection example

performance but have to guarantee functions. Hence designs must be simple, resulting in easy verifications.

The CTV paradigm[18] is such a design methodology, where designers are focusing on typical cases rather than are 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[9]. 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[25]. The CTV relies on timing speculation, and thus sometimes timing violations occur, resulting in possible logic errors. Some fault tolerance mechanisms are provided to recover circuits from logical errors.

The concept of the CTV is as follows. We design every timing critical function in a chip by two methods. The design consists of two components as shown in Fig. 1. One is called main part, and the other is called checker part. While two parts share the single function, their roles and implementations are mutually different. The main part is designed with performance considerations, but might cause timing violations. That is, it is implemented by the performance-oriented design. The checker part is a safety net for the main part. It detects timing violations that occur in the main part, and thus it has to satisfy all timing constrains in the chip. However, designers do not have to optimize performance nor power but only have to guarantee the function. That is, it is implemented by the function-guaranteed design. If a timing violation is detected by the checker part, the circuit state has to be recovered to a safe point by any means.

4. Instruction Collapsing Boosted by CTV

4.1 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[17]. Each strand is speculatively executed in a single cycle, and hence dependent instructions are collapsed.

In order to detect transient operands, we utilize the operand table[17] shown in the middle of Fig. 2. 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 Fig. 2),

the consumer field (`Cons` in Fig. 2), and the consumer counter (`Cnt` in Fig. 2), 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.

Using an example of an instruction sequence shown in Fig. 3, we explain how the operand table works. Figure 4 shows how a transirnt operand, R_{11}, is detected. The operand table entries for registers R_{10}, R_{11}, and R_{12} are focused. First, all entries are invalidated and their fields are vacant. When Insn#0 is committed, its PC, PC_i, is registered in the `Prod` field of R_{10}. Next, when Insn#1 is committed, its PC, PC_{i+1}, is registered in the `Prod` field of R_{11} and in the `Cons` field of R_{10}. The `Cnt` field of R_{10} is set to 1. Similarly, when Insn#2 is committed, PC_{i+2}, is registered in the `Prod` field of R_{12} and in the `Cons` fields of R_{10} and R_{11}. The `Cnt` field of R_{10} is incremented to 2 and that of R_{11} is set to 1. When Insn#3 is committed, the entry of R_{10} already has valid values. Its `Prod` field should be replaced by PC_{i+3} and its `Cons` and `Cnt` fields should be cleared. Before doing that, the `Cnt` field is checked. Because its value is 2, the last R_{10} is not a transient operand. Similarly, when Insn#4 is committed, R_{11}'s `Prod` field should be replaced by PC_{i+4} and its `Cons` and `Cnt` fields should be cleared. The `Cnt` field is checked and its value is 1. Hence, a transient operand R_{11} is detected.

4.2 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 Fig. 2.

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 Fig. 2, 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 Fig. 2) 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.

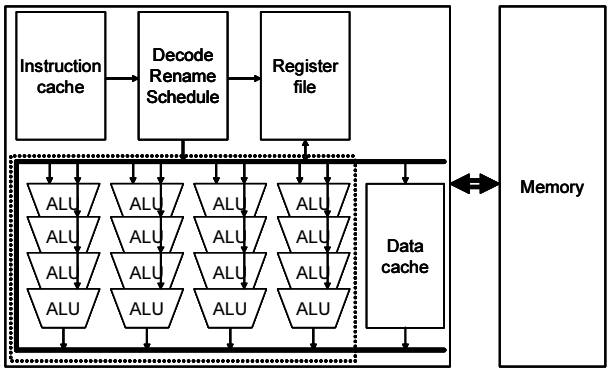


Fig. 5 Instruction collapsing

4.3 Strands dispatch

After a strand is formed, it is registered in the strand cache shown in the top of Fig. 2. 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 (`Insn1` in Fig. 2). In order to identify the strand as a macro-instruction, it has a strand identifier (`MACI` in Fig. 2).

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.

When all source operands except transient operands produced by the strand are ready, it is issued into a collapsing ALU. As shown in Fig. 5, 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[3], 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. 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. Bypassing paths are provided only from the bottom ALUs to the top ones. Intermediate values are not bypassed. This does not affect performance, since intermediate values are transient operands.

4.4 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

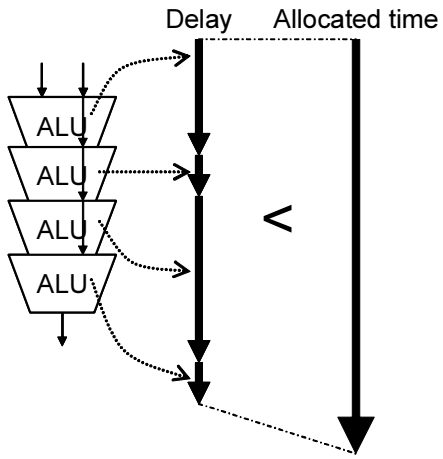


Fig. 6 Combining Instruction collapsing with CTV

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

4.5 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[14], [18]. 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. Currently, we write transient values into the register file as well.

Table 1 Processer 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

5. Methodology

We will evaluate the potential of the proposed technique based on some assumptions explained below. We use 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. Currently, our simulator has the following restriction. The oracle scheduler is assumed and only correctly-speculated strands are dispatched and executed. Hence, evaluations do not suffer from any misspeculation penalties and the results will be slightly optimistic.

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 possible assumption as follows. Modern processors spend half of the execution cycle on ALU execution and half on full bypass[17] and Intel Pentium 4 perform two 16-bit ALU operations in a single cycle[6]. Some specialized

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

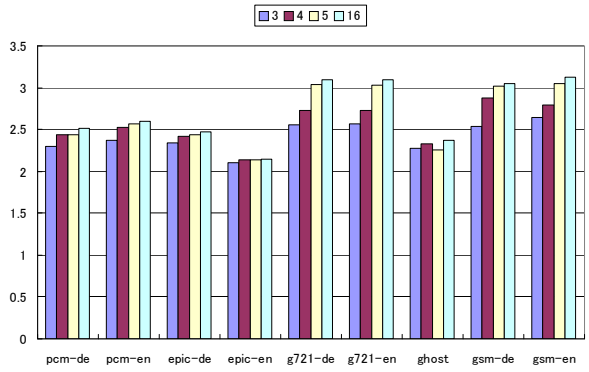


Fig. 7 The number of instruction per strand

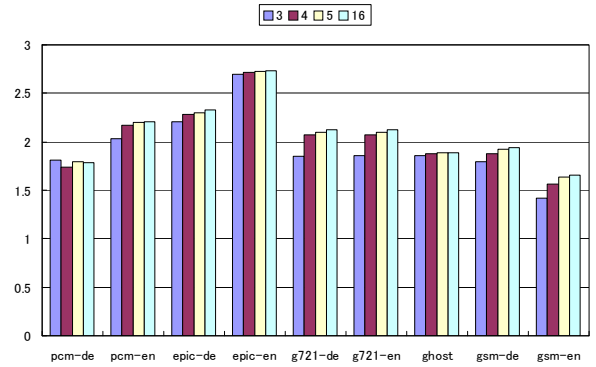


Fig. 8 The number of operands per strand

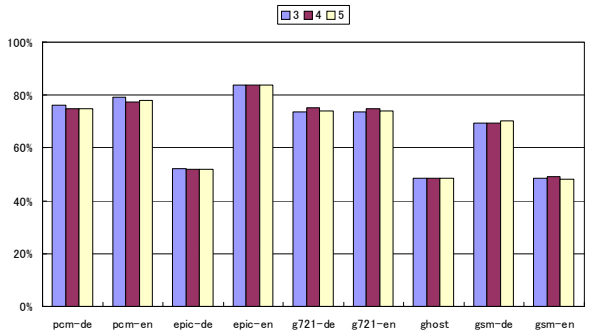


Fig. 9 Percentage of instructions with less-than 16b operands

or hybrid circuits could be used for reducing execution latency. It is well known that carry save adder used in the collapsing ALU[26] operates multiple inputs in a small delay. We have already found that the increase of 25% in circuit delay made it possible to cascade two adders[27]. Considering obove, tightly-connected ALUs might 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[10] 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.

6. 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 Fig. 7. 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.

The average number of operands requested by every strand is shown in Fig. 8. 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. 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

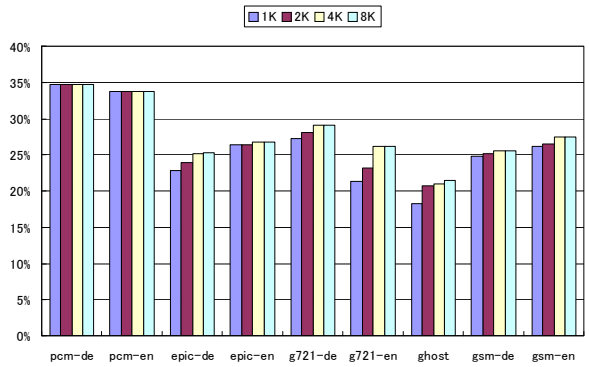


Fig. 10 Strand cache hit rate

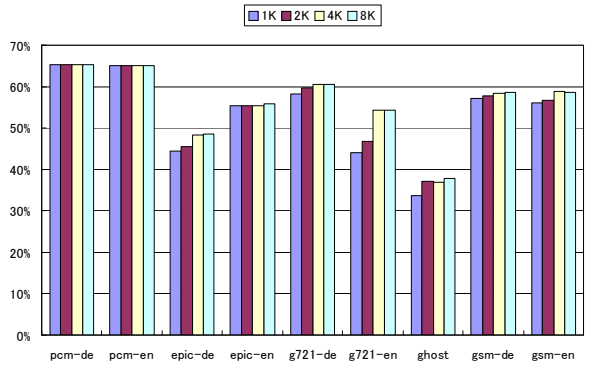


Fig. 11 Percentage of instructions dispatched as strands

executed. This misspeculation penalty diminishes processor performance. Next, we evaluate how frequently strands are executed in a single cycle. Figure 9 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 locality[19], [29] and the most frequent value is zero.

Figure 10 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,

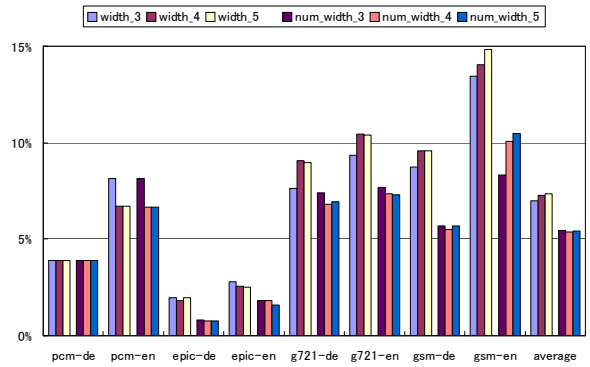


Fig. 12 IPC speedup

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[17] does while the former has larger cache size than the latter does.

Considering the all observations above, Figure 11 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.

Figure 12 shows the IPC (Instructions committed Per Cycle) 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 read 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.

While the aggregated performance gain is not encouraging, there are some potential to boost performance. First, the dynamic instruction scheduler will be improved. If each strand were handled as if it were

a quasi instruction, the efficiency in the instruction scheduler entries would be improved. Since the instruction scheduler is a critical structure in modern superscalar processors, here is a potential for performance gain. Second, the pressure on registers will be mitigated. If mechanisms for precise interruption or further novel concepts on computation were invented, transient operands should not be written into registers. Since the number of register file entries is also critical in modern processors, here is another potential for performance gain.

7. Conclusions

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. 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.

Acknowledgements

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

References

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

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

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

[4] 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, 2003.

[5] S. Ghosh, S. Bhunia, and K. Roy, "A New Paradigm for Low-power, Variation-Tolerant Circuit Synthesis Using Critical Path Isolation", International Conference on Computer Aided Design, 2006.

[6] 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, 2001.

[7] 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, 1997.

[8] Y. Kunitake, A. Chiyonobu, K. Tanaka, and T. Sato, "Challenges in Evaluations for a Typical-Case Design Methodology", 8th International Symposium on Quality Electronic Design, 2007.

[9] T. Kuroda, "10 Tips for Low Power CMOS Design", Tutorial, Design Automation Conference, 2003.

[10] 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, 1997.

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

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

[13] 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, 2002 (in Japanese).

[14] 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, 2004.

[15] 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.

[16] H. Sasaki, M. Kondo, and H. Nakamura, "Dynamic Instruction Cascading on GALS Microprocessor", International Workshop on Power and Timing Modeling, Optimization and Simulation, 2005.

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

[18] T. Sato and I. Arita, "Potential of Constructive Timing-Violation", IEICE Transactions on Electronics, vol.E85-C, no.2, 2002.

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

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

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

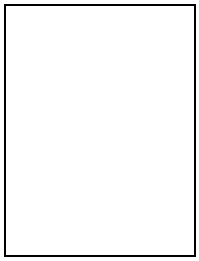
[22] N. R. Shanbhag, "Reliable and Efficient System-on-chip Design", IEEE Computer, vol.37, no.3, 2004.

[23] 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, 2003.

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

[25] 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, 1998.
[26] S. Vassiliadis, J. Phillips, and B. Blaner, "Interlock Collapsing ALU's", IEEE Transactions on Computers, vol.42, no.7, 1993.
[27] S. Watanabe, M. Hashimoto, and T. Sato, "ALU Cascading for Improving Performance Yield", IPSJ SIG Technical Report, 2008-ARC-177, 2008 (in Japanese).
[28] M. Yamahara, K. Mima, A. Chiyonobu, and T. Sato, "A Fast Fault Detection Circuit for Low-power Adders with Timing Error Tolerance", IPSJ Transactions on Advanced Computing System, vol.47, no.SIG18 (ACS16), 2006 (in Japanese).
[29] 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, 2000.



Toshinori Sato received the BS, MS, and PhD degrees in electronic engineering from Kyoto University in 1989, 1991, and 1999, respectively. From 1991 to 1999, he was with Toshiba Coorperation, where he was engaged in the research on multicore processors and the developments of embedded processors. He served on the faculty of Department of Artificial Intelligence at Kyushu Institute of Technology until 205 and of System LSI Research Center (SLRC) at Kyushu University until 2008. He is currently a professor of Depertment of Electronics Engineering and Computer Science at Fukuoka University. He also serves as a research professor of SLRC. His research interests include microprocessor architecture and design methodology. He is a member of IPSJ and a senior member of ACM and IEEE.