IPSJ Transactions on Advanced Computing Systems Vol. 2 No. 2 30–47 (July 2009)

Regular Paper

An Instruction Scheduler for Dynamic ALU Cascading Adoption

Jun Yao,[†1] Kosuke Ogata,[†2] Hajime Shimada,[†1]
Shinobu Miwa,[†3] Hiroshi Nakashima[†1]
and Shinji Tomita[†1]

To reduce the processor energy consumption under low workload and low clock frequency executions, a possible solution is to use ALU cascading while keeping the supply voltage unchanged. This cascading scheme uses a single cycle to execute multiple ALU instructions which have a data dependence relationship between them and thus saves clock cycles for the whole execution. Since the processor energy consumption is the product result of both power and execution time, ALU cascading is expected to help energy optimization for microprocessors operating under low frequency status. To implement ALU cascading in a current superscalar processor, a specific instruction scheduler is required to wakeup a pair of cascadable instructions simultaneously despite there being a data dependence relationship between them. Furthermore, ALU cascading is only applied under low clock frequency execution mode so that the instruction scheduler must support standard scheduling for the normal clock frequency execution. In this paper, we propose an instruction scheduling method that enables the additional wakeup features for the utilization of ALU cascading without large hardware extensions. With this scheduler, the average IPC improvement becomes 3.7% in SPECint2000 and 6.4% in Mediabench, as compared to the baseline execution. The delay of additional hardware required for the ALU cascading purpose is also evaluated to study the complexity of ALU cascading.

1. Introduction

Nowadays, the design of microprocessors has entered a period when power and thermal problems become major restrictions for the advancements of processor performance. To relieve this problem, many mechanisms either in architecture or circuit fields have been designed for the optimization of energy utilization. In recent years, Dynamic Voltage and Frequency Scaling (DVFS) as an architecture level energy saving technique, has been widely studied and adopted in both researches and industries.

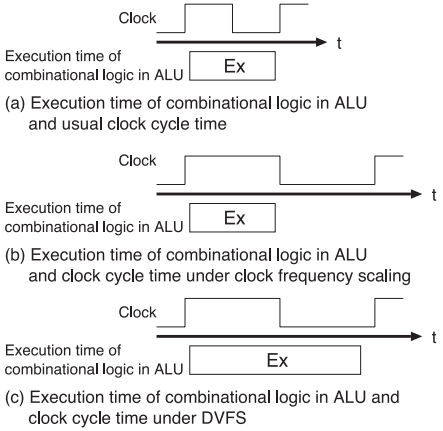


Fig. 1 Extended clock cycle time under DVFS-like techniques.

As shown in **Fig. 1** (a), in general pipeline processors, a combinational logic takes a full clock cycle time to finish its operations. When the processor downscales the clock frequency, the combinational logic can finish operations prior to the next clock pulse (Fig. 1 (b)). DVFS uses this idle period for a scaling of the supply voltage which extends the operation time of the combinational logic (Fig. 1 (c)). Since the supply voltage takes a quadric factor in power equation, DVFS has great efficiencies in energy consumption reduction. However, concerns also arise for the adoption of DVFS in future semiconductor process technology, where the supply voltage scaling may be difficult because of process deviations, noise margins, restrictions on the threshold voltage scaling caused by leakage current, and so on. For these reasons, DVFS may not be as efficient in future processors.

There is another idea which makes use of the idle part of the clock cycle time shown in Fig. 1 (b). The idea is expressed as data collapsing [1] or ALU cascading [2)–4)]. **Figure 2** (a) shows an example of cascaded ALU operations. Assume that two arithmetic instructions $i1$ and $i2$ have a data dependence relationship.

†1 Graduate School of Informatics, Kyoto University
†2 Information Technology R&D Center, Mitsubishi Electric Corporation
†3 School of Engineering, Tokyo University of Agriculture and Technology

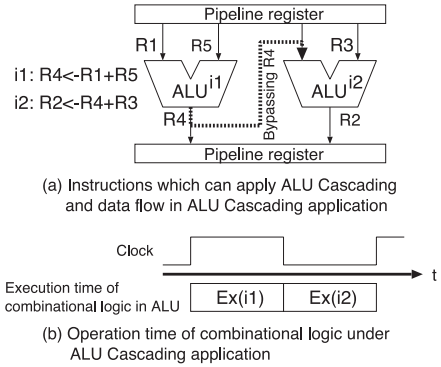


Fig. 2 ALU Cascading.

Supposing that the frequency is scaled to half of the maximum value, the processor can connect an output of an ALU to an input of another ALU and bypass the result of $i1$ to $i2$. In the first half of the clock cycle time, the arithmetic of $i1$ is executed in the left ALU. After the execution, the result of $i1$ is sent to both the pipeline register and an input of the right ALU. In the latter half of the clock cycle time, the arithmetic of $i2$ is executed in the right ALU. By this means, ALU cascading can utilize the latter half of the clock cycle time for a second operation, as shown in Fig. 2 (b). As the two instructions are now finished in a single cycle, ALU cascading is expected to increase the Instructions Per Cycle (IPC) for the program execution. Therefore, it may accelerate program execution, or provide opportunities for further frequency scaling while keeping a given throughput. Both of these two utilizations help achieve additional energy reductions for the already decreased power consumption by the frequency downscaling.

There are several studies on ALU cascading utilizations in media processors and vector processors. However, there are few researches on superscalar processors. It is partly because of the complexity in appending new features for the cascaded execution scheduling to the instruction scheduler in superscalar processors. The difficulties may come from the following aspects.

(1) Current schedulers are specialized to wakeup dependent instructions in the instruction window in consecutive cycles.

(2) The instruction scheduler is one of the bottlenecks for clock cycle time scaling. Therefore, a large overhead introduced from additional features is not acceptable.

(3) The gains from ALU cascading are relatively insignificant so that the additional hardware must be minimized, so as not to overwhelm the achievements.

(4) ALU cascading is only applied in low frequency mode. The instruction scheduler must support standard scheduling for the normal frequency execution.

In this paper, we propose an instruction scheduler which supports ALU cascading in view of the above problems. The proposed scheduler is based on the Dependence Matrices Table (DMT) instruction scheduler [5)–7)]. To support scheduling for ALU cascading, we extended the utilization of the dependence matrix table, and added data path for cascading bypasses. Our scheduler in this paper achieves a better performance than the prior instruction grouping method which is another ALU cascading implementation, by exploiting more cascading opportunities, even with a smaller hardware cost. The delay of additional logics is also evaluated in our research.

The rest of this paper is organized as follows. Section 2 introduces the DMT instruction scheduler as a baseline instruction scheduler of our proposal. Section 3 introduces the instruction grouping method which is a different implementation of instruction scheduling method with ALU cascading. Section 4 describes our designed instruction scheduler that supports ALU cascading. Section 5 presents the evaluation results of IPC improvements after different ALU cascading scheduling policies. A preliminary implementation of the circuit to execute cascaded instructions is given in Section 6, together with some overhead analyses. Section 7 shows some other related works. Finally, Section 8 concludes the paper.

2. Dependence Matrices Table (DMT) Based Instruction Scheduling

In this section, we introduce the Dependence Matrices Table (DMT) instruction scheduler [5)–7)] which is the baseline instruction scheduler of our proposal.

2.1 Outline of DMT Structures

DMT was introduced as a high-speed instruction scheduling logics to accelerate

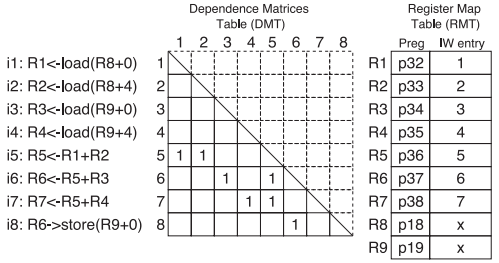


Fig. 3 DMT and its corresponding Register Mapping Table (RMT).

the wakeup-selection phase. **Figure 3** shows the structures required by the DMT instruction scheduler, working as an alternative to the conventional CAM based instruction scheduling logics. The left part of Fig. 3 shows DMT structure. The dependences between instructions are stored in a bottom-left $WS \times WS$ matrix structure[⋆1], where WS is the actual instruction window size. DMT can be implemented as a SRAM array in which each cell takes a 1-bit storage. The row and column numbers denote the entry numbers of the instruction window. Assume that the c-th instruction (consumer) depends on the result generated by the p-th instruction (producer), where c and p are their positions in the instruction window. The DMT element at row c and column p will be set to 1 accordingly, representing the dependence relationship and its direction as well. As an example, an instruction $i5$ in the left of Fig. 3 has the dependences $i1 \rightarrow i5$ and $i2 \rightarrow i5$. To represent these dependences, the DMT elements at positions $(producer, consumer)$ as $(1, 5)$ and $(2, 5)$ will be flagged. Other flags in the figure will be set similarly as shown in Fig. 3. Rows with no flags like $i1$ to $i4$ in Fig. 3 denote that the corresponding instructions have no unresolved dependences so that they are marked as operand-ready.

Moreover, to update the DMT correctly, Register Mapping Table (RMT) which is originally prepared for register renaming is extended to indicate the position of the producer instruction. As shown in the right part of Fig. 3, Each RMT entry represents the relationship between a logical register number and an assigned physical register number ($Preg$). The column "*IW entry*" is added to save a pointer to the instruction that updates the $Preg$ content. It is represented as the entry index in the instruction window (IW). Considering the example shown in Fig. 3, $R5$ entry shows that the 5th positioned instruction in IW updates physical register $p36$, which is assigned to this logical register $R5$. A successive instruction finds the positions of its producers by looking up this table with both source logical register numbers.

⋆1 The up-right matrix structure is also used to support circular usage of instruction window (e.g., $i1$ again after i_{max}).

2.2 Detailed Operations of DMT under Instruction Scheduling

The information stored in DMT is useful to trace the data dependence relationships, as well as the status of the resolved dependences. The operations of DMT will be illustrated with a sample in this section. Assume that a 4-way out-of-order superscalar processor is operating on the instruction series as in **Fig. 4**. To help the introduction, we use a simple pipeline architecture which contains Fetch, Decode, Issue, Execute, Writeback and Commit stages to illustrate the DMT operations. Before the moment in Fig. 4 (a), the instructions prior to $i1$ have been executed. Suppose that we are under an initial status that $i1$ to $i4$ are decoded and stored in the IW. The DMT and RMT now represent the information of these four instructions. At this point, no DMT elements are flagged since the four load instructions are assumed to depend on issued instructions. The operating status related to DMT and RMT in the next cycle will conform to the following steps.

(1) Decoding of block from $i5$ to $i8$

Consider that the four instructions from $i5$ to $i8$ enter the decode phase at the same cycle. To set flags of DMT appropriately, the processor has to detect the IW entry number of the producer instructions for the two source operands in each decoding instruction. In the decode stage, this operation is done with two micro-actions. Firstly, dependences between the simultaneously decoded instructions are checked by comparing their source logical register numbers with destination logical register numbers. Secondly, source logical register numbers are used to lookup RMT and get their corresponding contents in "*IW entry*". To reduce delay, those two actions can be done simultaneously. In the event that multiple results are retrieved for a single source operand, the logics will choose the information of the latest producer.

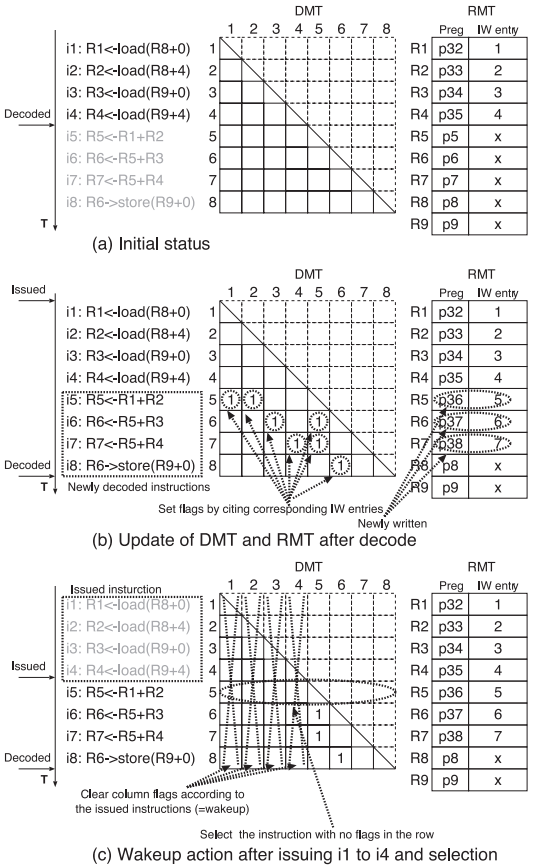


Fig. 4 Instruction scheduling based on DMT.

We use instructions $i5$ and $i6$ as the example. Under the first dependence check action, the processor compares two source logical register numbers of $i6$ ($R5$ and $R3$) with the destination logical register number of $i5$ ($R5$). As a result, the processor detects that the left source operand of $i6$ has a dependence to $i5$. Also under the second step, the processor looks up RMT with source logical register numbers $R1$, $R2$, $R5$, and $R3$. After these lookups, the logic detects their dependences on $i1$, $i2$, and $i3$, respectively. By putting priority on the first action, the dependence of $i6$'s left source operand will be set to $i5$. All these dependences detected by either comparisons or RMT lookups, will then be translated to the DMT element indices as $(1, 5)$, $(2, 5)$, $(3, 6)$, and $(5, 6)$. They will be stored in the matrix by putting flags in those memory cells. The row vectors of $i7$ and $i8$ in the DMT are constructed by the same process as shown in Fig. 4 (b).

Meanwhile, information for destination registers related to $i5$ to $i8$ (allocated physical register number and IW entry) will be updated into RMT entries accordingly. The dashed lines and ellipses in Fig. 4 (b) demonstrate these information updates in the decode cycle of this instruction block.

(2) Wakeup/select after the issue of block from *i*1 to *i*4

Assume that the four load instructions (from $i1$ to $i4$) have been selected and passed to the execution phase simultaneously. The wakeup operation after the issue of these four instructions is done by clearing the DMT columns indexed by the entries of issued instructions. As a result, the dependences to the issued instructions are now marked as resolved or none in the updated DMT. The clearing operation is demonstrated as the dashed crossing in Fig. 4 (c). After this clearing operation, the columns which have been cleared are forced to be zero even if a newly decoded instruction tries to set the flag.

After the clearing actions, the vector of row 5 in the DMT structure has been totally cleared and vector bit OR result becomes 0. It indicates an operand-ready status for the corresponding instruction $i5$. At this point, the other row vectors still have flagged bits and cannot be marked as operand-ready.

In the select phase, $i5$ will be a candidate for issue, taking a data forwarding path from the execution stages of both $i1$ and $i2$ to accelerate the program execution. In a following cycle, the issue of $i5$ will clear DMT column 5 and thus $i6$ and $i7$ become operand-ready.

By these DMT manipulations, the instruction scheduler can wakeup all the consumer instructions when the producer is about to execute.

3. ALU Cascading Adoption with Instruction Grouping Method

There is a similar research which added features into the instruction scheduler to achieve ALU cascading [8]. In that paper, they gave a specialized instruction window (IW), as shown in the right part of **Fig. 5**. Each entry of the modified

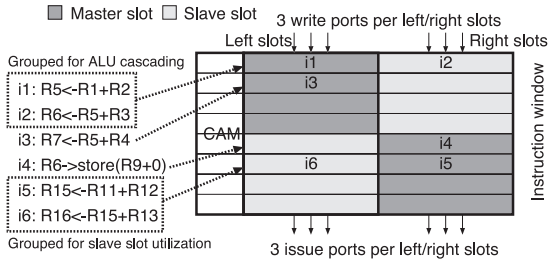


Fig. 5 Outline of Instruction Grouping Method.

IW now contains two slots, known as the master and slave. With these two slots, they could group a producer instruction p and a consumer instruction c which has no unresolved dependence other than p, into a package. The instruction package is then fitted into the master and slave slots in one instruction entry. The wakeup of the master slot will also wake the consumer instruction in the corresponding slave slot. As an example, $i1$ and $i2$ in the sample code in Fig. 5 will be put into one entry since they satisfy the grouping conditions.

If the processor uses the slave slot for the purpose of ALU cascading only, the utilization is limited by the rate of cascadable instructions and will not be high. Assume that data dependences to registers $R11$, $R12$, $R15$ and $R13$ in the example have already been resolved when $i5$ and $i6$ enter the decode stage. In this case, the processor can apply a VLIW-like grouping like $i5$ and $i6$ in Fig. 5 which have no unresolved dependences to improve the slave slot usage. Moreover, a feature that swaps the master and slave slots in the latter half part of the IW (as in Fig. 5) has been employed to balance the read/write port usage of left/right slots.

This method gives additional performance improvements from both the cascaded operations and increased in-flight instructions from slave slots utilization. However, there are still some restrictions in this method.

- ALU cascading is only applied to a single producer/consumer pair despite whether or not there are any other consumer candidates.
- Only one unresolved dependence is allowed in the consumer before it can be grouped into the cascaded pair in decode stage.
- The total read/write ports are distributed equally to the left/right slots to use both slots, as in Fig. 5. This organization gives additional restrictions in the instruction issue (e.g. if ready instructions are concentrated in the left slots, the processor can only utilize half of the total issue width.).
- Additional costs will be introduced to implement the slave slots.

Our proposal in Section 4 relieves all the above restrictions by using an extension on the DMT instruction scheduler without touching the instruction window. However, as both methods give ALU cascading implementations, the comparison results of this grouping method and our proposal will be studied in Section 5.

4. Dynamic Instruction Scheduling with ALU Cascading Based on Extended DMT

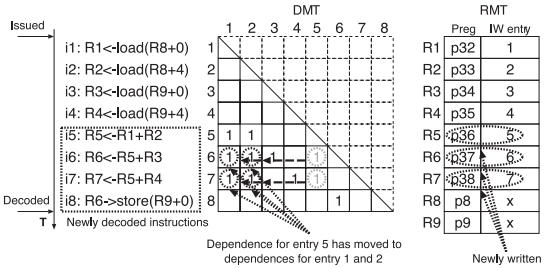
Based on the background technology introduced in Section 2, we designed an instruction scheduling scheme using ALU cascading. As in the instruction series shown in Fig. 4, the destination operand of $i5$ serves the left source operand for either $i6$ or $i7$. If the right source operand for $i6$ or $i7$ has already been resolved, the processor can apply ALU cascading as $i5 \rightarrow i6$ or $i5 \rightarrow i7$. However, to issue $i6$ or $i7$ at the same cycle of $i5$'s issue, the operand-ready marks of $i6$ and $i7$ are required to be set one hop prior—at the same point when the source operands of $i5$ are ready. For this purpose, we modified the DMT method to support ALU cascading.

4.1 Modified DMT Operating Scheme to Support ALU Cascading

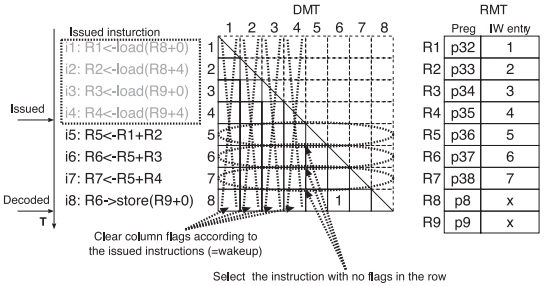
We use the same instruction series as in Fig. 4 to illustrate the differences in the instruction scheduling, with the wakeup supports for the ALU cascading. Similarly like Section 2.2, the initial status of the table structures used in the new scheduler is the same as Fig. 4 (a).

(1) Decoding of instruction block from $i5$ to $i8$

Though the hardware structures of DMT and RMT are not changed in the design, we included some modifications in the DMT flag setting part to support the wakeup of cascadable instructions. The main idea is to replace the dependent bit of a consumer instruction from its direct producer to the two-hop away producers (as shown in **Fig. 6** (a)). Thus, from the grandparent producer instruction's viewpoint, its selection can wakeup both its direct consumer and two-hop away consumers by a single clearing action. To achieve this, the dependent check logics



(a) Update of DMT and RMT after decode



(b) Wakeup action after issuing i1 to i4 and selection

Fig. 6 Modified DMT based scheduling with ALU cascading.

are modified to perform one more iteration of producer back-tracing. Similarly as in the introduction of DMT, we use instruction block from $i5$ to $i8$ to describe the actual actions performed in this decode stage.

I) Read access of RMT

As in Section 2.2, source logical register numbers are used to lookup RMT. For instruction $i5$ to $i8$, the processor can establish the direct producer/consumer relationships $i1 \rightarrow i5$, $i2 \rightarrow i5$, $i3 \rightarrow i6$, and $i4 \rightarrow i7$ by this lookup.

II) Detecting the two-hop away producer/consumer chain

At the same time as the RMT lookups, dependence checks are performed between the instructions that decoded simultaneously. In detail, the source and destination logical register numbers of instructions from $i5$ to $i8$ will be compared. By this comparison result, the processor can find the $i5 \rightarrow i6$, $i5 \rightarrow i7$, and $i6 \rightarrow i8$ dependences. By combining this detection and *I)*, we can achieve back-trace to the two-hop away producer instructions. For example, with the detection of $i5 \rightarrow i6$ from this source and destination register numbers comparison and the detection of $i1 \rightarrow i5$ and $i2 \rightarrow i5$ dependences from the RMT lookup in the same decode phase, the vision of two-hop dependence chains of both $i1 \rightarrow i5 \rightarrow i6$ and $i2 \rightarrow i5 \rightarrow i6$ can be established.

III) Updating DMT according to two-hop away producers

Based on *II)*'s result, together with the knowledge that $i5$ and $i6$ are both ALU operations, we can set them as a candidate pair for the ALU cascading. Under this situation, the two-hop away providers are used to wakeup the left source operand of $i6$. If $i6$'s right source operand can be ready no later than $i5$'s wakeup, $i6$ can be woken up at the same time as $i5$. Specifically, for the constructing of $i6$'s left source operand related DMT row vector, we designed a logic that follows these steps: 1) The usual scheduling bit vector for $i6$'s left source is "00001000" according to the direct dependence detected in *II)*; 2) $i5$ is in the same decode phase and the result from *I)* will give its dependence bit vector as "11000000" for its source operands $R1$ and $R2$; 3) choose the producer $i5$'s dependence bit vector by using 1)'s result as a selection information. By these steps, the dependence vector for $i6$'s left source operand has been updated to its producer's, as "11000000". The DMT elements at the positions of $(producer, consumer)$ as $(1, 6)$ and $(2, 6)$ will be flagged, instead of $(5, 6)$ which is used in the original DMT design (The dashed arrow in DMT row 6 in Fig. 6 (a) demonstrates this modification).

We can also detect that $i5$ and $i7$ are candidates for the ALU cascading, following a similar process. The DMT element $(1, 7)$ and $(2, 7)$ will be flagged accordingly.

IV) Other DMT updates

For source operands without the two-hop away producer information, the direct producers will be used to update DMT. In this example, DMT cells at positions $(1, 5)$, $(2, 5)$, $(3, 6)$, $(4, 7)$, and $(6, 8)$ are updated under the conventional scheme.

By using these four steps, DMT element can now be used to indicate the relationship between the two-hop away producers and the consumer instructions. Studying the process from the producer's viewpoint, the column vector of $i1$ now contains its nearest indirect consumers' information, which can be used for

wakeup/selection in latter cycles.

(2) Wakeup-select stage after the issue from $i1$ to $i4$ block

After the selection of the four instructions from $i1$ to $i4$, the issue logic will clear their corresponding DMT columns, as Fig. 6 (b) shows. As a consequence, $i5$, $i6$ and $i7$ will be marked as operand-ready since the newly updated DMT indicates all the three corresponding row vectors are without unresolved dependence flags.

With sufficient ALU resources, $i5 \rightarrow i6$ and $i5 \rightarrow i7$ chains can be filled into the cascaded ALUs path. Here, $i5$ serves as the same producer for both $i6$ and $i7$. We call it 1-to-N ALU cascading. The detailed execution after issue is described in Section 6.1.

If the ALU resources are insufficient for the cascading purpose, issuing all ready instructions is not possible. The execution will fall back to the conventional scheme automatically by setting priority on older instructions as the usual instruction scheduling. For example, if there is only one free ALU at the Fig. 6 (b) situation, only $i5$ will be selected and issued. $i6$ and $i7$ will be issued in following cycles and there's no difference compared to the usual execution. By applying priority to older instructions, we don't have to add an additional arbiter which prevents the consumer running ahead of the producer.

For this requirement, it is better to use the oldest-first selection policy [9] in implementing the selection logic. Considering another popular selection policy which is based on location [10], it will always grant the leftmost instruction window entry with the highest priority. It will fail when above cascadable chain $i5 \rightarrow i6$ crosses the instruction window boundary, back to top, especially under a shortage of free ALU resources. Nevertheless, as indicated in paper 10), the location-based selection logic can also be used to implement the oldest-first policy by applying instruction window left compacting. Or we can add specific rules to prevent above $i6$'s selection prior to $i5$. These supplemental schemes may help our instruction scheduler to work with a location-based selection policy. However, they come with the cost of performance degradation so that oldest-first selection is assumed in this research.

4.2 Extension to Support Cascading among Distant Instructions

We use distant instructions here to refer the instructions beyond a simultaneous decode stage. Considering the detection of two-hop away producer information

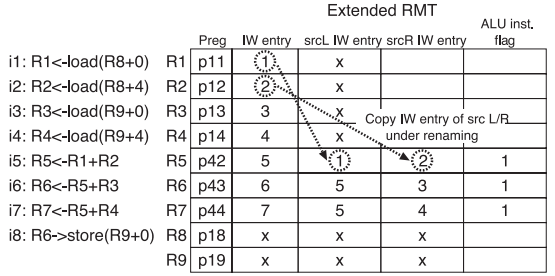


Fig. 7 Extended RMT to support ALU cascading among distant instructions.

from RMT in one single lookup, ALU cascading between the instructions inside the whole instruction window can be possible. **Figure 7** presents the extended RMT (eRMT) to achieve this idea. In this figure, values of *"src L/R IW entry"* are appended to save the information of two parent instructions of the instruction in *"IW entry"*. *"ALU inst. flag"* is added to denote whether the instruction in *"IW entry"* is of ALU operation or not. The constructing of eRMT line will follow the dashed circles and arrows in Fig. 7.

Assume that instructions $i5$ and $i6$ are decoded in separate cycles. When $i6$ is under decoding, lookups of eRMT will be done by using its two source operands. As for right source operand $R3$, no difference will arise as compared to Fig. 6 (a). For the left operand $R5$, both $i5$ and $i1$, $i2$ will be returned, as the direct producer and the parents of direct producer $i5$. And since the *"ALU inst. flag"* bit in the $R5$ row is 1, the cascadable pair of $i5 \rightarrow i6$ can be determined. DMT elements of $(1, 6)$, $(2, 6)$, and $(3, 6)$ will then be flagged, which is the same as Fig. 6 (a).

For each instruction, the eRMT update is performed in the following steps: (1) Read the two producers' information in eRMT with the source logical register numbers; (2) Write allocated physical register number, *IW entry*, and the producers' *IW entry* from (1) into the eRMT entry (in Fig. 7) indexed by the destination logic register. These steps may trigger a read-after-write operation if the producer of this instruction is in the same renaming phase. However, as the usual renaming also covers the read-after-write operation like in instruction $R3 \leftarrow R3 + R4$, the eRMT update will not extend the renaming delay.

By utilizing this extended RMT, the processor can establish the two-hop pro-

ducer/consumer chain like $i1 \rightarrow i5 \rightarrow i6$ by a single eRMT lookup even if $i5$ and $i6$ are in different decode phases. However, this extension doubles the area of RMT which is frequently accessed. Using CACTI 4.2 [11] tool set to give an evaluation, we found that the energy consumption penalty caused by the increased area is in the same order as the possible savings from the improved performance. For these reasons, we limit the utilization of ALU cascading inside the simultaneously decoded instructions. As shown in Section 5.2, ALU cascading is still effective even if we prohibit the cascading between distant instructions. Also, Section A.1 gives a study of the distribution of the distance between cascaded instructions detected with this eRMT structure. It may help define the suitable cascading opportunity search range.

5. Performance Simulation Results

As introduced in Section 1, our goal is to enhance current low energy techniques by employing the previously proposed idea of ALU cascading. According to its working scheme, ALU cascading can save execution time by the cascaded executions in Fig. 2 under a halved operating frequency. Since the cascading does not require voltage scaling, the performance improvements can be applied orthogonally to other energy saving methods that work under a down-scaled frequency. As energy is the product of both averaged power and the total execution time, if the utilization of ALU cascading can improve performance without large hardware extensions, the final energy amount can be reduced. Therefore, we use performance margin to study the effectiveness of our instruction scheduler which supports cascading scheduling. In this section, we evaluate possible performance improvements—measured by IPC—by executing a series of benchmarks in the employed simulation environment.

5.1 Simulation Methodology

We used a detailed cycle-accurate out-of-order execution simulator—SimpleScalar Tool Set [13] to measure the possible IPC improvements by the ALU cascading method. **Table 1** lists the configuration information of the baseline processor with an assumed 12-stage pipeline, as shown in **Fig. 8**. In the simulation, we assumed a separate load/store architecture that divides the operation of memory instruction into address generation and memory access internal instructions. Thus, ALU cascading can be applied on both ALU operation instructions (SHIFTs are also included) and address generation instructions derived from memory accesses.

Table 1 Baseline processor configuration.

Processor	8-way out-of-order issue, 64-entry RUU, 32-entry LSQ, 8 int ALU, 4 int mult/div, 8 fp ALU, 4 fp mult/div, 8 memory ports
Branch prediction	10 K-entry bimode [12] (4 K-entry x2 direction PHT, 2 K-entry choice PHT, 12-bit history), 2 K-entry BTB, 16-entry RAS, 10-cycle misprediction penalty
L1 I-cache	64 KB/32 B line/2-way
L1 D-cache	64 KB/32 B line/2-way
L2 unified cache	2 MB/64 B line/4-way
Memory	64 cycles first hit, 2 cycles burst interval
TLB	16-entry I-TLB, 32-entry D-TLB, 80 cycles miss latency

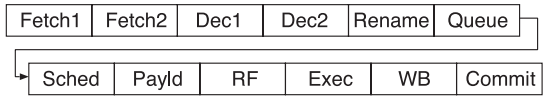


Fig. 8 The assumed 12-stage processor pipeline.

Table 2 lists the benchmarks which we used for evaluation. We chose eight benchmarks from SPECint2000, compiled with gcc Ver. 2.7.2.3 for SimpleScalar PISA. Also, six benchmarks from Mediabench [14] are selected, excluding those too short programs (less than 50 M instructions). The Mediabench programs are executed from beginning to end. For each SPECint2000 benchmark, 1.5 billion instructions are simulated with skipping first 1 billion instructions.

5.2 Performance Improvements

The IPCs of the processor without ALU cascading adoption are collected from the simulation results as a baseline performance measurement, which are listed

Table 2 Benchmark programs.

benchmark		baseline IPC
SPECint2000	bzip2	3.57
	gcc	2.14
	gzip	1.77
	mcf	0.48
	parser	1.56
	perlbmk	1.72
	vortex	3.14
	vpr	1.50
Mediabench	G721 decode	2.30
	G721 encode	2.00
	GSM decode	3.10
	GSM encode	3.85
	MPEG2 decode	2.92
	MPEG2 encode	1.43

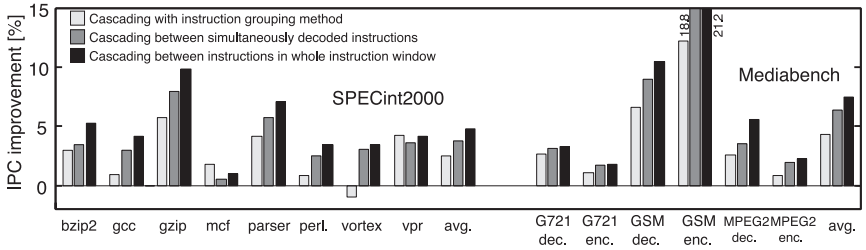


Fig. 9 Normalized IPC improvements after ALU cascading adoption.

in Table 2. All IPC results of other executions are normalized to these values. **Figure 9** shows the IPC improvements by using different instruction scheduling methods for the ALU cascading purpose. In Fig. 9, the vertical axis gives the normalized IPC improvements and the horizontal axis shows three bars of different scheduling policies in each benchmark. The middle bar of each benchmark depicts the IPC improvement with our proposed instruction scheduler that supports ALU cascading among simultaneously decoded instructions. For comparison, the prior instruction grouping method (Section 3) and the method of the extreme ALU cascading within the whole instruction window (Section 4.2) are also demonstrated in Fig. 9, as the left and right bars in each benchmark. Note that the instruction grouping method uses an extended dual-slot instruction window of 64 entries, so that a maximum of 128 instruction slots can be used simultaneously.

The results of the left bars show that the average IPC improvement becomes 2.5% in SPECint2000 and 4.3% in Mediabench with the grouping method. Because of the issue of port restriction described in paper 8), it slightly degrades performance in some high IPC benchmark such as vortex. But it achieves higher improvements compared to our proposal in benchmarks mcf and vpr because it can utilize a maximum of twice as many instructions with a dual-slot instruction window.

With cascading among simultaneously decoded instructions which is our major design introduced in Section 4, the average IPC improvement becomes 3.7% in SPECint2000 and 6.4% in Mediabench, respectively. Our proposal achieved higher average improvements compared to the grouping method even with a smaller instruction window area. In the extreme ALU cascading utilization which applies ALU cascading in the whole instruction window, the broadened cascadable instructions choice provides further performance improvement opportunities. The average improvement becomes 4.8% in SPECint2000 and 7.5% in Mediabench. Studying these values, the improvement ratio in SPECint2000 from middle bar to right bar is larger than in Mediabench, indicating that it might be beneficial to broaden the cascadable instruction searching range. However, compared to the large additional hardware required to implement the extreme ALU cascading, the improvement from the middle to the right bar is relatively insignificant, as per the cost effective consideration. Therefore, ALU cascading among simultaneously decoded instructions is used in our research.

Figure 10 shows the cascading ratios in each scheduling policy. The horizontal axis is the same as in Fig. 9 and the vertical axis now depicts the cascaded ratios. In the figure, the blank bars denote the percentage of ALU instructions in each program execution. After adding address generation instructions to ALU cascading candidates, 50–70% instructions become ALU-like operations, suitable for cascading. The gray bars are the ratios of instructions which are actually cascaded into a latter half of clock cycle execution, after applying the three instruction scheduling methods. As shown in the figure, ALU cascading is comparatively easy in Mediabench, which is also reflected by the larger IPC im-

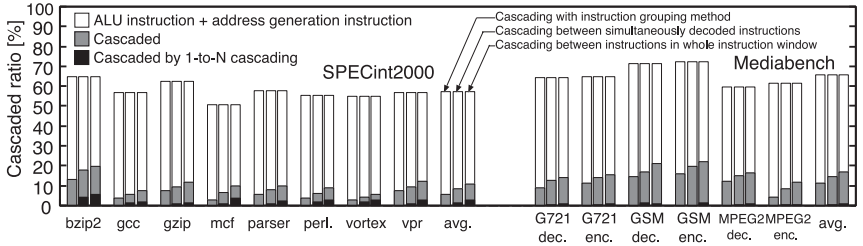


Fig. 10 Ratio of instruction which is executed as a latter instruction of ALU cascading.

provements from Fig. 9. Also, comparing the left and the middle bars in Fig. 10, we can see that our ALU cascading among the simultaneously decoded instructions exploits more cascading opportunities than the grouping method. This is because in the grouping method, only one unresolved dependence is permitted in the consumer instruction before it can be packed with its producer for the cascaded execution. The black bar in each benchmark denotes the percentage of instructions which are executed as consumer instructions in 1-to-N cascading. Note that the grouping method only allows cascading between one producer and one consumer so that there are no 1-to-N cascading results. Different to the gray bars, the adoption of 1-to-N cascading is more frequent in SPECint2000. It gives a vision that SPECint2000 has more complex data dependences, as compared to Mediabench.

However, as shown from Fig. 9 and Fig. 10, there is a weak correlation between IPC improvement and cascaded ratio. By additional analyses, we found that the gain from ALU cascading can be easily overwhelmed if the benchmark experiences many pipeline hazards. For example, the IPC improvement of mcf by cascading is trivial compared to the other benchmarks despite the fact that there is only a small difference in the cascading ratios. The reason is that the processor throughput of mcf is greatly limited by its high cache miss rate characteristics. One main memory access caused by a cache miss may introduce an 80-cycle pipeline hazard. Since the gain from an ALU cascading is only 1 cycle, if the saved cycles from cascaded executions overlap with the long pipeline hazard, the effectiveness of ALU cascading is concealed. Therefore, ALU cascading will become more apparent if combined with other pipeline hazard reducing technologies. If a perfect cache is assumed, the IPC improvement of mcf becomes 3.7%, which is of a similar level as the other benchmarks. Furthermore, if we assume a more ideal processor (perfect caches, perfect branch predictions, a 1024-entry instruction window and a 128-way superscalar processor), the effectiveness of ALU cascading becomes much larger than in Fig. 9. The improvement is 28.4% for SPECint2000 and 33.8% for Mediabench, averaged from the evaluated benchmarks. Detailed results after applying idealized environments are listed in Section A.2.

If the processor employs a narrower decode width compared to the baseline processor in Table 1, the IPC improvement and the cascaded ratio in the middle bar may become smaller because of the reduced cascading candidate searching range. In such cases, we can implement a decoded result buffer which holds the decode result of a prior cycle to compare with a current decode result. By implementing this buffer, we can obtain a doubled comparison width to detect more cascading opportunities. The influences from changing the decode/issue width are briefly studied in Section A.3.

6. Preliminary Implementation and Overhead Estimation

Previous sections have introduced the proposed DMT-based instruction scheduler with ALU cascading supports in detail. The designed wakeup/selection in the scheduling method can help issue the dependent ALU instructions like $i5 \rightarrow i6$ in Fig. 4 in the same cycle. However, we have to prepare additional logics and data paths for the execution of cascaded instructions. This section introduces an implementation candidate of the execution stage that supports cascaded executions. Based on the implementation, the overhead introduced from the added circuits are preliminarily studied.

6.1 Execution Stage with ALU Cascading

After the issue stage, there is no data dependence information among instructions. The execution logics must detect the cascadable pair itself. This step is very similar to the conventional result forwarding, because usual execution logics also need to find out the data dependences between current instruction and instructions in latter stages to choose the proper forwarding path. Therefore, comparators are required to detect the dependences from both instructions in

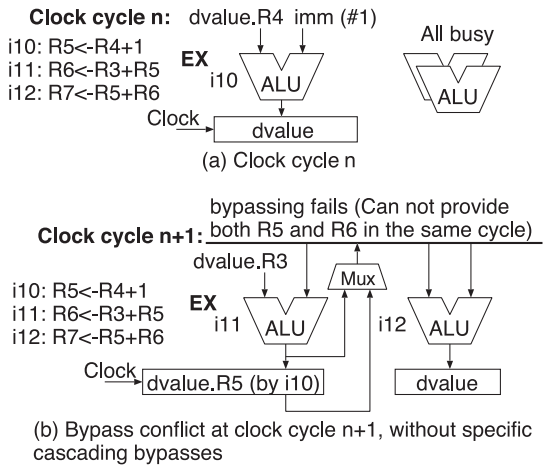


Fig. 11 An example of bypassing conflicts, without specific cascading bypasses.

latter stages and instructions in the same stage, for the purpose of correct source operand selection among both result buses and cascading paths.

Meanwhile, to execute cascadable chain $i5 \rightarrow i6$ (as shown in Fig. 6) in the same cycle, a cascading connection between $i5$'s ALU output to $i6$'s ALU input is required. Basically, this cascading connection may be achieved by adding a bypass to the pipeline register after $i5$'s ALU output, and making use of a bypass network which is originally for data forwarding between different stages. However, it is possible that the bypass route may be simultaneously occupied to forward data from the last to the current execution cycle. As an example, **Fig. 11** uses a simple program block from instruction $i10$ to $i12$ to illustrate this conflict. Specifically, the execution stage units including the ALU and the pipeline register (denoted as *dvalue*) after it are shown in the figure. Assume that only $i10$ is executed in clock cycle n due to the resource limitation, as shown in Fig. 11 (a). Studying the program block, the producer/consumer pair $i11 \rightarrow i12$ can form a cascadable chain. Under the condition of Fig. 11 (a), if no specific cascading bypasses are used, it is impossible to use a single data bypassing route to provide the two input data to the ALU that executes $i12$ in cycle $n + 1$, as depicted in Fig. 11 (b). The cascaded execution of $i11 \rightarrow i12$ will thus fail. To avoid the conflict on that bypass, an arbitration is required to keep correct data passing. The arbitration complicates the design and may also extend the critical path. In view of this consideration, we use specific cascading bypasses in this implementation without violating original forwarding bypasses.

Figure 12 illustrates the organization of register fetch (RF) stage and execution (EX) stage, which has included additional bypassing routes and comparators for ALU cascading. The shaded parts denote newly added hardware to support ALU cascading. These hardware units are separated into comparators, bypassing routes, and multiplexers. The two thick dashed lines (marked as (1) and (2) in Fig. 12) which pass through multiple blocks are the data paths of either cascading or forwarding. The delay of these two paths will be evaluated in Section 6.2. Latches known as pipeline registers are placed between stages. In general, *op* represents the operation code. *dtag* is the destination register number and $stagL/R$ stands for the two source register numbers. All the registers here refer to the physical ones. $valueL/R$ and *imm* show values which are read from the register file or given as an immediate value. The latches $fselL/R$ and shaded $cselL/R$ are control signals for the multiplexers that determine the selection of source operands. $fselL/R$ saves the information if the source operand can come from the result forwarding, and $cselL/R$ contains the information if the source can be provided from ALU cascading. These signals are created by comparators in RF stage. *dvalue* is the produced output of the ALU logics and will be committed into the destination register in the latter writeback phase.

In usual processors, there are no shaded parts and the logic only considers whether the source operand arrives from the register file or one of the *dvalue* latches. The value of $fselL/R$ is given by the comparison result of $stagL/R$ and *dtag* in the latter pipeline stages.

The cascaded execution will be achieved by using the following hardware units. Firstly, we added comparators for $cselL/R$ creation, which compare the *dtag* and *stag* of instructions in the same stage. Secondly, we present an additional bypass network from the outputs of the ALU to the multiplexer which selects the source operand. To reduce additional delay from the load capacitance, we prepare signal drivers to the input of additional bypass network. Thirdly, we prepare two more multiplexers, together with the original forwarding data selection multi-

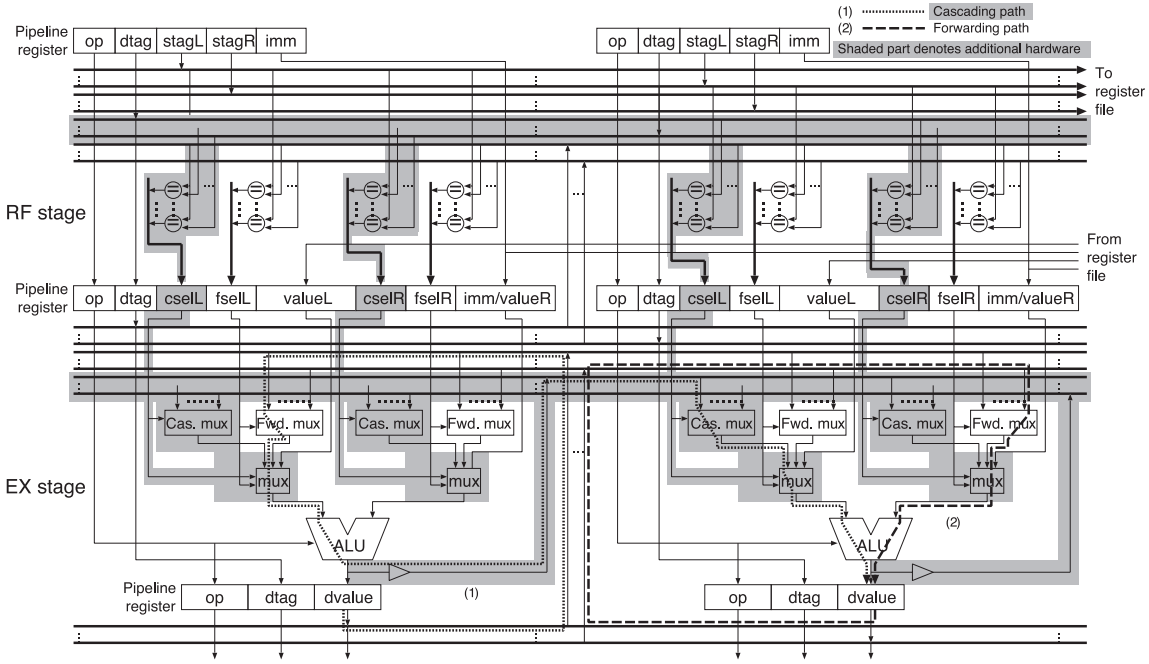


Fig. 12 Register fetch (RF) stage and execution (EX) stage with bypassing routes for ALU cascading.

plexer denoted as *Fwd.Mux*, to construct a two-level hierarchy before each ALU input. The three multiplexers can provide an appropriate selection for the source operand, either from the register file, data forwarding, or the other ALU output.

The added hardware units introduce some delay penalty for EX stage because the additional load for ALU output and the new multiplexers are on the critical path. The evaluation result of the critical path extension is shown in the next section.

6.2 Delay Overhead in the Execution Stage

In this section, we show an estimation result of the delay overhead in the execution stage with the additional logic units for ALU cascading. We designed the circuit listed in Fig. 12 with Verilog HDL. For comparison, original execution logic with only one multiplexer hierarchy before the ALU was also studied. Since ALU cascading in this paper is only applicable for ALU operations with short delays, the ALU used in the logic design was limited to have the functions of NOT, AND, OR, XOR, ADD, SUB and SHIFTs $^{\star 1}$. The data of ALU operations were set to be 32-bit. We evaluated the delay with 8 ALUs and data paths which fully support result forwarding and ALU cascading between all ALUs. Above logics were synthesized with Synopsis Design Compiler under Rohm 0.18 μm cell library. Note that current implementation and overhead studies are still at preliminary stages. Wire delay is not considered in the following estimation. Detailed circuit level designs including wire delay consideration and influences from miniaturized process technology will be one of our major future tasks.

⋆1 SHIFT operations include SHIFT left, arithmetic and logical SHIFT right, and ROTATE.

Table 3 Delay of units in execution stage.

Unit	Delay (FO4)	
	w/o cascading	w cascading
Bypass before ALU	2.91	3.42
32-bit ALU cell	18.83	18.83
Pipeline register	3.69	3.69
Total	25.43	25.94
Normalization	100%	102.0%

Table 3 lists the delay results of the major units in the execution stage under non ALU cascading execution. The latency of each unit is expressed in fan-out-of-four (FO4) inverter delays. Columns 2 and 3 are the delay results without and with ALU cascading logic units, respectively. We divided the data path (as the rough dashed line "(2)" in Fig. 12) in each execution stages into three parts: the result forwarding starts from the pipeline register to the ALU input, the ALU circuit, and the data committing from ALU output into the pipeline register ⋆1. Since ALU and pipeline register remain unmodified whether there is support for ALU cascading or not, the only difference is the bypass route before each ALU, which is either of one-level or two-level multiplexer hierarchy. The delay in Table 3 shows that the two-level multiplexer hierarchy increases a 0.51 FO4 inverter delay for the bypass before each ALU input. However, since the major delay of the execution stage comes from the ALU operations which are not affected by cascading logics, the overhead from two-level multiplexor hierarchies is lessened. The total delay increase for one execution stage is about 2.0%. This increase is expected to be smaller if more complicated ALU operations are used in the real system. Moreover, if this increase can be concealed by the other critical paths, the clock frequency will not be influenced and ALU cascading becomes implementable in that processor.

It is expected that under a current or a future process technology, the wire delay will contribute more to the critical path in the execution stage. As a result, the ratio from the delay of cascading bypass may also have an increasing tendency. However, the major part of the cascading bypass works in parallel to the traditional bypass for data forwarding. If the data forwarding bypass is applicable, the cascading bypass can be similarly implemented.

⋆1 Strictly speaking, the latch overhead of the pipeline register includes the setup and hold delays for the latch unit that cross the clock edge. For simplicity, these two delays are calculated as a whole.

Table 4 Delay of ALU cascading execution.

Unit	Delay (FO4)
Bypass before ALU 1	3.42
32-bit ALU cell 1	18.83
Bypass before ALU 2	3.42
32-bit ALU cell 2	18.83
Pipeline register	3.69
Total	48.19

Table 4 lists the delay of an ALU cascading execution which includes two dependent ALU subtraction operations. The data path is illustrated as the finer dashed line "(1)" in Fig. 12. Supposing that all the prerequisites are fulfilled for the ALU cascading execution, the critical path of the execution stage will start from the result forwarding after the pipeline register, running through two cascaded ALUs via the proper bypassing route, and committing into the pipeline registers. Here, only one latch overhead of the pipeline register is included in the critical path of the ALU cascading. The total delay for the two subtraction operations is 48.19 FO4 inverter delays. This delay is slightly smaller than twice of the clock cycle indicated in Table 3. Therefore, the cascaded execution can be safely finished in half of the frequency, which fits in with our target environment that ALU cascading is applicable under the half of the maximum clock frequency.

7. Related Works

There are some researches on ALU cascading adoptions in the processors excluding dynamic cascading employment for out-of-order issue superscalar processors. There is an adoption example for asynchronous superscalar processor [15),16)]. The processor issues several instructions which are fetched simultaneously without considering data dependences. To resolve the data dependence, the execution stage of the processor checks data dependence and connects an output of ALU into an input of the other ALU. The cycle time of the execution stage is automatically extended because of asynchronous operation. The performance potential of data collapsing, which is a similar idea of ALU cascading, was studied in a superscalar processor based on SPARC V.8 architecture in paper 1). In that

research, the collapsing opportunities were ideally utilized based on tracing data while the dynamic scheduling method was not presented. Also, there is a proposal to adopt ALU cascading to usual superscalar processor [4),17)]. However, detailed instruction scheduling for cascading is not presented.

Researches were carried out in papers 18) and 19) to combine multiple short-delay instructions which may contain true dependences into Macro-OPs (MOP). These schemes illustrate large similarities to the ALU cascading implementation in paper 8), which has been introduced in Section 3. They may have similar problems due to the fixed grouping feature after decoding.

Mini-graph [20),21)] and instruction subgraph [22)] give ideas to shorten critical paths in programs by grouping instructions with dependences based on subgraphs. Compiler supports are required to delineate the subgraphs.

ALU cascading is only adopted under low clock frequency without supply voltage scaling. There are ideas which utilize low clock frequency operation without supply voltage scaling, which are known as dynamic pipeline scaling [23)], pipeline stage unification or PSU [24),25)], or adaptive pipeline depth control [26)]. Those ideas shrink pipeline depth by merging pipeline stages under low clock frequency without the supply voltage scaling. There is a research which proposes to adopt ALU cascading under PSU adoption [27)]. They showed that the effectiveness of PSU could be improved by adding ALU cascading adoption.

There is a proposal which utilizes ALU cascading to alleviate circuit delay variations from process variations [28)]. In that research, they designed methods to compensate performance degradation from variations by ALU cascading, because the deep logic depth usually provides more tolerance for variations.

8. Conclusions

In this paper, we proposed an instruction scheduler for ALU cascading. The designed scheduler can issue producer and consumer instructions simultaneously to achieve ALU cascading in the execution stage. We evaluated the scheduler with SPECint2000 and Mediabench benchmarks. On average, the proposed instruction scheduler achieves a 3.7% and a 6.4% IPC improvement by ALU cascading executions, respectively. It outperforms a prior ALU cascading implementation which uses instruction grouping, by making use of more cascading opportunities even with a smaller hardware cost. Since the energy consumption of the processor is represented by the product result of power and execution time, the IPC improvements will lead to energy reductions if the power overhead from additional hardware is controlled under an acceptable level.

We also evaluated the logic delay caused by additional hardware based on a preliminary implementation. The evaluation result indicates that ALU cascading extends the critical path of the execution stage by 2.0%. If this extension can be concealed by the other critical paths, ALU cascading becomes effective. Detailed circuit level implementation will be studied in future works.

Acknowledgments This work is supported by VLSI Design and Education Center (VDEC), University of Tokyo with the collaboration of the Synopsys Corporation. This work is also supported by JST CREST.

References

1) Sazeides, Y., Vassiliadis, S. and Smith, J.E.: The Performance Potential of Data Dependence Speculation & Collapsing, *Proc. 29th Annual Int. Symp. on Microarchitecture*, pp.238–247 (1996).
2) Sasaki, H., Kondo, M. and Nakamura, H.: Dynamic Instruction Cascading on GALS Microprocessor, *Proc. Int. Workshop on Power and Timing Modeling, Optimization and Simulation 2005* (*PATMOS 2005*), *Lecture Notes in Computer Science 3728*, pp.30–39 (2005).
3) Sasaki, H., Kondo, M. and Nakamura, H.: Dynamic Instruction Cascading on GALS Microprocessors (Japanese), *IPSJ SIG Notes, 2005-ARC-164*, pp.67–72 (2005).
4) Kise, K., Katagiri, T., Honda, H. and Yuba, T.: A Super Instruction-Flow Architecture for High Performance and Low-Power Processors, *Proc. Int. Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems* (*IWIA'04*), pp.10–19 (2004).
5) Goshima, M., Nishino, K., Konishi, M., Nakashima, Y., Mori, S., Kitamura, H. and Tomita, S.: Evaluation of Matrix-based Out-of-Order Scheduling Schemes (Japanese), *IPSJ Trans. High Performance Computing Systems*, Vol.43, No.SIG 6(HPS 5), pp.13–23 (2002).
6) Goshima, M., Nishino, K., Nakashima, Y., Mori, S., Kitamura, T. and Tomita, S.: A High-Speed Dynamic Instruction Scheduling Scheme for SuperScalar Processors, *Proc. 34th Annual Int. Symp. on Microarchitecture*, pp.225–236 (2001).
7) Goshima, M., Nishino, K., Hai-Ha, N., Agata, A., Nakashima, Y., Mori, S., Kitamura, H. and Tomita, S.: A High-Speed Dynamic Instruction Scheduling Scheme for Superscalars (Japanese), *IPSJ Trans. High Performance Computing Systems*,

Vol.42, No.SIG 9(HPS 3), pp.77–92 (2001).
8) Sasaki, H., Kondo, M. and Nakamura, H.: Instruction Grouping: Providing an Efficient Execution Using Dependence Information (Japanese), *IPSJ SIG Notes, 2006-ARC-170*, pp.73–78 (2006).
9) Butler, M. and Patt, Y.: An Investigation of the Performance of Various Dynamic Scheduling Techniques, *SIGMICRO Newsl.*, Vol.23, No.1-2, pp.1–9 (1992).
10) Palacharla, S., Jouppi, N.P. and Smith, J.E.: Complexity-Effective Superscalar Processors, *Proc. 24th Int. Symp. on Computer Architecture*, pp.206–218 (1997).
11) Shivakumar, P. and Jouppi, N.P.: CACTI3.0: An Integrated Cache Timing, Power, and Area Model, Technical report (2001).
12) Lee, C.-C., Chen, I.-C. and Mudge, T.: The Bi-Mode Branch Predictor, *Proc. 30th Annual Int. Symp. on Microarchitecture*, pp.4–13 (1997).
13) Burger, D. and Austin, T.M.: The SimpleScalar Tool Set, Version 2.0, Technical Report CS-TR-97-1342, University of Wisconsin-Madison Computer Sciences Dept. (1997).
14) Lee, C., Potkonjak, M. and Mangione-Smith, W.H.: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems, *Proc. 30th Annual Int. Symp. on Microarchitecture*, pp.330–335 (1997).
15) Ozawa, M., Imai, M., Ueno, Y., Nakamura, H. and Nanya, T.: Performance Evaluation of Cascade ALU Architecture for Asynchronous Super-Scalar Processors, *Proc. 7th Int. Symp. on Asynchronous Circuits and Systems*, pp.162–172 (2001).
16) Ozawa, M., Nakamura, H. and Nanya, T.: Cascade ALU Architecture: Preserving Performance Scalability with Power Consumption Suppressed, *Int. Symp. on Low-Power and High-Speed Chips* (*COOL Chips V*), pp.171–185 (2002).
17) Meng, L. and Oyanagi, S.: Dynamic RENAME Technique and CHAIN Technique in a Superscalar Processor (Japanese), *Proc. H18 IPSJ Kansai Chapter Convention*, pp.207–210 (2006).
18) Kim, I. and Lipasti, M.H.: Macro-op Scheduling: Relaxing Scheduling Loop Constraints, *Proc. 36th Annual Int. Symp. on Microarchitecture*, pp.277–290 (2003).
19) Hu, S., Kim, I., Lipasti, M.H. and Smith, J.E.: An Approach for Implementing Efficient Superscalar CISC Processors, *Proc. 12th Int. Symp. on High-Performance Computer Architecture*, pp.41–52 (2006).
20) Bracy, A., Prahlad, P. and Roth, A.: Dataflow Mini-Graphs: Amplifying Superscalar Capacity and Bandwidth, *Proc. 37th Annual Int. Symp. on Microarchitecture*, pp.18–29 (2004).
21) Bracy, A. and Roth, A.: Serialization-Aware Mini-Graphs: Performance with Fewer Resources, *Proc. 39th Annual Int. Symp. on Microarchitecture*, pp.171–184 (2006).
22) Yehia, S., Clark, N., Mahlke, S. and Flautner, K.: Exploring the Design Space of LUT-based Transparent Accelerators, *Proc. 2005 Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, pp.11–21 (2005).
23) Koppanalil, J., Ramrakhyani, P., Desai, S., Vaidyanathan, A. and Rotenberg, E.: A Case for Dynamic Pipeline Scaling, *Proc. 5th Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pp.1–8 (2002).
24) Shimada, H., Ando, H. and Shimada, T.: Pipeline Stage Unification for Low-Power Consumption, *Int. Symp. on Low-Power and High-Speed Chips* (*COOL Chips V*), pp.194–200 (2002).
25) Shimada, H., Ando, H. and Shimada, T.: Pipeline Stage Unification: A Low-Energy Consumption Technique for Future Mobile Processors, *Proc. Int. Symp. on Low Power Electronics and Design 2003*, pp.326–329 (2003).
26) Efthymiou, A. and Garside, J.D.: Adaptive Pipeline Depth Control for Processor Power-Management, *Proc. Conf. on Computer Design 2002*, pp.454–457 (2002).
27) Ogata, K., Shimada, H., Nakashima, Y., Mori, S. and Tomita, S.: ALU Inlining within Pipeline Stage Unification (Japanese), *Proc. H18 IPSJ Kansai Chapter Convention*, pp.203–206 (2006).
28) Watanabe, S., Hashimoto, M. and Sato, T.: ALU Cascading for Improving Timing Yield (Japanese), *Proc. Annual Symp. on Advanced Computing Systems and Infrastructures SACSIS 2008*, pp.115–122 (2008).
29) Hinton, G., Upton, M., Sager, D., Boggs, D., Carmean, D., Roussel, P., Chappell, T., Fletcher, T., Milshtein, M., Sprague, M., Samaan, S. and Murray, R.: A 0.18-μm CMOS IA-32 Processor with a 4-GHz Integer Execution Unit, *IEEE Journal of Solid-State Circuits*, Vol.36, No.11, pp.1617–1627 (2001).
30) Intel Corporation: Intel Itanium 2 Processor Reference Manual—For Software Development and Optimization, (2004).

Appendix

A.1 Distance between Cascaded Instructions

Figure 13 depicts the distribution of the distance between cascaded instructions in an 8-way superscalar processor, following the whole instruction window search policy which is expressed as the third bar in each benchmark in Fig. 9. The distance results are averaged from benchmarks in either SPECint2000 or Mediabench. The x-axis denotes the distance. At each x-position, the y value along the vertical axis gives the cumulative frequency of 1 to x distances.

It can be observed from Fig. 13 that most of the consumer instructions of the cascaded pairs in Mediabench applications can find their producers in an 8-entry moving window. Note that in an 8-way superscalar processor, the decode width

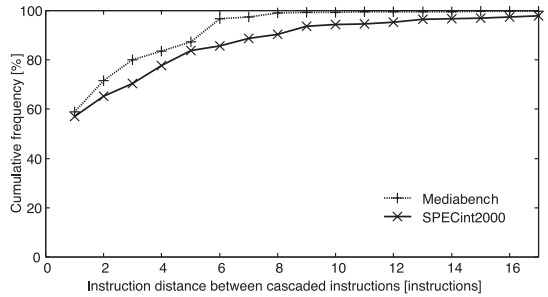


Fig. 13 Distance between instructions with dependences.

is actually up-to-8 ⋆1 so that there is still a small gap between the averaged IPC improvements in Mediabench in Fig. 9. Benchmarks from SPECint2000 are relatively more complex, where the cumulative frequency will not converge near 100% until the distance exceeds 16. This observation can help determine the suitable cascadable instruction search range. For Mediabench applications, it is efficient to search backward to 8 prior instructions while for SPECint2000 programs, a large search window is preferred.

A.2 IPC Improvements under Idealized Environments

Figures 14 and **15** illustrate the possible IPC improvements after including cascaded executions under idealized simulation environments. Two different cascading search policies introduced in previous sections are employed. The two figures have similar formats like Fig. 9.

Figure 14 uses perfect caches in the simulation. The main difference to Fig. 9 is in benchmark mcf which is a memory intensive application. The L2 cache misses in the application will conceal the performance gaining from cascading executions when the miss and the cascading happen in parallel. Using a perfect cache can give a vision of the effectiveness from cascading without the influence from cache misses, as shown in Fig. 14.

Figure 15 applies a more ideal environment with perfect caches, perfect branch predictions, a 1,024-entry instruction window, and a 128-way superscalar processor. It can be observed that the performance margins from cascaded execution are magnified in this idealized environment. After minimizing the performance impediments from other processor limitations, the averaged IPC improvements become 28.4% in SPECint2000 and 33.8% in Mediabench, following the search policy among the whole 1024-entry instruction window. These values indicate the performance increasing potential from ALU cascading.

⋆1 Other than the width, the cache line boundary and the taken branch will also end current decode.

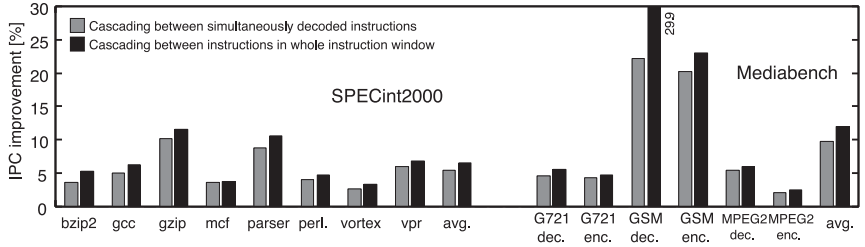


Fig. 14 Normalized IPC improvements after ALU cascading adoption under perfect cache assumption.

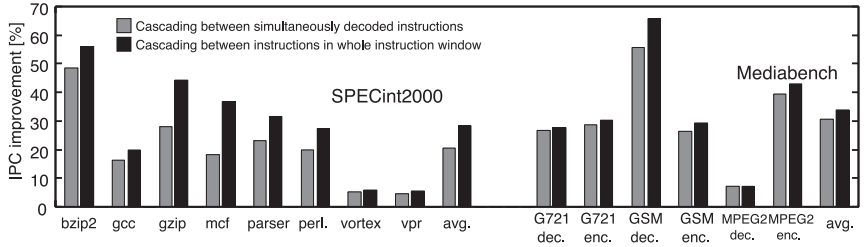


Fig. 15 Normalized IPC improvements after ALU cascading adoption under further perfect environment.

A.3 Influences from Decode and Issue Width

The simulation in Section 5 is based on an 8-way superscalar processor. However, it is possible that reducing the decode/issue width will drawback the efficiency of ALU cascading, especially under the simultaneously decoding search policy.

Figure 16 gives the averaged IPC improvements when the fetch/decode/issue/

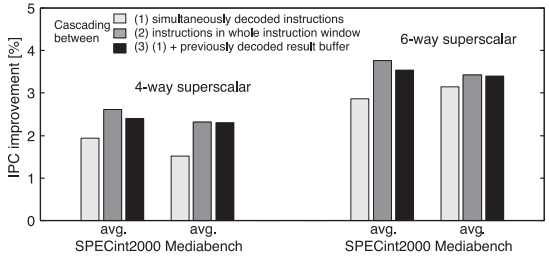


Fig. 16 IPC improvements under 4-way and 6-way processors.

commit widths are reduced to 4 and 6 ⋆1, respectively. In the 4-way superscalar processor, the margins from cascading between whole window cascading to simultaneously decoded instructions are large, as shown in the first and second bars in both the SPECint2000 and Mediabench.

In such cases, we can implement a decoded result buffer which has been introduced in Section 5.2. The buffer holds the decode result of the prior cycle to compare with a current decode result. By implementing this buffer, we can obtain a doubled comparison width to detect more cascading opportunities. The third bar in each benchmark group denotes the IPC improvement from this idea. It adds back the performance improvement toward the value of cascading among the whole instruction window.

The IPC improvements of the 4-way superscalar processor from cascaded execution are smaller than the values of the 8-way superscalar processor in Fig. 9, because of the shortage of free ALU resources to execute cascaded pairs. Since it might be possible to design a complex processor architecture like current Pentium 4 and Itanium 2 under future processor technologies, we employed a relatively heavy processor design in this research.

(Received October 3, 2008)
(Accepted January 19, 2009)

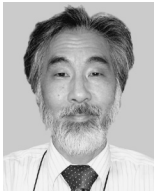
⋆1 Pentium 4 [29] and Itanium 2 [30] employ 6-way superscalar processor architecture.

Jun Yao was born in 1978 and received his B.E. and M.E. degrees from Tsinghua University in 2001 and 2004, respectively. He has been a researcher at the Graduate School of Informatics, Kyoto University since October, 2007. His research interests are computer architecture and storage area networks. He is currently a member of IPSJ and IEICE.

Kosuke Ogata was born in 1983 and received his B.E. and M.E. degrees from Kyoto University in 2006 and 2008 respectively. Since 2008, he has been an employee of Mitsubishi Electric Corporation. His work interest is mainly in Factory Automation technologies.

Hajime Shimada was born in 1976 and received his B.E., M.E. and D.E. degrees from Nagoya University in 1998, 2000 and 2004 respectively. He was a research associate in the Graduate School of Informatics, Kyoto University in 2005 and is now an assistant professor in the same faculty. He is currently focusing on computer architecture related researches. He is a member of IPSJ and IEICE.

Shinobu Miwa was born in 1977 and received his Ph.D. degree from Kyoto University in 2007. He has been an assistant professor in the Graduate School of Engineering, Tokyo University of Agriculture and Technology since 2008. His research interests are computer architecture and neural networks. He is a member of IEICE, IPSJ and JSAI.



Hiroshi Nakashima received his M.E. and Ph.D. from Kyoto University in 1981 and 1991 respectively, and was engaged in research on inference systems with Mitsubishi Electric Corporation from 1981. He became an associate professor at Kyoto University in 1992, a professor at Toyohashi University of Technology in 1997, and a professor at Kyoto University in 2006. His current research interests are the architecture of parallel processing systems and the implementation of programming languages. He received the Motooka Award in 1988 and the Sakai Award in 1993. He is a member of IPSJ, IEEE-CS, ACM, ALP and TUG.

Shinji Tomita was born in 1945 in Japan. He received the B.E., M.E. and D.E. degrees from Kyoto University in 1968, 1970 and 1973 respectively. He is currently Dean of the Graduate School of Informatics, Kyoto University. His major research interests are computer architecture and parallel processing. He is a member of IPSJ, ACM and IEEE.