PAPER

Improvement of Renamed Trace Cache through the Reduction of Dependent Path Length for High Energy Efficiency*

Ryota SHIOYA†a) *and* Hideki ANDO†, *Members*

SUMMARY Out-of-order superscalar processors rename register numbers to remove false dependencies between instructions. A renaming logic for register renaming is a high-cost module in a superscalar processor, and it consumes considerable energy. A renamed trace cache (RTC) was proposed for reducing the energy consumption of a renaming logic. An RTC caches and reuses renamed operands, and thus, register renaming can be omitted on RTC hits. However, conventional RTCs suffer from several performance, energy consumption, and hardware overhead problems. We propose a semi-global renamed trace cache (SGRTC) that caches only renamed operands that are short distance from producers outside traces, and solves the problems of conventional RTCs. Evaluation results show that SGRTC achieves 64% lower energy consumption for renaming with a 0.2% performance overhead as compared to a conventional processor.

key words: *superscalar processor, register renaming, trace cache, energy efficiency*

1. Introduction

Single thread performance remains important even in the era of multi-core processors, and major developers have extended out-of-order superscalar processors to improve this performance. For example, recent commercial processors, such as IBM POWER8 [2], Intel Haswell [3], and ARM Cortex A72 [4], are so large that they can execute eight or more instructions in each cycle. Although the performance level of out-of-order superscalar processors is high, they consume much more energy than do in-order processors, because a large amount of energy is consumed by hardware for out-of-order execution.

In an out-of-order superscalar processor, a *register map table (RMT)* is one of the most energy consuming units. An out-of-order superscalar processor dynamically renames register numbers to remove false dependencies between instructions, and this process is called *register renaming*. An RMT is a table for register renaming, and it holds mapping information from logical to physical register numbers. An RMT in general consists of a heavily multi-ported memory and occupies a large area. For example, the area of the rename logic in Alpha 21464 is larger than that of the L1D cache [5]. Moreover, the number of accesses to an RMT is also large, and consequently, it consumes a considerable amount of energy. As a result, the energy consumption of an RMT is comparable to that of a reservation station [6], and its power density is the fourth highest on chip [7].

A *renamed trace cache (RTC)* was proposed for reducing the energy consumption of an RMT. An RTC extends a trace cache (TC) [8] and caches and reuses renamed instructions. Instructions fetched from an RTC have renamed operands, and thus, register renaming can be omitted, which reduces the energy consumption.

Researchers have proposed two types of RTCs [9], [10], which we refer to as *Local-RTC (LRTC)* and *Global-RTC (GRTC)* for simplicity.

LRTC directly extends a TC and caches renamed source operands that refer to producers in the same trace**. LRTC renames the logical register numbers of source operands to *instruction positions* in a trace. When an instruction and its producer are in the same trace, the position of the producer in the trace is constant, and consequently, LRTC can reuse the renamed source operands. The problem of LRTC is that it cannot reuse renamed operands that refer to producers outside a trace. As a result, its advantage is limited, and it can cache renamed operands only for approximately 30% of operands (Sect. 6).

GRTC renames source operands to *displacements* between instructions and caches them according to each past executed path. This scheme allows GRTC to cache renamed operands outside a trace, and thus, it does not have the limitation of LRTC in terms of caching renamed operands.

However, GRTC significantly reduces the capacity efficiency. A TC suffers from a similar problem, but that of GRTC is more serious. In a TC, there are multiple traces that start from the same basic block, and consequently, its capacity efficiency is lower than that of a general instruction cache. Additionally, in GRTC, there are multiple renamed traces depending on *each backward executed path outside a trace*. The number of renamed traces is increased exponentially with the length of a backward executed path, and thus, the capacity efficiency is significantly reduced in GRTC. Moreover, in GRTC, it is necessary to store the target addresses of indirect jumps to its tag array for checking a path, and it considerably increases the circuit area and energy consumption. As a result, its energy overhead mostly cancels most of the reduced energy consumption of the RMT, as described in Sect. 6.

Manuscript received July 10, 2015.
Manuscript revised October 16, 2015.
Manuscript publicized December 4, 2015.

†The authors are with Graduate School of Engineering, Nagoya University, Nagoya-shi, 464–8603 Japan.

*This paper is an extended version of [1], containing a more detailed description and analysis.

a) E-mail: shioya@nuee.nagoya-u.ac.jp

DOI: 10.1587/transinf.2015EDP7270

**The term *trace* means instructions ordered by a dynamic execution sequence.

We propose *semi-global renamed trace cache (SGRTC)*, which is based on GRTC and solves the problems of the conventional GRTC. SGRTC restricts caching renamed operands to those having a short off-trace distance to their producer. The number of renamed traces is increased exponentially with the distance, and thus, this restriction makes it possible to reduce the number of generated traces, and this significantly improves the capacity efficiency. In addition, SGRTC also solves the problem of GRTC that it needs to store the target address of indirect jumps. SGRTC restricts the off-trace distance of a backward executed path, and thus, indirect jumps rarely exist in a backward executed path. As a result, target address storages can be omitted without significant performance degradation, and energy consumption is thus reduced.

The evaluation results presented in Sect. 6 show that the energy consumption for register renaming is reduced by 64% as compared to a processor with a conventional instruction cache, and only a 0.2% performance degradation is incurred. Further, the evaluation results show that the proposed method achieves an 8.5% and 4.5% higher performance, 23% and 60% lower energy consumption, and 41% and 151% higher performance-energy ratio (the inverse of the energy-delay product (EDP)) than conventional LRTC and GRTC, respectively.

Researchers of conventional LRTC and GRTC previously stated that the complexity of a rename logic can be mitigated; however, they did not evaluate energy consumption quantitatively. One of the contributions of this paper is a quantitative evaluation of the energy consumption and performance of these conventional methods.

The rest of the paper is organized as follows. In Sect. 2, a description of an RMT and the caching of renamed operands provides background knowledge. In Sects. 3 and 4, GRTC and its problems are described, respectively. In Sect. 5, our proposed method, SGRTC, is described. In Sect. 6, our evaluation results are presented. In Sect. 7, related work is summarized.

2. Background

In this section, we first describe the problems related to an RMT, and then, we briefly explain why it is in general difficult to cache renamed operands and how the manner in which LRTC overcomes this difficulty.

2.1 Problems of Register Map Table

Register numbers are renamed by reading and writing an RMT. An RMT is a table containing the relationship information between the logical and physical register numbers; in general, it comprises multi-ported RAM [11], [12]†.

An RMT occupies a considerably large circuit area. For general ISAs with a three-operand format, an RMT requires four ports per instruction [11], [13]. For example, in a processor having a decode width of eight, such as IBM POWER 8 [2], a straightforward implementation of an RMT requires 32 ports. The area of a RAM grows proportionally with the square of the number of ports [14], and therefore, the area of the RMT is large despite its small number of entries.

Moreover, spreading SMT processors makes the RMT considerably larger. SMT processors require RMTs with capacities that are proportional to the number of threads for retaining the thread contexts. In fact, Alpha 21464 with 4-thread SMT has a rename logic that is larger than its L1-data cache [15]. The recent IBM POWER 8 [2] supports 8-thread SMT and this problem becomes more severe.

As a result, the energy consumption and generated heat of the RMT is a very important issue. The energy consumption of a RAM is proportional to its area and access frequency [14]. Almost all instructions access the RMT more than once, and thus, the number of accesses to it is considerably high. Consequently, the RMT consumes significantly more energy than a data cache in the same area.

In fact, the RMT in the Intel P6 architecture accounts for 4% of the energy consumed by the processor, which is comparable to that of its reservation station [6]. The RMT in the ARM Cortex A15 accounts for 6% of the energy consumption of the processor when its loop cache is disabled [16]. When the loop cache is enabled and operates effectively, the proportion of energy consumed by the RMT is practically increased, because the loop cache can omit institution fetch, decode, and branch prediction, which accounts for 40% of the total consumed energy. In recent processors, such as IBM POWER 8, the rename width is significantly wider than those of the Intel P6 and ARM Cortex A15, and since they support SMT, their RMT is also significantly larger.

2.2 Caching Renamed Operands

As a means of mitigating the problems of an RMT, caching and reusing renamed operands is an attractive idea; however, in general, this approach is difficult for the following reasons;

1. The producers of source operands are changed depending on the executed paths, and consequently, the physical register numbers of source operands are not reusable.
2. Physical registers are assigned from a free list, and consequently, assigned register numbers are not reproducible and reusable.

We show an example in Fig. 1. This figure shows the control flow graph, and each node represents an instruction. In this figure, I_{t2} depends on r1 and r2. The physical register numbers assigned for each operand are denoted by [pN], for example, [p8]. The first operand r1 can be produced by I_{p0}, I_{p1}, or I_{p2} depending on its executed path, and consequently, the physical register number can be changed and is not reusable. The second operand r2 is always produced by

†There is a method with a CAM-based RMT [5], and our proposal also can be implemented on a CAM-based method.

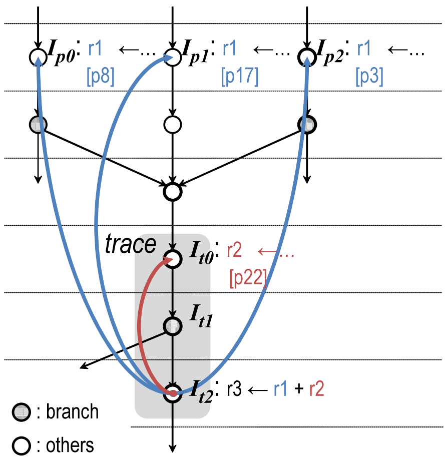

Fig. 1 Non-reusability of renamed operands

I_{t0}, but its physical register number is assigned from a free list and is not reproducible.

2.3 Local Renamed Trace Cache

LRTC, described in Sect. 1, avoids the above problems and partially reuses renamed operands by using the following methods. LRTC uses a physical register file (RF) that is local for each trace, in addition to a conventional physical RF. Each entry of this local physical RF is sequentially assigned to each instruction in a trace, and is referred to with an offset from the start of a trace[†]. In a trace, it is guaranteed that the positions of each instruction and its execution paths are set according to the behavior of TC. Consequently, LRTC can resolve the above two problems and reuse renamed operands in the trace.

We show an example in Fig. 1. In this figure, I_{t0}, I_{t2}, and I_{t3} are fetched as a single trace. I_{t0} is the producer of the source operand r2 of I_{t3}, and they are in the same trace. In this case, the r2 of I_{t3} is renamed to the position of I_{t0} (0) in the trace. In the trace, it is guaranteed that the position of I_{t0} and the execution path from I_{t0} to I_{t3} are set according to the behavior of TC, and thus, a renamed operand can be reused.

However, this means that LRTC can cache renamed operands only when source operands refer to destination operands generated in the same trace. For example, in Fig. 1, the source operand r1 of I_{t3} refers to the producers outside the trace, and its producer is changed depending on the executed path outside the trace. Consequently, its renamed operand cannot be reused and it is renamed as a conventional processor. As a result, LRTC cannot operate effectively and can cache only approximately 30% of renamed operands (Sect. 6).

3. Global Renamed Trace Cache

Unlike LRTC, GRTC was proposed for caching renamed

[†]There is a more optimized implementation, but we describe this implementation for simplicity.

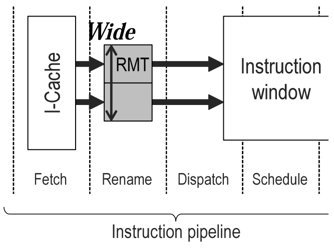

Fig. 2 General front-end

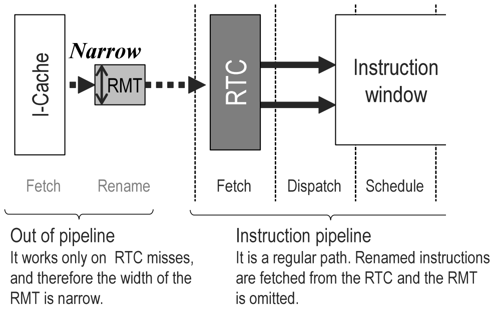
I-Cache
Narrow
RMT
RTC
Instruction window
Fetch
Rename
Fetch
Dispatch
Schedule
Out of pipeline
It works only on RTC misses, and therefore the width of the RMT is narrow.
Instruction pipeline
It is a regular path. Renamed instructions are fetched from the RTC and the RMT is omitted.

Fig. 3 GRTC's front-end

operands that refer to producers outside a trace [10]. Figures 2 and 3 show the front-end of a conventional superscalar processor and that with GRTC, respectively. A conventional superscalar processor renames operands for each fetched instruction. On the other hand, in GRTC, fetched instructions are directly dispatched to the instruction window, because their operands are already renamed. Operands are renamed only when traces are generated on an RTC miss, and consequently, the number of ports of an RMT can be reduced without performance degradation. This reduction of ports considerably reduces the RMT's area and energy consumption.

GRTC modifies the allocation and reference of physical RFs, and allows the reuse of renamed operands without the restriction of caching in LRTC. This section describes GRTC.

3.1 Register File Method

GRTC uses the two RFs, the physical register file (PRF) and logical register file (LRF). The entries in these RFs are allocated in the same manner as a reorder buffer and LRF in a processor with a reservation station. However, only the allocation method is the same. The issue queue in GRTC does not have values and the RFs are accessed only after issuing from the issue queue. In particular, the PRF has a ring structure and its entries are sequentially allocated. The LRF holds the execution results of retired instructions. Instructions write their execution results to the LRF in sequential order.

3.2 Instruction Method

In GRTC, source operands are referred to with the *displace-*

```
I0 :  ld    r1 ← …
I1 :  bgt   r1 > 0 then I3
I2 :  neg   r1 ← - r1
I3 :  add   r2 ← r1  + …
I4 :  …
```

(a) Original instructions

```
I0 :  ld    r1 ← …
I1 :  bgt   [-1] > 0 then I3
I3 :  add   r2 ← [-2] + …
I4 :  …              trace Tc
```

(b) I_1 is taken

```
I0 :  ld    r1 ← …
I1 :  bgt   [-1] > 0 then I3
I2 :  neg   r1 ← - [-2]
I3 :  add   r2 ← [-1] + …
I4 :  …              trace Tb
```

(c) I_1 is not taken

Fig. 4 Example of conversion to dualflow ops

ment between instructions as "the execution result before n instructions." This displacement is equal to that between entries allocated to instructions in the PRF, because the entries of the PRF are sequentially allocated to instructions. Each instruction obtains the physical register number of the source operands by adding its displacement to the physical register number of its own destination operand. When a producer is at a greater distance than WS, which is the size of an instruction window[††], it is guaranteed that the producer is retired and its execution result is obtained from the LRF.

An instruction with this displacement is called a *dualflow op*. GRTC dynamically converts instructions to dualflow ops for generating traces on an RTC miss. This conversion is performed with an RMT that is similar to that in a usual rename logic [10].

3.3 Caching Dualflow Ops

GRTC generates traces including dualflow ops for each execution path, and separately caches them. This caching uses a structure that is similar to a conventional TC. The conventional TC generates and caches traces for each execution path *in a trace*. On the other hand, GRTC generates and caches traces for each backward execution path *outside a trace*, in addition to those in a trace.

We explain this scheme by using the example presented in Fig. 4. In this figure, instructions in (a) are converted to dualflow ops in (b) when the branch I_1 is taken, and those in (c) are converted vice versa. I_3 in (b) and (c) has a different displacement, -2 and -1, respectively. The displacements differ according to the backward execution path of I_3. GRTC generates traces T_b and T_c for each pattern and caches them for each backward execution path.

GRTC distinguishes traces by using *path information* that is similar to that used in a conventional TC. The term "path information" is specifically the directions of branches and the target addresses of indirect jumps. GRTC stores this path information and the start address of the path to a tag array. When GRTC determines an RTC hit or miss, it generates path information from the PC and branch history information and compares it with the path information stored in a tag array.

3.4 Reusing Renamed Operands

GRTC resolves the two problems that prevent the reuse of renamed operands (Sect. 2.3). First, the non-reproducibility of physical register numbers is solved by the sequential assignment of registers and operand access with displacement. The displacement, which is a static distance between instructions, is always constant. Consequently, the displacement, once converted, can be reused. Second, the change in the producer for each path is solved by caching traces for each backward execution path. As a result, GRTC can cache and reuse renamed operands.

4. Problems of Global Renamed Trace Cache

This section describes two problems of GRTC.

4.1 Low Capacity Efficiency

The first problem of GRTC is its low capacity efficiency. As compared to TC and LRTC, each trace depends on its backward path outside a trace, and it is necessary to cache different traces starting from the same basic block for each backward path. The increase in the length of a dependent path exponentially increases the number of generated traces, and the capacity efficiency is considerably reduced.

We explain this problem by using Fig. 5. This figure shows the control flow graph, where each node represents an instruction. In the following, we explain the fetch of the trace that starts from I_{t_start} to I_{t_end} by comparing TC and GRTC. Note that we do not explain a case of LRTC, because the fetch methods of LRTC and TC are identical.

(1) TC: A trace hits if the path from I_{t_start} to I_{t_end} in the trace matches the path information in the tag array. In this case, the length of path information is three instructions, which is the length of a trace.

(2) GRTC: In addition to the path in a trace, GRTC requires that the backward execution the path of a trace and path information in the tag array match. The length of the backward path of a trace is determined by the largest displacement in the source operands in a trace (we refer to it as the *dependent path length*). For example, when the source operands of I_{t_start}, I_{t_br}, and I_{t_end} refer the destination operand of I_{p1}, the length of the path information is six instructions, three instructions in the trace and three instructions in the backward path from I_{p1} to I_{t_start}. This is longer than that of TC, which is three instructions.

The maximum dependent path length in GRTC is the size of the instruction window, WS. This is because producers that are distant more than WS must be retired, and it is guaranteed that their results are in the LRF, as described

[††] WS is the size of a reorder buffer.

in Sect. 3.1.

Because of the difference in dependent path lengths, the numbers of generated traces in TC and GRTC are considerably different. For example, in TC the length of a trace of which is three instructions, the number of generated traces that started from the same address is $2^3 = 8$ in the worst case when all the instructions in a trace are branches. On the other hand, in GRTC, each trace depends on its backward path outside a trace, in addition to the inside of a trace. The maximum dependent path length is WS, which is more than 200 instructions in recent processors [2], and consequently, the number of generated traces is more than 2^{200} in the worst case. As described above, the number of generated traces is considerably higher and the capacity efficiency of GRTC is consequently decreased.

4.2 Overhead of Tag

The second problem is the hardware overhead of the stored path information in a tag;

(1) Increased Information for Branch Directions: The path information of GRTC is similar to the information of branch directions in a trace in TC. As described previously, the dependent path length of GRTC is considerably longer than that of TC, and consequently, more hardware is necessary to store it.

(2) Stored Target Address of Indirect Jump: In TC, the target address of an indirect jump is not stored to its tag array in order to avoid hardware overhead. Consequently, a trace is complete if an indirect jump exists in a case of TC. This does not decrease the performance significantly, because the appearance frequency of indirect jumps is sufficiently low.

On the other hand, in GRTC, if the target address of an indirect jump is not stored to a tag array, its performance is degraded sufficiently [10]. This is because the hit/miss determination in GRTC depends on its backward path outside a trace. When the target addresses are not stored to a tag array, it is unable to cache a trace with indirect jumps that are placed backward within the dependent path length of the trace, because matching its path is not guaranteed. For example, in Fig. 5, if I_{p_br} is an indirect jump, instructions above I_{p1} cannot be cached. As a result, when an indirect jump appears, it is unable to cache a considerable number of instructions in front of the indirect jump.

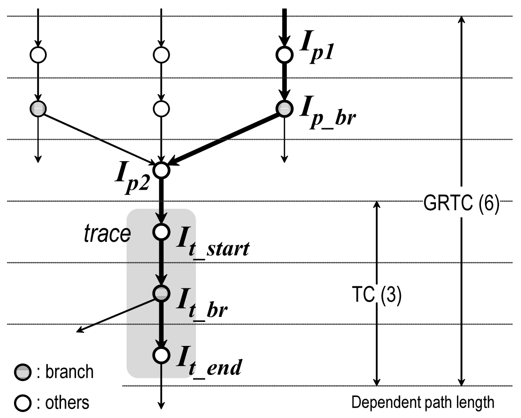

Fig. 5 Control flow and path information

Thus, GRTC stores some target addresses of indirect jumps to a tag array. If the number of indirect jumps is greater than the maximum number, traces are not cached and are renamed with an RMT. Ichibayashi et al. explained that it requires two-target addresses in a tag array for avoiding performance degradation; however, they did not evaluate its overhead such as energy consumption [10].

However, the overhead of its additional hardware is considerably large. For example, when an instruction length is four bytes and a trace length is four instructions, the required capacity for the data part of a trace is 16 bytes. On the other hand, when the length of a target address is 64 bits, the tag part with two target addresses for the path information consumes capacity greater than its data part. Moreover, it requires WS directions of branches, and consequently, the capacity of the tag array is considerably larger than that of a conventional cache and TC.

5. Semi-Global Renamed Trace Cache

We propose *semi-global renamed trace cache (SGRTC)*, which is based on GRTC. In our proposed method, only renamed operands that are a short distance from the producers outside traces are cached.

5.1 Motivation

Many source operands of consumers in general refer to producers near the consumers. Figure 6 shows the distribution of the distances from each source operand to its producer in SPECCPU INT 2006 benchmark suite [17]†. The horizontal axis shows the distance from each source operand to its producer, and the vertical axis shows the cumulative ratio to all source operands. For example, the point plotted at 10 on the horizontal axis and 0.7 on the vertical axis represents that source operands having a distance to their producers of less than 10 instructions occupy 70% of all source operands. This preliminary evaluation result shows that about 70% of

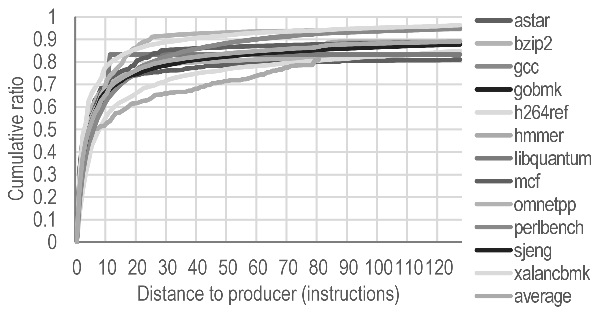

Fig. 6 Distribution of distances to producers

†Evaluation environment is the same as that used in Sect. 6.

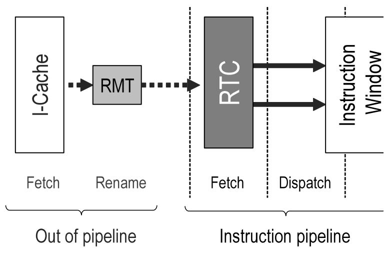

Fig. 7 GRTC's front-end

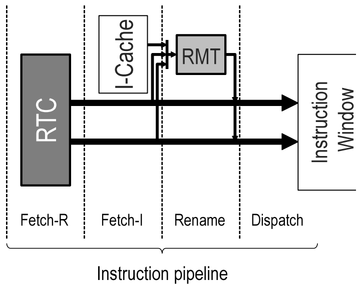

Fig. 8 SGRTC's front-end

source operands refer to producers within 10 instructions in SPECCPU INT 2006.

On the other hand, the dependent path length of a trace (Sect. 4.1) is considerably longer than the distance from each source operand to its producer. This is because the dependent path length of a trace is determined by the largest distance from a source operand to a producer in the trace. As a result, although the distance from most source operands to those producers is within 10 instructions, the average dependent path length is over 80 instructions in GRTC.

5.2 Limiting Caching by Dependent Path Length

We focus on the fact that many source operands generally refer to producers near to the consumers, and propose a method that caches only renamed operands that are a short distance from the producers outside traces. The other operands are stored to an RTC as logical register numbers and are renamed with an RMT after instruction decoding.

We describe our proposal, SGRTC, by comparing the conventional GRTC and SGRTC. Figures 7 and 8 show the front-end pipeline of GRTC and SGRTC, respectively. In GRTC, on an RTC hit, fetched traces are immediately dispatched to the instruction window. On an RTC miss, instructions are fetched from an instruction cache, and then, they are renamed. The stages accessing an instruction cache and an RMT are placed outside the instruction pipeline and operate only on an RTC miss. On the other hand, SGRTC has stages that can access an instruction cache and an RMT, regardless of the RTC hit/miss, and these stages operate as follows.

(1) RTC Hit: In Fig. 8, the successive Fetch-I stage does *not* access the instruction cache. The successive Rename stage renames only operands stored in traces as logical register numbers.

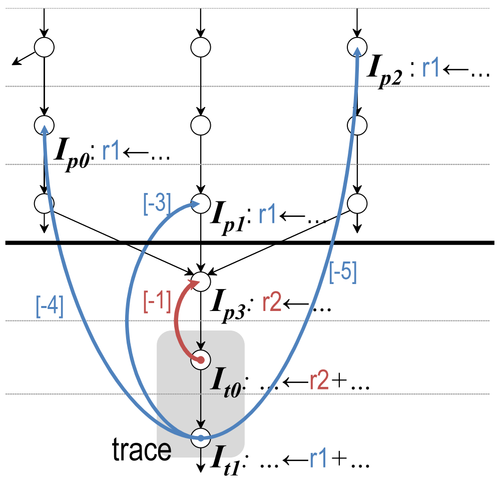

Fig. 9 Reference distance for each instruction

(2) RTC Miss: The Fetch-I stage accesses the instruction cache, and the Rename stage renames operands.

The traces in RTC are replaced and filled in the same way as in a conventional TC [8]. As described above, on an RTC miss, instructions are fetched from the instruction cache and renamed. Then, the renamed instructions are dispatched to the instruction window, and at the same time, fed to a fill unit with path information. The fill unit generates trace data from the fed instructions and path information, and it is filled to RTC. In this time, a replacement target is determined using the LRU policy in the same way as conventional caches.

SGRTC can reduce the number of ports of its RMT as can the conventional GRTC. When the number of ports is small, the front-end is stalled, and the operands in a fetch group are renamed in multi-cycles.

5.3 Effects of Proposed Method

Although our proposed method is very simple, it can effectively solve the two problems of the conventional GRTC described in Sect. 4 and can cache renamed operands more frequently than LRTC does.

5.3.1 Improving Capacity Efficiency

Our proposed method limits the dependent path lengths of traces to short distances and improves the capacity efficiency of RTC. We explain this improvement by using the example shown in Fig. 9, which is a control flow graph similar to that in Fig. 5. In this figure, each instruction has only one source operand. We focus on the trace with I_{t0} and I_{t1}; and there are three backward paths that arrive in this trace.

In general, when the distance from a consumer to a producer increases, the number of paths to its producer also increases. The producer of I_{t0} is near to I_{t0}, and is always I_{p3}, regardless of which backward path is executed. On the other hand, the producers of I_{t1} are different for each backward path, and they are I_{p0}, I_{p1}, and I_{p2}.

In conventional GRTC, the dependent path length is determined by the *largest* displacement in the source operands in a trace. Consequently, different traces are generated for every three paths arriving at I_{t0}.

On the other hand, we assume a case of SGRTC that limits the dependent path length to one instruction. The producer of I_{t0} is within one instruction from I_{t0}, and consequently, its renamed operand is cached. On the other hand, the renamed operands of all the producers of I_{t1} that are more than one instruction distant from I_{t1} are not cached. That is, unrenamed r1 is cached to RTC as it is, regardless of which backward path is executed, and then it is renamed even if RTC hits.

In this case, there is only one path that arrives at I_{t0} from one previous instruction, and consequently, the number of traces that may be generated is one. The number of generated traces is reduced, and this improves its capacity efficiency.

As described in Sect. 4, an increase in the dependent path lengths of traces exponentially increases the number of generated traces, and the capacity efficiency of RTC is considerably reduced. Our proposed method can shorten the dependent path lengths of traces from over 200 instructions to approximately 12 instructions (Sect. 6), and consequently, the number of generated traces is considerably reduced, which significantly improves its capacity efficiency.

Renaming operands stored as logical register numbers does not incur significant overheads. As described previously, most source operands refer to producers within 10 instructions, and therefore, the number of source operands stored as logical register numbers is small. Consequently, these source operands can be renamed by a small RMT that was originally used for renaming when traces are generated on RTC misses in GRTC, and its energy consumption is small.

5.3.2 Reducing Hardware Overhead of Tag

Our proposed method can omit the target address storage of indirect jumps to the tag array without significant performance degradation. As described in Sect. 4.2, the conventional GRTC requires target addresses in a tag array to avoid significant performance degradation. On the other hand, our proposed method can shorten the dependent path lengths of traces to approximately 10 instructions. Consequently, if the target addresses are not stored to the tag array, and SGRTC cannot cache traces with indirect jumps that are placed backward within the dependent path lengths of the traces, the number of non-cachable instructions is not more than 10.

5.3.3 Penalty of Prolonged Pipeline

The pipeline length of SGRTC is slightly longer than that of GRTC, but this does not cause serious problems. The pipeline of SGRTC presented in Fig. 8 can be optimized as follows; 1) Only the tag of RTC is accessed in the Fetch-R stage for hit/miss detection, and then, 2) the trace-data of RTC is accessed in the Fetch-I stage in parallel with the access of the instruction cache. As a result, the Fetch-R stage can be implemented as a single stage, because only the tag is accessed, and consequently, the pipeline length of SGRTC is slightly longer than that of GRTC by one stage. The prolonged pipeline in general increases the misprediction penalty, but the IPC degradation for a single stage increase is slight [18]–[20], no more than 2% even in the worst cases [20]. In addition, the RMT of SGRTC is much smaller than that in conventional processors, and consequently, the pipeline length of the rename stage can be shortened and this cancels the prolonged fetch stages. The evaluation results presented in Sect. 6 show that the performance degradation of the prolonged pipeline is negligible.

6. Evaluation

We evaluated SGRTC and other methods, and the results are presented in this section.

6.1 Evaluation Environment

We evaluated the IPCs using Onikiri2 cycle-accurate processor simulator [21]. We used all the programs from the SPECCPU 2006 benchmark suites [17] with *ref* data sets. The programs were compiled using gcc 4.2.2 with the "-O3" option. We skipped the first 10 G instructions and evaluated the next 100 M instructions.

We evaluated the energy consumption and the area required for register renaming by evaluating the arrays and their peripheral circuits related to our proposed method, which are the RMT, TC, RTC and instruction cache. We did not evaluate the energy consumption of the comparators for dependency checking on register renaming [11], [13], but

Table 1 Baseline configuration

fetch width	8 inst.
rename width	8 inst.
issue width	8 inst.
issue queue	64 entries, unified
execution unit	int:4, fp:4, mem:4
ROB	224 entries
branch predictor	8 KB g-share, 2K entries BTB
SMT	8 threads
L1DC	64 KB, 8 way, 64 B/line, 2 cycles
L1IC	32 KB (34.3 KB†), 8 way, 64 B/line, 2 cycles
L2C	512 KB, 8 way, 64 B/line, 8 cycles
L3C	8 MB, 8 way, 64 B/line, 24 cycles
main memory	200 cycles
ISA	Alpha

Table 2 Configurations of TC and RTC

TC	1 K traces (37 KB†), 8 way, trace:8 inst., 3 cycles
LOCAL	1 K traces (39 KB†), 8 way, trace:8 inst., 3 cycles
GLOBAL	1 K traces (82.9 KB†), 8 way, trace:8 inst., 3 cycles
S_GLOBAL	1 K traces (39.4 KB†), 8 way, trace:8 inst., 3 cycles

†These capacities include the tag parts.

each comparator consists of a few transistors, and thus, we considered their energy consumption to be comparatively small. Moreover, our proposed method omits this dependency checking on an RTC hit, which results in a conservative energy comparison for our technique.

The energy consumption for register renaming was evaluated by using CACTI 6.5 [22], which simulates cell arrays and peripheral circuits. We assumed 32 nm technology (ITRS-HP/nominal), 4 GHz operating frequency, 0.9 V VDD, and 320 K junction temperature. We also assumed that each cell consisted of a regular 4 T storage and two access transistors per port. The arrays were not banked[††]. The static and dynamic access energy of the arrays was calculated by CACTI under the stated PVT assumptions. The temperature was assumed to be fixed and leakage is thus modeled as a constant. The number of accesses to each array structure and the elapsed time is provided by our cycle-accurate processor simulator. The dynamic array access energy from CACTI was multiplied by the array access frequency from the simulator to form the total dynamic energy.

In addition, we evaluated the energy efficiency impact of our proposed method on an entire processor by using McPAT [24]. The processor and device configurations used in this evaluation were the same as those used in the other evaluations.

6.2 Evaluated Models

Table 1 lists the configuration of a baseline processor. Its major micro-architectural parameters are based on those of IBM POWER 8 [2], which include parameters such as fetch width, the size of the instruction window, the number of FUs, and cache hierarchies. Note that we used the configuration based on the IBM POWER 8 with a wide front-end, because it is well known that the influence of the throughput of a processor front-end on its performance is significant [8], [25], [26], and the front-end width of recent high-performance commercial processors is increased [2], [3], [27]. Moreover, recent high-performance processors in general are equipped SMT [2], [3], [27], [28], and thus, the area and energy consumption of their RMTs are considerable (Sect. 2). Our proposed method allows the problems of such high-performance processors to be resolved.

We evaluated the following models based on baseline configuration:

(1) BASE: A model with an I-cache only.
(2) TC: A model with TC.
(3) LOCAL: A model with LRTC.
(4) GLOBAL: A model with GRTC.
(5) S_GLOBAL: A model with SGRTC, which is our proposed method.

Table 2 summarizes the configurations of TC and RTC used in these models. The capacity of RTC in S_GLOBAL was set to 1 K traces, which has a similar area to that of the conventional I-cache. In each model, each trace includes two branches. The TC and RTC in the other models have the same capacity. The hit/miss of TC and RTC is determined in the first cycle, and then, the I-cache is accessed on TC/RTC misses. In GLOBAL, the number of indirect jumps stored to a tag array is two (Sect. 4.2).

The RMT in BASE and TC has 32 ports in total. An RMT generally requires one write and three read ports per single 2-source operand instruction [11], [13]: 1) one write port for updating new destination mapping, 2) two read port for reading source operand mapping, and 3) one read port for reading old destination mapping. As a result, each instruction with 2-source operand requires four ports. We assume that the rename-width of the processor is 8, as presented in Table 1, and thus, the RMT requires $4 \times 8 = 32$ ports in total. In the other models, the number of the read ports of the RMT is reduced. The specific parameters of the read ports are determined by the evaluation presented in the next section.

6.3 Investigating Configurations

First, to determine the configuration of S_GLOBAL, we evaluated its performance while varying its dependent path length and the number of read ports of its RMT. Figure 11 shows the IPCs of S_GLOBAL relative to BASE. These IPCs are on the geometric mean of all benchmark programs. Each line labeled "RMT-*N*p" shows the IPCs of a configuration with an RMT that has *N* read ports. Figure 11 shows that the IPCs are degraded if the number of read ports of the RMT is less than four, and in the configuration with an RMT with four read ports. The configuration having a dependent path length of 12 instructions has the highest performance. Thus, hereafter, we evaluate S_GLOBAL with a four read port RMT and 12-instruction dependent path length.

6.4 Performance

Figure 10 shows the IPCs for each model relative to BASE. S_GLOBAL degrades the IPC of BASE by only 0.2% on a geometric mean. As described in Sect. 5.3.1, our proposed method improves the capacity efficiency, and consequently, its performance penalty, described in Sect. 4.1, is considerably reduced, and is canceled out by the improved fetch throughput in the same manner as TC. On the other hand, S_GLOBAL degrades the IPC of TC by 3.7% on the geometric mean, because TC also improves the fetch throughput.

LOCAL degrades the IPC of BASE by 8.5% on the geometric mean, and its IPC degradation is considerably larger than that of S_GLOBAL. This is because LOCAL can cache only 31.7% of renamed operands on average, as shown in Table 3, and this ratio is about half of that in S_GLOBAL. As a result, the shortage of the RMT ports stalls the front-end, which degrades its performance.

GLOBAL also degrades the IPC of BASE by 4.5% on the geometric mean. This is because of its low capacity efficiency described in Sect. 4.1.

[††]Details are described in [22], [23].

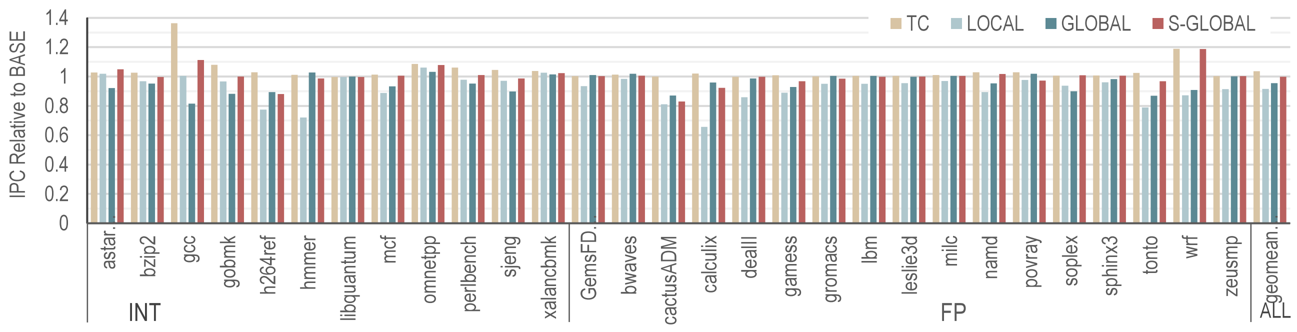

Fig. 10 IPC relative to BASE

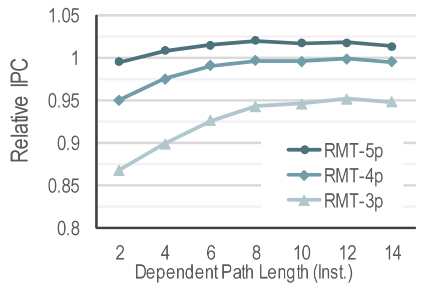

Fig. 11 IPC versus RMT read ports and dependent path length

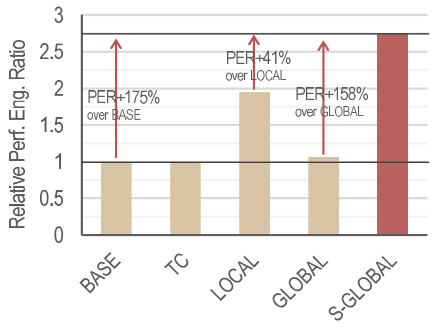

Fig. 13 Performance energy ratio (Inverse of EDP) relative to BASE

Table 3 Ratio of cached renamed operands

LOCAL	GLOBAL	S_GLOBAL
31.7%	69.3%	57.3%

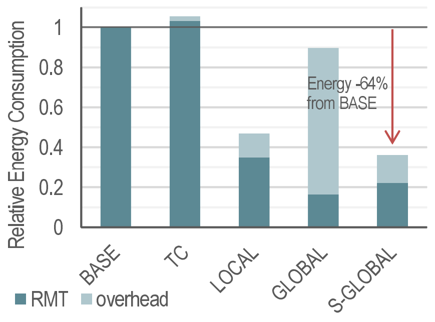

Fig. 12 Energy consumption of RMT relative to BASE

6.5 Energy Consumption

Figure 12 shows the energy consumption[†] of the RMT in each model normalized by BASE. This energy consumption is the average of all benchmark programs. "overhead" in each model shows the overhead energy incurred by the addition of the TC or RTC as compared to BASE. This overhead energy includes the energy consumption for reading the instruction cache and filling the generated traces on an RTC miss. In S_GLOBAL, the RMT is used for renaming source operands stored as logical register numbers into traces on an RTC hit, as described in Sect. 5.3.1. The part labeled RMT in Fig. 12 includes the energy consumption for this renaming.

S_GLOBAL considerably reduces the energy consumption of the RMT, including its overhead. The energy consumption of the RMT in S_GLOBAL is reduced by 64% as compared with BASE, because both the area and the access frequency of the RMT are reduced. The energy consumption is reduced by 65%, 23%, and 60% as compared with TC, LOCAL, and GLOBAL, respectively. LOCAL does not reduce its energy consumption as compared to S_GLOBAL, because LOCAL cannot cache renamed operands sufficiently, as described above. GLOBAL has a large overhead caused by the tag array in RTC, as described in Sect. 4, and consequently, this overhead cancels out the reduction in the energy consumption in the RMT. The energy consumption of the RMT without the overhead in GLOBAL is smaller than that of the RMT in S_GLOBAL; they are 16% and 22% of that of the RMT in BASE, respectively. This is because renaming is completely omitted in GLOBAL on an RTC hit, but a few operands are still renamed in S_GLOBAL, as described in Sect. 5.3.1. However, this energy consumption for renaming on an RTC hit in S_GLOBAL is negligible, and the total energy consumption in S_GLOBAL is significantly reduced as compared with that in GLOBAL and BASE, as described previously.

6.6 Performance Energy Ratio

In this section, we show the performance/energy ratio (PER) of each model, which is equal to the inverse of the EDP.

[†]This includes both dynamic and static energy consumption.

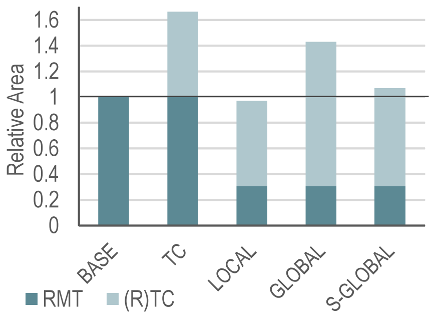

Fig. 14 Area of RMT relative to BASE

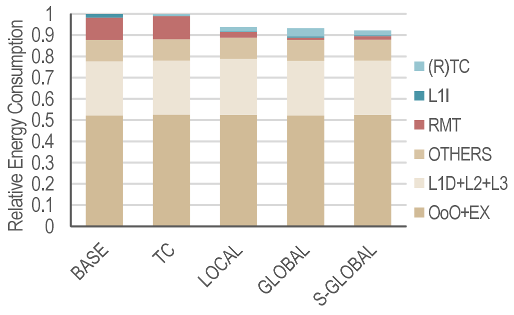

Fig. 15 Energy consumption relative to BASE on entire proccesor

A higher PER indicates higher-level efficiency. Figure 13 shows the PER relative to that of BASE. S_GLOBAL considerably improves its PER as compared to the other models, because the performance is improved/maintained and the energy consumption is reduced at the same time. Figure 13 shows that the PER of S_GLOBAL is improved by 175%, 180%, 41%, and 158% as compared to BASE, TC, LOCAL, and GLOBAL, respectively.

6.7 Area

Figure 14 shows the circuit area of the RMT and RTC in each model. These areas are normalized by that of BASE. S_GLOBAL considerably reduces the area of the RMT as compared to BASE, but the addition of RTC slightly increases its total area by 7.0%.

6.8 Energy Efficiency Impact on Entire Processor

Figure 15 shows the energy consumption of the RMT and RTC on the entire processor[†]. SGRTC reduces the energy consumption by 7.6% on the entire processor. In BASE, the RMT accounts for 10.5% of the energy consumption of the entire processor. This ratio is larger than the 6% of ARM Cortex A15, described in Sect. 2.1. This is because BASE has a wider rename-width and supports SMT, which increase the area of the RMT, as described in Sect. 2.1.

[†]These results are basically consistent with the previous results about energy consumption, but they slightly differs from the previous ones. This is because the simulator used in this evaluation differs from that used in the previous evaluation, as described in Sect. 6.1.

7. Related Work

This section describes related work for mitigating the problems of an RMT described in Sect. 2.

Some researchers focused on the fact that all the ports of an RMT are in general not entirely utilized, and they proposed methods that reduce the number of the ports of an RMT in order to mitigate its complexity [29], [30]. These methods can reduce the number of ports proportionally to the effective use rate of the ports, but this reduction is limited.

Moshovos et al. proposed a method that focuses on checkpointing an RMT for recovering from a branch misprediction [7]. This method takes checkpoints only for branch instructions with low confidence, and this allows the resources required for the RMT to be reduced. However, this method requires an additional confidence estimator for branch instructions, which consumes additional energy.

Liu et al. proposed a method that uses a *register map cache (RMC)* [31]. The RMC is smaller than the main RMT and can reduce its latency and energy consumption on a hit; however, its miss penalty may degrade performance. Miwa et al. proposed another method using an RMC [32]. This method is based on an unique pipeline structure for a register file [18] and effectively mitigates the problems related to miss penalties. However, an RMC still consumes a large amount of energy, because it consists of a heavily multi-ported CAM [32].

There are only two conventional methods that cache renamed operands, to the best of our knowledge: LRTC [9] and GRTC [10]. In our opinion, this is because caching renamed operands is essentially difficult, as described in Sect. 2. The related studies described in this section are orthogonal to LRTC, GRTC, and our proposed method, which cache renamed operands. Consequently, a combined design that achieves higher energy efficiency is possible.

8. Conclusion

We proposed SGRTC, which caches only renamed operands that are a short distance from producers outside traces. Our evaluation results show that SGRTC achieves 64% lower energy consumption for register renaming with only a 0.2% performance overhead as compared to a conventional processor. Further, the evaluation results show that the proposed method achieves an 8.5% and 4.5% higher performance, 23% and 60% lower energy consumption, and 41% and 151% higher performance-energy ratio (the inverse of the EDP) than conventional LRTC and GRTC, respectively.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Number 24680005.

References

[1] R. Shioya and H. Ando, "Energy efficiency improvement of renamed trace cache through the reduction of dependent path length," IEEE International Conference on Computer Design (ICCD), pp.416–423, 2014.

[2] B. Sinharoy, J.A. Van Norstrand, R.J. Eickemeyer, H.Q. Le, J. Leenstra, D.Q. Nguyen, B. Konigsburg, K. Ward, M.D. Brown, J.E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J.W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K.M. Fernsler, "IBM POWER8 Processor Core Microarchitecture," IBM Journal of Research and Development, vol.59, no.1, pp.2:1–2:21, Jan. 2015.

[3] K. Krewell, "Intel's Haswell Cuts Core Power," Microprocessor Report 9/24/12, pp.1–5, Sept. 2012.

[4] L. Gwennap, "ARM Optimizes Cortex-A72 For Pheones," Microprocessor Report 5/11/15, pp.1–5, May 2015.

[5] R. Kessler, "The Alpha 21264 Microprocessor," IEEE Micro, vol.19, no.2, pp.24–36, 1999.

[6] S. Manne, A. Klauser, and D. Grunwald, "Pipeline Gating: Speculation Control for Energy Reduction," Proc. Int. Symp. on Computer Architecture, vol.26, no.3, pp.132–141, 1998.

[7] A. Moshovos, "Checkpointing Alternatives for High-performance Power-aware Processors," Proc. Int. Symp. on Low Power Electronics and Design, pp.318–321, 2003.

[8] E. Rotenberg, S. Bennett, and J.E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," Proc. Int. Symp. on Microarchitecture, pp.24–34, 1996.

[9] S. Vajapeyam and T. Mitra, "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences," Proc. Int. Symp. on Computer Architecture, vol.25, no.2, pp.1–12, 1997.

[10] H. Ichibayashi, R. Shioya, H. Irie, M. Goshima, and S. Sakai, "Anti-dualflow architecture," IPSJ Trans. Advanced Computing Systems (ACS), vol.1, no.2, pp.22–33, 2008.

[11] K. Yeager, "The MIPS R10000 Superscalar Microprocessor," IEEE Micro, vol.16, no.2, pp.28–41, 1996.

[12] S. Palacharla, N.P. Jouppi, and J.E. Smith, "Quantifying the complexity of superscalar processors," Tech. Rep., University of Wisconsin-Madison, Nov. 1996.

[13] E. Safi, A. Moshovos, and A. Veneris, "On the Latency and Energy of Checkpointed Superscalar Register Alias Tables," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol.18, no.3, pp.365–377, March 2010.

[14] N.H.E. Weste and D.M. Harris, CMOS VLSI Design: A Circuits and Systems Perspective 4th Ed., Pearson/Addison-Wesley, 2011.

[15] R.P. Preston, R.W. Badeau, D.W. Bailey, S.L. Bell, L.L. Biro, W.J. Bowhill, D.E. Dever, S. Felix, R. Gammack, V. Germini, M.K. Gowan, P. Gronowski, D.B. Jackson, S. Mehta, S.V. Morton, J.D. Pickholtz, M.H. Reilly, and M.J. Smith, "Design of An 8-Wide Superscalar RISC Microprocessor with Simultaneous Multithreading," Proc. Int. Solid-State Circuits Conference, vol.1, pp.334–472, 2002.

[16] NVIDIA, "NVIDIA Tegra 4 Family CPU Architecture — 4-PLUS-1 Quad core." Whitepaper, 2013.

[17] The Standard Performance Evaluation Corporation, SPEC CPU2006 suite http://www.spec.org/cpu2006/.

[18] R. Shioya, K. Horio, M. Goshima, and S. Sakai, "Register Cache System not for Latency Reduction Purpose," Proc. Int. Symp. on Microarchitecture, pp.301–312, 2010.

[19] R. Shioya, M. Goshima, and H. Ando, "A Front-end Execution Architecture for High Energy Efficiency," Proc. Int. Symp. on Microarchitecture, pp.419–431, 2014.

[20] E. Sprangle and D. Carmean, "Increasing Processor Performance by Implementing Deeper Pipelines," Proc. Int. Symp. on Computer Architecture, vol.30, no.2, pp.25–34, 2002.

[21] R. Shioya, M. Goshima, and S. Sakai, "Design and implementation of processor simulator Onikiri2," Symposium on Advanced Computing Systems and Infrastructures (SACSIS), pp.120–121, 2009.

[22] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "CACTI 6.0: A tool to model large caches," Tech. Rep., HP Laboratories, 2009.

[23] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N. Jouppi, "CACTI 5.1.," Tech. Rep., HP Laboratories, 2008.

[24] S. Li, J.H. Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," Proc. Int. Symp. on Microarchitecture, pp.469–480, 2009.

[25] E. Rotenberg, S. Bennett, and J.E. Smith, "A Trace Cache Microarchitecture and Evaluation," IEEE Transactions on Computers, vol.48, no.2, pp.111–120, 1999.

[26] B. Black, B. Rychlik, and J.P. Shen, "The Block-based Trace Cache," ACM SIGARCH Computer Architecture News, vol.27, no.2, pp.196–207, 1999.

[27] B. Sinharoy, R. Kalla, W.J. Starke, H.Q. Le, R. Cargnoni, J.A. Van Norstrand, B.J. Ronchetti, J. Stuecheli, J. Leenstra, G.L. Guthrie, D.Q. Nguyen, B. Blaner, C.F. Marino, E. Retter, and P. Williams, "IBM POWER7 Multicore Server Processor," IBM J. Res. Dev., vol.55, no.3, pp.1:1–1:29, May 2011.

[28] T.R. Halfhill, "Oracle says SPARC is tops," Microprocessor Report 4/15/13, pp.1–6, 4 2013.

[29] A. Moshovos, "Power-aware register renaming," Computer Engineering Group Technical Report, University of Toronto, pp.1–8, 2002.

[30] R. Sangireddy, "Reducing Rename Logic Complexity for High-speed and Low-power Front-end Architectures," IEEE Transactions on Computers, vol.55, no.6, pp.672–685, 2006.

[31] T. Liu and S.-L. Lu, "Performance Improvement with Circuit-level Speculation," Proc. Int. Symp. on Microarchitecture, pp.348–355, 2000.

[32] S. Miwa, P. Zhang, H. Yokoyama, Y. Horibe, and H. Nakajo, "Area-efficient register map table using a cache," IPSJ Trans. Advanced Computing Systems (ACS), vol.3, no.3, pp.44–55, 2010.

Ryota Shioya was born in 1981. He received his M.E. and Ph.D. in Information and Communication Engineering from the University of Tokyo in 2008 and 2011, respectively. He was a research fellow of the Japan Society for the Promotion of Science from 2009 to 2011. Since 2011, he has been an assistant professor at the Graduate School of Engineering, Nagoya University. He is a member of IEICE, IPSJ, and IEEE.

Hideki Ando received his B.S. and M.S. degrees in Electronic Engineering from Osaka University, Suita, Japan, in 1981 and 1983, respectively. He received his Ph.D. degree in Information Science from Kyoto University, Kyoto, Japan, in 1996. From 1983 to 1997, he was with Mitsubishi Electric Corporation, Itami, Japan. From 1991 to 1992, he was a visiting scholar at Stanford University. In 1997, he joined the faculty of Nagoya University, Nagoya, Japan, where he is currently a Professor in the Department of Electrical Engineering and Computer Science. He received IPSJ best paper awards in 1998 and 2002, and a best paper award at the Symposium on Advanced Computing Systems and Infrastructures in 2013. His research interests include computer architecture and compilers.