PAPER

MLP-Aware Dynamic Instruction Window Resizing in Superscalar Processors for Adaptively Exploiting Available Parallelism

Yuya KORA^{†*}, Kyohei YAMAGUCHI^{††***a)}, *Nonmembers*, and Hideki ANDO^{††b)}, *Member*

SUMMARY Single-thread performance has not improved much over the past few years, despite an ever increasing transistor budget. One of the reasons for this is that there is a speed gap between the processor and main memory, known as the memory wall. A promising method to overcome this memory wall is aggressive out-of-order execution by extensively enlarging the instruction window resources to exploit memory-level parallelism (MLP). However, simply enlarging the window resources lengthens the clock cycle time. Although pipelining the resources solves this problem, it in turn prevents instruction-level parallelism (ILP) from being exploited because issuing instructions requires multiple clock cycles. This paper proposed a dynamic scheme that adaptively resizes the instruction window based on the predicted available parallelism, either ILP or MLP. Specifically, if the scheme predicts that MLP is available during execution, the instruction window is enlarged and the window resources are pipelined, thereby exploiting MLP. Conversely, if the scheme predicts that less MLP is available, that is, ILP is exploitable for improved performance, the instruction window is shrunk and the window resources are de-pipelined, thereby exploiting ILP. Our evaluation results using the SPEC2006 benchmark programs show that the proposed scheme achieves nearly the best performance possible with fixed-size resources. On average, our scheme realizes a performance improvement of 21% over that of a conventional processor, with additional cost of only 6% of the area of the conventional processor core or 3% of that of the entire processor chip. The evaluation results also show 8% better energy efficiency in terms of 1/EDP (energy-delay product).

key words: *microprocessor, superscalar processor, memory-level parallelism, instruction-level parallelism*

1. Introduction

LSI technology is still advancing according to Moore's law. The current trend in using the increased transistor budget is the integration of multiple cores and units that were previously placed outside the processor chip, including the last-level cache (LLC), graphics unit, and memory controller. This aims to increase processing throughput in a multiprogramming environment and to reduce the total system cost. Unfortunately, the core architecture has not changed significantly, and single-thread performance has improved only slightly. This paper proposes a new architectural technique

to improve single-thread performance by effectively using the increased transistor budget.

One of the problems preventing improvement in single-thread performance is the large speed gap between the processor and main memory. This gap is often referred to as the memory wall. The memory wall causes a very long (several hundred cycles) latency for a load that has missed the LLC. Conventional methods to overcome this memory wall include using a large cache together with increasing the cache hierarchical levels and incorporating a prefetcher. However, a large cache is very costly and often insufficient even when comprising several megabytes. Prefetchers [1], [2] are generally inexpensive but are effective only for regular access patterns. Some methods that are effective for irregular patterns have been documented in the literature, but these are very costly [3] because the correlation between irregular accesses must be recorded.

A promising solution involves significantly increasing the number of in-flight instructions supported by the processor. Here, the in-flight instructions are those currently being processed by the processor. Some of the in-flight instructions that have not yet begun being executed are scheduled and then reordered from the program order to achieve high performance. Increasing the number of in-flight instructions gives the processor greater opportunity to reorder instructions. This aggressive instruction execution reordering allows many cache-miss-causing loads to be executed within a short period, overlapping their access to memory. Parallel memory access reduces the memory access time per load. This type of parallelism is called *memory-level parallelism* (MLP).

Implementation of this approach, however, is problematic because the instruction window resources must be allocated to all the in-flight instructions and therefore, need to be greatly enlarged. Here, the instruction window resources include the reorder buffer (ROB) [4], issue queue (IQ) [5], and load/store queue (LSQ)^{***} (see the processor organization in Fig. 1). Simply enlarging these resources significantly lengthens the clock cycle time, offsetting the benefit of MLP exploitation, or even reduces the performance in programs with less available MLP.

^{***}We assume an Intel P6-type architecture [6] in this study, where each ROB entry has a physical register and the IQ holds operands read from either the ROB or the architectural register file. We also assume that the processor has a map table that translates a logical register into a physical register field in the ROB to implement the ROB with a RAM organization.

Manuscript received June 3, 2014.

Manuscript revised August 21, 2014.

Manuscript publicized September 22, 2014.

[†]The author is with the Department of Computational Science and Engineering, Nagoya University, Nagoya-shi, 464–8603 Japan.

^{††}The authors are with the Department of Electrical Engineering and Computer Science, Nagoya University, Nagoya-shi, 464–8603 Japan.

^{*}Presently, with the Transportation Bureau, Nagoya.

^{**}Presently, with Renesas Electronics Corporation.

a) E-mail: kyohei.yamaguchi.xb@renesas.com

b) E-mail: ando@nuee.nagoya-u.ac.jp

DOI: 10.1587/transinf.2014EDP7177

Pipelining is a general solution to avoid the adverse effect on the clock cycle time caused by the long latency of a logic circuit. For window resources, pipelining can be adopted to avoid lengthening the clock cycle time. However, this prevents instruction-level parallelism (ILP) from being exploited mainly because of the pipelined IQ. A pipelined IQ is unable to issue dependent instructions back-to-back, because the wakeup-select loop does not complete in a single cycle. This degrades IPC in compute-intensive programs.

In this paper, we propose *dynamic instruction window resizing*, which adapts the window size to the available parallelism (ILP or MLP). In this scheme, if more MLP is predicted to be exploitable, the window resources are enlarged and pipelined to exploit MLP. Conversely, if ILP is more valuable to improve performance, the window resources are shrunk and de-pipelined to exploit ILP. Prediction whether MLP is exploitable is based on a characteristic of LLC miss occurrences, where LLC misses are typically clustered with respect to time. Specifically, if an LLC miss occurs once, we predict that MLP can be exploited for a while thereafter. Conversely, once the memory latency has elapsed after the last LLC miss, we predict that MLP will not be exploitable.

This paper is an extension of our previous conference paper [7], evaluating the scheme with a more optimized configuration and providing additional evaluation results.

The remainder of this paper is organized as follows. Section 2 briefly explains the base processor organization we assume in this study. Section 3 summarizes the tradeoffs in the instruction window size. Our dynamic instruction window resizing scheme is introduced in Sect. 4, while the evaluation results are presented in Sect. 5. Related work is discussed in Sect. 6 and our conclusions are given in Sect. 7.

2. Processor Organization

We briefly explain the base processor organization assumed in this paper. For further details, refer to the various textbooks [8]–[10].

Figure 1 shows an overview of the organization of the processor. Multiple instructions are fetched from the L1 instruction cache, and are then decoded. The instruction window resources are allocated to the decoded instructions. The logical registers of the decoded instructions are then mapped to entries in the ROB at the rename stage. The renamed instructions, except for memory instructions, are inserted into the IQ with source operands fetched from the register file or physical registers of the ROB. The memory instructions, however, are split into address calculation instructions and memory access instructions. The address calculation instructions are inserted into the IQ, in the manner previously described for normal instructions, while the memory access instructions are inserted into the LSQ. The instructions in the IQ await the resolution of their register dependences, whereas those in the LSQ await the resolution of their memory dependences. Instructions with resolved dependences in the IQ and LSQ, respectively, are issued to a function unit

Fig. 1 Base processor organization.

or access the L1 data cache. The results of instruction execution are written into the allocated physical registers of the ROB. The head of the ROB is committed if the associated instruction has written the result into the corresponding physical register. Specifically, the result is written into the register file if no exception has occurred during execution; otherwise the exception is handled.

3. Tradeoffs in the Instruction Window Size

As previously mentioned, the following tradeoff with regard to the size of the instruction window exists when exploiting ILP and MLP.

1. A large instruction window resource is beneficial for exploiting MLP in memory-intensive programs. However, it must be pipelined so as not to lengthen the clock cycle time. This makes it impossible to issue dependent instructions back-to-back because of pipelining the IQ, thereby preventing ILP from being exploited. This results in performance degradation in compute-intensive programs.
2. Conversely, a small non-pipelined instruction window resource is beneficial for exploiting ILP in compute-intensive programs. However, this makes it difficult to overlap many memory accesses caused by miss loads, resulting in deterioration in MLP. This means that the performance of memory-intensive programs is barely improved.

To illustrate this tradeoff, Fig. 2 shows the performance

Fig. 2 IPC for a varying instruction window size.

(IPC) for *libquantum* (a memory-intensive program) and *gcc* (a compute-intensive program) when varying the instruction window size. The configuration of the base processor used in the evaluation is given in Table 1. The horizontal axis denotes the size and pipeline depth of the IQ, which is representative of the instruction window resources. The sizes of the other window resources are changed in proportion to the size of the IQ, while the pipeline depth is the same as that of the IQ. Each bar represents the IPC relative to that in the smallest window case, while the line represents the relative IPC of the ideal processor, in which the window resources are enlarged but not pipelined.

The bars in the two figures show the tradeoff described previously. That is, large window resources are beneficial for *libquantum*, a memory-intensive program, whereas they are harmful for *gcc*, a compute-intensive program.

The following two points should also be noted. First, the deterioration in ILP with large window resources, which is caused by pipelining these resources, hardly affects the performance of *libquantum*, because the memory-access time dominates execution time. As shown in Fig. 2 (a), although the non-pipelined ideal processor does not reduce ILP exploitation, the IPC difference for the pipelined processor is very small.

The second point to be noted is that enlarging the window resources is not particularly beneficial for compute-intensive programs, as seen in the IPC for the ideal processor in Fig. 2 (b). Enlarging the window resources increases IPC only slightly, even if the adverse effects of pipelining are removed. This means that a small window size is sufficient to exploit ILP.

Detailed evaluation results highlighting this tradeoff are presented in Sect. 5.3.

4. MLP-Aware Dynamic Instruction Window Resizing

This section introduces dynamic instruction window resizing. In this study, we assumed an Intel P6-type architecture as the base architecture, where the instruction window resources include the ROB, IQ, and LSQ. All these resources have a FIFO structure. Therefore, when at a particular time the region from the head to a particular entry is used, resizing is carried out by moving the boundaries of the used and unused regions as shown in Fig. 3. During resizing, the processor is stalled for several cycles to configure the pipeline depth and activate or disable the circuits (e.g., enabling or disabling precharge for dynamic circuits) in used and un-

Fig. 3 Resizing resources by changing the level.

Fig. 4 Histogram of L2 cache miss occurrences for miss intervals (*sphinx3*).

used regions. We refer to these cycles as the *level transition penalty*.

Here, we define the *instruction window resource level* (henceforth simply referred to as the *resource level* or *level*, depending on the context) as comprising values of the size and pipeline depth of the resource ($level = \{size, pipeline-depth\}$). As the level number increases, so too does the corresponding size. Each resource with a particular size is pipelined so that it does not increase the clock cycle time.

4.1 Overview of Enlarging and Shrinking Window Resources

In this study, we assume the L2 cache to be the LLC. In general, L2 cache misses tend to be clustered with respect to time. This is because there is a moment when the degree of locality of memory accesses decreases because of a phase change in the program execution. Figure 4 shows the histograms of the L2 cache miss occurrences for miss intervals in *sphinx3* with a bin size of eight cycles (see the processor configuration in Table 1), as an example. As shown in the figures, the vast majority of L2 cache misses occur within a short interval, although the average interval calculated by misses per instruction and IPC is 274. Furthermore, note that there is another peak at around 300 cycles, because memory latency is 300 cycles in this evaluation. The pipeline stalls during this period because it runs out of instruction window resources after an L2 miss occurs. Then, once that miss has been resolved, another cluster of misses occurs. Thus, if we enlarge the instruction window, this peak will move to the left on the x-axis, causing nearby occurrences of more L2 cache misses.

Based on this, our scheme predicts that once one L2 cache miss has occurred, further misses will occur contin-

```

1 cycle = 0;           // current clock cycle
2 level = 1;           // instruction window resource level
3 shrink_timing = -1;  // timing of shrinking
4 do_shrink = 0;       // flag instructing the shrinking of resources
5
6 for each cycle {
7     if (L2_miss) {
8         level = min(level + 1, max_level); // enlarge the resources
9         shrink_timing = cycle + memory_latency;
10        do_shrink = 0;
11    } else if (cycle == shrink_timing) {
12        do_shrink = 1;
13    }
14    if (level > 1 && do_shrink) {
15        // check if the regions of ROB, IQ, and LSQ to be removed by shrinking are vacant
16        if (is_shrinkable(level)) {
17            level = level - 1; // shrink the resources
18            shrink_timing = cycle + memory_latency;
19            do_shrink = 0;
20        } else {
21            stop_alloc(); // stop resource allocation to increase the vacancies in resources
22        }
23    }
24 }

```

Fig. 5 Algorithm for dynamic instruction window resizing.

uously for a while, and thus, the window resources are enlarged. Specifically, when an L2 cache miss occurs, the level of each window resource increases by one (if it is already at its maximum, it does not change).

In contrast, our scheme shrinks the window resources once the main memory latency has elapsed from the time when the last L2 cache miss occurred. Specifically, the level of each window resource decreases by one (if it is currently at level 1, the level does not change). Note that shrinking the window resources is delayed until the regions of the ROB, IQ, and LSQ that are to be removed by the shrink simultaneously become vacant.

4.2 Algorithm

The algorithm for our dynamic instruction window resizing scheme is summarized using pseudo code in Fig. 5.

In a cycle when an L2 cache miss occurs, the instruction window resource level is increased by one (if it is already at its maximum, it does not change) (line 8). Then, the time at which the resources should be shrunk, *shrink_timing*, is set to the current cycle plus the memory latency to find the time for the subsequent shrink (line 9). Also, *do_shrink*, the flag that instructs shrinking of the resources, is cleared (line 10).

If the cycle reaches *shrink_timing* without an L2 cache miss occurring, the flag *do_shrink* is set to allow the resources to be shrunk in this cycle or a later cycle (line 12).

If the current resource level is greater than one and flag *do_shrink* is set (line 14), the algorithm checks whether all the instruction window resources (i.e., ROB, IQ, and LSQ) can be shrunk simultaneously by checking whether the regions that must be removed are all vacant (line 16).

- If vacant, the resource level is decreased by one (line

Fig. 6 Resource level transitions with L2 cache miss occurrences.

17). This means that the resources are shrunk. Then, *shrink_timing* is set to the current cycle plus the memory latency for the next shrink (line 18), and the flag *do_shrink* is cleared (line 19).

- If not vacant, shrinking is not performed in this cycle, but is postponed until a later cycle. Resource allocation at the front is stopped to drain instructions from the window and allow the regions that need to be removed to become vacant (line 21).

Figure 6 illustrates how the resource level transition occurs, with a maximum level of three in this example. At time *t0*, an L2 cache miss occurs, and thus the level is increased by one. Similarly, at time *t1*, another L2 cache miss occurs, and the level is again increased by one to three. At time *t2*, an L2 cache miss occurs again, but this time the level is not increased because it is already at the maximum. At time *t4*, the memory latency elapses after the last L2 miss at *t2*. The level must now be decreased by one, but this is postponed until *t5* because the scheme must wait for the regions to be removed to become vacant by halting resource allocation. A second memory latency subsequently elapses, and the level is again decreased by one at time *t6*. At this time, the regions to be removed are all vacant, and the resources

are immediately shrunk. Note that MLP is exploited from time $t1$ to $t3$ (with the memory accesses overlapping during this period), while ILP is exploited before $t0$ and after $t6$.

5. Evaluation

We first explain the assumptions of size and pipeline depth of the instruction window resources for several resource levels in Sect. 5.1. Then, we evaluate the performance when we introduce dynamic instruction window resizing in Sects. 5.2 and 5.3; the configuration of the base processor used in our evaluation is given in Table 1. Note that we introduced a stride prefetcher based on the proposal in [2]. We chose the stride prefetcher as a data prefetcher because commercial processors (e.g., IBM Power 5, 6, and 7, Intel Sandy Bridge, and AMD Opteron) use a stream or stride prefetcher. In Sects. 5.4 and 5.5, we evaluate energy efficiency and the cost/performance ratio, respectively, while in Sects. 5.6, 5.7, and 5.8, we evaluate the performance impact of the pipeline depth of the instruction window resources, the level transition penalty, and the number of configurable levels, respectively. In Sect. 5.10, we compare the performance of the proposed dynamic instruction window resizing scheme with *runahead execution* [11], which is a scheme that effectively exploits both ILP and MLP, and which is used in commercial processors (the Sun Rock processor [12] and IBM Power6 [13]). Finally, in Sect. 5.11, we evaluate our scheme in the case of a three-level cache hierarchy.

5.1 Size and Pipeline Depth of Window Resources

Unless specified otherwise in the following evaluation, the processor incorporating the dynamic instruction window resizing scheme has physical window resources four times larger than those in the base processor, and two levels can be configured with our scheme as shown in Table 2. This number of levels is optimal in terms of performance, as described in Sect. 5.8.

The pipeline depth of the IQ was determined by evaluating the delay using an HSPICE circuit simulation after drawing the layout, based on a study in the literature [14]. In this simulation, we assumed MOSIS design rules [15] for 32nm LSI technology, and used the predictive transistor model [16] developed by the Nanoscale Integration and Modeling Group of Arizona State University for HSPICE. We assumed that the clock cycle time was determined by the delay of the IQ in the base processor (64 entries). An IQ with a particular size was pipelined so that the clock cycle time does not increase. If we let the delay of the IQ at level L be $D(L)$, then the pipeline depth of the IQ at level L is given by $\left\lceil \frac{D(L)}{D(1)} \right\rceil$.

The pipeline depth of the ROB is not a concern in terms of allocating and committing, because it does not affect IPC. However, the pipeline depth for reading the register fields does affect IPC, because it changes the branch misprediction penalty. We obtained the delay of a register field read using

Table 1 Configuration of base processor.

Pipeline width	4-instruction wide for each of fetch, decode, issue, and commit
ROB	128 entries
Issue queue	64 entries
LSQ	64 entries
Branch prediction	16-bit history 64K-entry PHT gshare, 2K-set 4-way BTB, 10-cycle misprediction penalty
Function unit	4 iALU, 2 iMULT/DIV, 2 Ld/St, 4 fpALU, 2 fpMULT/DIV/SQRT
L1 I-cache	64KB, 2-way, 32B line
L1 D-cache	64KB, 2-way, 32B line, 2 ports, 2-cycle hit latency, non-blocking
L2 cache	2 MB, 4-way, 64B line, 12-cycle hit latency
Main memory	300-cycle min. latency, 8B/cycle bandwidth
Data prefetcher	stride-based, 4K-entry, 4-way table, 16-data prefetch to L2 cache on miss

Table 2 Number of entries and pipeline depths of window resources at each level and the assumption of a cycle penalty at the level transition.

resource	parameter	level	
		1	2
IQ	entries	64	256
	pipeline depth	1	2
ROB	entries	128	512
	pipeline depth	1	2
LSQ	entries	64	256
	pipeline depth	1	2
Level transition penalty		10 cycles	

CACTI 6.5 [17], and then determined the pipeline depth.

The pipeline depth of the LSQ was simply set to be identical to that of the IQ at each level. Although this is not entirely accurate, we consider it to be acceptable for our experiments.

In Table 2, we also include an assumption of the level transition penalty. Without an actual physical design, accurately estimating this penalty is difficult. The impact on and sensitivity to performance of this penalty is evaluated in Sect. 5.7.

5.2 Environment for Performance Evaluation

We built a simulator based on the SimpleScalar Tool Set version 3.0a [18] to evaluate performance. The instruction set used is the Alpha ISA. We used all programs from the SPECint2006 benchmark suite and 16 programs from the SPECfp2006 benchmark suite (only *wrf* is excluded because it does not run correctly on our simulator at present). The programs were compiled using gcc ver.4.5.3 with option -O3. In Table 3, the benchmark programs are categorized in terms of whether they are memory- or compute-intensive, with a threshold for average load latency of 10 cycles. This categorization was used in the following evaluation results. We simulated 100M instructions after the first 16G instructions were skipped with the *ref* inputs.

In the following sections, to save space we give

Fig. 7 IPC normalized by the base.

Table 3 Benchmark programs.

category	programs
memory-intensive	int: hmmer, libquantum, mcf, omnetpp, xalancbmk
	fp: GemsFDTD, lbm, leslie3d, milc, soplex, sphinx3
compute-intensive	int: astar, bzip2, gcc, gobmk, h264ref, perlbench, sjeng
	fp: bwaves, cactusADM, calculix, dealII, gamess, gromacs, namd, povray, toonto, zeusmp

the results of only several selected programs (eight of the memory-intensive programs and six of the compute-intensive programs). We also give “GM mem”, “GM comp”, and “GM all”, denoting the geometric means of all memory-intensive programs, all compute-intensive programs, and all programs, respectively, listed in Table 3 including the he non-selected programs.

5.3 Performance

We evaluated the performance of the following three models:

- **Fixed size model:** The size of the window resources is fixed during execution and the resources are pipelined. We evaluated three sub-models with different resource sizes, i.e., levels 1, 1.5, and 2. The sizes at levels 1 and 2 are given in Table 2. The sizes at level 1.5 are the averages of the sizes at levels 1 and 2; that is, IQ, ROB, and LSQ have 160, 320, and 160 entries, respectively, with a pipeline depth of 2 for all window resources. (these pipeline depths were also determined by HSPICE or CACTI simulation). Instruction issues from the IQ and LSQ require multiple cycles of pipelining. Moreover, an additional branch misprediction penalty is imposed for the extra delay related to the enlarged IQ and from reading the enlarged ROB.
- **Dynamic resizing model:** Each of the window resources has a maximum physical size (i.e., the size at resource level 2), but is resized dynamically using our

scheme with the number of configurable levels being two. Both the issue delay and extra branch misprediction penalty are imposed, as per the fixed size model, depending on the pipeline depth of the resource.

- **Ideal model:** Each window resource has an identical size to the corresponding resource in the fixed size model, but it is not pipelined. Thus, no extra issue delay or branch misprediction penalty is imposed. We also assume that there is no adverse effect on the clock cycle time.

Figure 7 shows the evaluated IPC relative to that of the base model. The three bars on the left of each graph show the relative IPC for resource levels 1, 1.5, and 2 in the fixed size model (labeled “Fix”), while the bar on the right shows the relative IPC for the dynamic resizing model (labeled “Res”). The line shows the relative IPC of the ideal model. The first eight graphs from (a) to (h) show the relative IPCs for the selected memory-intensive programs, and the six graphs from (j) to (o) show those for the selected compute-intensive programs. As described before, “GM mem” (i), “GM comp” (p), and “GM all” (q) depict the geometric means in all memory-intensive programs, all compute-intensive programs, and all evaluated programs of SPEC2006, respectively, including the non-selected programs.

As shown in the figures, the fixed size model achieves the best performance at level 2 for all memory-intensive programs. MLP is exploited aggressively with large window resources and there is a significant improvement in performance as the resource level increases.

Conversely, for compute-intensive programs, the performance of the fixed size model is not as sensitive to the level, and even decreases as the level increases in several programs. The deterioration in ILP caused by pipelining of the resources overshadows the benefit derived from greater exploitation of MLP.

Although the optimal resource level differs in the fixed size model, depending on the program, the dynamic resizing

Fig. 8 Number of cycles as a percentage with the window resources configured to particular levels.

model achieves a performance that is as good as the best performance for all levels of the fixed size model. This implies good adaptability of our dynamic instruction window resizing scheme. In terms of the geometric means, the speedup over the base is 52% (GM mem), 4% (GM comp), and 21% (GM all) for all memory-intensive, all compute-intensive, and all evaluated programs, respectively.

Compared with the best performance for all levels of the ideal model, which has no drawbacks due to enlargement, there is no significant degradation in the performance of the dynamic resizing model for any program, as confirmed in Fig. 7. In terms of the geometric means, the dynamic resizing model is inferior to the ideal model by only 2%, 3%, and 3% for all memory-intensive, all compute-intensive, and all evaluated programs, respectively. This further indicates that our dynamic resizing scheme is highly adaptable.

Figure 8 shows the number of cycles as a percentage, where the window resources were configured to particular levels in the dynamic resizing model. Unsurprisingly, resource level 1 is generally selected most often in compute-intensive programs, whereas resource level 2 is generally selected most often in memory-intensive programs.

In contrast to such typical programs, *omnetpp* is exceptional and very interesting. This program is memory-intensive, and its best performance is achieved at level 2 in the fixed size model. However, the dynamic resizing model outperforms this model by 5%. This is because, in this program, the compute-intensive and memory-intensive phases are well mixed, and dynamic resizing works well and reacts adaptively to these phases, exploiting both MLP and ILP.

5.4 Energy Efficiency

This section evaluates the energy efficiency, i.e., the performance per unit of energy (which is proportional to 1/EDP (energy-delay product)), using the McPAT [19] and assuming 32nm LSI technology and a temperature of 350K. Figure 9 shows the results. The vertical axis is the IPC per unit of energy normalized according to that of the base processor.

As observed in the figure, dynamic resizing performs significantly better than the base for memory-intensive pro-

Fig. 9 Energy efficiency (1/EDP).

Table 4 Additional cost vs. speedup (*1: based on the area of the base processor).

additional cost	value	1.6mm ²
	vs. base core	6%
	vs. SB core	8%
	vs. SB chip	3%
speedup	achieved	21%
	expected based on Pollack's law*1	3%
	augmented L2 cache	1%

grams. This is because although the large window resources consume more power, the performance is significantly improved, as shown in Fig. 7. In particular, the improvement is dramatic for *libquantum* (432%).

Conversely, the energy efficiency for dynamic resizing is mostly equivalent to that of the base in compute-intensive programs. This is unsurprising, because level 1 is mostly selected in these programs, as shown in Fig. 8.

The average improvements are 38%, -7%, and 8% for all memory-intensive, all compute-intensive, and all evaluated programs, respectively.

5.5 Cost/Performance Ratio

In this section, we evaluate the cost/performance ratio of our scheme. The cost is estimated using the McPAT and assuming 32nm LSI technology. Table 4 lists the estimated additional costs along with the speedup. The rows labeled “vs. base core,” “vs. SB core,” and “vs. SB chip” for additional costs give the ratios of the additional costs to the area of the base core, a single core of the Intel Sandy Bridge, and an entire Sandy Bridge chip [20], respectively. These areas are 25mm², 19mm², and 216mm², respectively. Note that the Sandy Bridge has four cores and is fabricated using 32nm LSI technology. In the calculations of the additional costs for the Sandy Bridge chip, we assumed that our scheme would be used with all four cores.

As listed in the table, the additional costs compared with our base core and the Sandy Bridge core are 6% and 8%, respectively. The main reason why the ratio to the Sandy Bridge core is larger is that it includes only a 256KB L2 cache, while our base core includes a 2MB one. Compared with the entire Sandy Bridge chip, the additional cost is only 3%. Considering the significant speedup (21%) achieved over the base, the proposed architecture achieves

Fig. 10 Comparison with the enlarged L2 cache model.

a good cost/performance ratio that far exceeds that based on Pollack's law [21], which states that performance improves in proportion to the square-root of the processor area. This indicates that our scheme is significantly more effective in its use of the increased transistor budget. According to Pollack's law, a 6% cost increase relative to the base core should yield a speedup of only 3%.

The question then arises whether it is better to enlarge the L2 cache using the same additional cost. To answer this question, we evaluated the performance of the base processor with an enlarged 2.5MB, five-way L2 cache. Because the area of the 2MB, four-way cache (the base configuration) is 8.6mm² (calculated by the McPAT), the increased cost of a 2.5MB, five-way cache is approximately 1.3 times greater than the additional cost when using our scheme. The evaluation results shown in Fig. 10 confirm that the average IPC of a processor with a 2.5MB, five-way L2 cache increases by 0.6% over that of the base processor. Considering that the speedup of our dynamic resizing model increases by 21%, the area efficiency of our dynamic resizing scheme is significantly better.

5.6 Performance Impact of the Pipeline Depth of the Instruction Window Resources

We determined the pipeline depth of the window resources, assuming a particular circuit design [14], transistor models [16], and LSI technology in Sect. 5.1. Under different assumptions, the pipeline depths may differ. Therefore, this subsection evaluates the impact of the pipeline depth of the instruction window resources on performance.

Figure 11 shows the evaluated results when setting the pipeline depth of all instruction window resources to 2, 3, and 4 at level 2. The vertical axis represents IPC relative to that obtained for a pipeline depth of two for all window resources.

As expected, IPC degrades as the pipeline depth increases. The degradation is greater in memory-intensive programs than in compute-intensive ones, because the resource level remains at level 2 for a longer time than at level 1 in memory-intensive programs. However, the degradation is fairly limited, only 4% on average for the memory-intensive programs even at a pipeline depth of four (1% for the compute-intensive programs). This insensitivity stems

Fig. 11 Performance when varying the pipeline depth of the resources.

Fig. 12 Performance when varying the level transition penalty.

from the fact that memory-access time dominates execution time in memory-intensive programs. The speedup over the base is still large (18%) for all programs at a pipeline depth of four.

5.7 Performance Impact of Level Transition Penalty

Thus far, we have assumed that the level transition penalty is 10 cycles. As described in Sect. 5.1, it is difficult to estimate this penalty accurately without a physical design. Although we feel that a penalty of 10 cycles is fairly pessimistic, this section evaluates the performance impact and sensitivity of the penalty using even more pessimistic settings.

Figure 12 shows the performance when varying the level transition penalty. The vertical axis represents the IPC of the dynamic resizing model relative to that with a level transition penalty of zero cycles. The four bars per program represent the relative IPC when the level transition penalty is set to 0, 10, 20, and 30 cycles. Note that the vertical axis does not start at zero and we magnify the region close to 1.0 so that the differences are visible.

As shown in the figure, the performance impact of the level transition penalty is very small. This means that any effort to minimize the level transition penalty is not important. This low impact is because the program phase transition is not frequent, and thus the frequency of level transition is low. Table 5 gives the average interval in cycles of level transitions for each program. The average interval for all programs is 75,466 cycles while the median for all programs is 2,048 cycles. This low frequency has only a small impact on performance.

Table 5 Average cycle interval of level transition.

memory-intensive program	interval (cycles)	compute-intensive program	interval (cycles)
libquantum	1800488	gcc	3605
omnetpp	2971	gobmk	1856
GemsFDTD	87522	sjeng	1219
lbm	795	bwaves	160600
leslie3d	3884	dealII	6123
milc	1739	tonto	3068
soplex	2240		
sphinx3	1839		
average for all programs			75466
median for all programs			2048

Fig. 13 Performance when varying the number of configurable levels.

5.8 Performance Impact of Number of Configurable Levels

The number of configurable levels for a given physical maximum size of the instruction window affects the performance. As this number increases, effecting a change in the instruction window size becomes slower, because more events are needed for a change of the same size. With slower window enlarging, MLP is not fully exploited in memory-intensive programs, because MLP is exploited more with a larger window size. With slower reduction, ILP cannot be fully exploited in compute-intensive programs, because the pipeline depth of the resources is one only at level 1.

However, as the number of configurable levels increases, the instruction window resizing becomes more gradual. This may reduce the negative impact when the prediction for the window resizing is incorrect. An incorrect prediction occurs typically if an L2 miss is caused in an isolated way. In this case, MLP cannot be exploited and ILP is lost. As described in Sect. 4.2, the instruction window cannot be shrunk until the region to be removed becomes vacant after the memory latency since the last L2 miss has elapsed. This shrink delay becomes smaller with a smaller number of entries to be removed. Therefore, as the number of configurable levels increases, the shrink delay decreases, and thus the negative impact of incorrect prediction decreases.

Figure 13 shows the evaluated performance when varying the number of configurable levels. The vertical axis represents the IPC relative to that with two configurable levels. The three bars per program represent the relative IPC when

Table 6 Percentage of cycles when the instruction window is configured to the most or non-optimal levels.

(a) memory-intensive programs			
# of configurable levels	actual configured level		
	1	maximum	other
2	8.9%	91.1%	0.0%
4	6.1%	89.3%	4.6%
8	5.3%	88.7%	6.0%
(b) compute-intensive programs			
# of configurable levels	actual configured level		
	1	maximum	other
2	77.5%	22.5%	0.0%
4	71.6%	15.6%	12.8%
8	67.9%	14.2%	17.9%

the number of configurable levels is set to 2, 4, and 8, respectively. Note that the vertical axis does not start at zero, and we magnify the area around 1.0 so that the differences are visible.

As shown in the figure, in most programs, performance degrades more, as the number of configurable levels increases. However, the sensitivity is very small, because L2 cache misses are well clustered, and thus the instruction window is configured quickly to the optimal level even with a large number of configurable levels. Table 6 lists the percentage of cycles when the instruction window is configured to the minimum level (i.e., level 1), maximum level, and other levels. When the window is configured to the maximum or minimum levels, MLP or ILP is fully exploited, whereas at the other levels, it is not. As listed in the table, the percentage of cycles at the optimal levels is not that different for different numbers of configured levels. This minimizes the sensitivity of the number of configurable levels to performance.

Although the number of configurable levels has little impact on performance, a small number is slightly better. In addition, an increase in the number of configurable levels makes the hardware more complex. Thus, the smallest number, i.e., two, is the best.

5.9 Sensitivity to the Physical Size of the Instruction Window

Thus far, we have assumed that the physical size of the instruction window is four times larger than that in the base. This section evaluates the performance when varying the physical size of the instruction window.

Figure 14 shows the results. The vertical axis represents the IPC relative to that in the base. The four bars per program represent the IPC for the base, and dynamic resizing models with a 2-, 4-, and 8-fold larger physical instruction window that that in the base (henceforth, we refer to these three dynamic resizing models as the 2×, 4×, and 8× models, respectively). The pipeline depths of the IQ, ROB, and LSQ at the maximum size (i.e., level 2) are two, two, and three cycles for the 2×, 4×, and 8× models, respectively. These depths are derived as explained in Sect. 5.1.

Fig. 14 Performance when varying the size of the instruction window.

As expected, as the physical size of the instruction window increases, performance increases in memory-intensive programs. A more aggressive reordering with a larger instruction window exploits more MLP. The speedup over the base for memory-intensive programs is significant even in the 2 $\times$ model, which is 22% on average. The 8 $\times$ model achieves as much as 69% speedup. As LSI technology advances, the additional area for dynamic instruction window resizing is reduced. Thus, our scheme will become more cost-effective in the future.

In compute-intensive programs, the performance difference of the dynamic resizing model compared with the base is very small even in the 8 $\times$ model, where the pipeline depth at level 2 is three cycles.

Overall, the speedups over the base are 9%, 21%, and 28% for the 2 $\times$, 4 $\times$, and 8 $\times$ models, respectively.

5.10 Comparison with Runahead Execution

In this section, we compare the performance of *runahead execution* [11] with that of our scheme. Runahead execution is a scheme that exploits MLP by pre-execution. This scheme requires only small instruction window resources, and thus can also exploit ILP effectively. To the best of our knowledge, runahead execution is well known as one of the best schemes for exploiting both ILP and MLP in terms of performance and accommodating existing processor architectures. In fact, it has been adopted in commercial processors, including the Sun Rock processor [12] and the IBM Power6 [13].

In this scheme, if an L2 cache miss occurs, normal execution is halted with the architectural state checkpointed, and the scheme enters a special mode called runahead. In this mode, instructions following the missed load are executed until the triggered miss is resolved. If another L2 cache miss occurs while in runahead mode, MLP is exploited, overlapping the main memory access with that caused by the runahead triggered load. When the cache miss of the triggered load is resolved, the runahead mode ends, the normal mode is resumed, and execution restarts from the checkpoint. The re-executed load hits the cache this time.

Figure 15 compares the performance of our dynamic resizing scheme with that of runahead execution. The vertical axis represents the IPC normalized according to that of

Fig. 15 Performance comparison of the dynamic resource resizing scheme with runahead execution.

the base processor. The configuration of the runahead execution processor, based on that in [11], is the same as that of the base processor, except that it has two checkpointing register files for integer and floating-point and a two-port, 512-byte, four-way runahead cache for resolution of the memory dependences during the runahead mode. We assume that no cycle penalty is imposed for checkpointing or resuming the normal mode.

As shown in the figure, although runahead execution is effective for memory-intensive programs, it is inferior to the dynamic instruction window resizing scheme on average. The dynamic instruction window resizing scheme achieves 8% and 1% better performance on average than runahead execution for memory- and compute-intensive programs, respectively. This speedup is due to the fact that runahead execution cannot perform computations while MLP is being exploited in the runahead mode, whereas the large window scheme does not have this problem. This is not the case for extremely memory-intensive programs like *libquantum*, because computation almost stalls, but it is true for most memory-intensive programs. Runahead execution exploits MLP at the expense of abandoning computation because of the small window. In contrast, the large window scheme does not have to abandon computation. The large window accepts the fetched instructions even when a load causes an L2 miss, and issues other L2 miss loads while simultaneously executing computational instructions. In other words, the large window can exploit MLP without abandoning computation.

Note that IPC for runahead execution is lower than that of the base in *milc*. This is caused by useless runahead, where a few L2 misses occur while in the runahead mode. This is equivalent to stalling execution without MLP exploitation. In general, this situation occurs in programs where L2 misses do not occur that frequently and are not clustered. This situation is also disadvantageous to our dynamic instruction resizing scheme, but the performance loss is not as high (ILP is lost) as the significant loss in runahead execution, because the runahead period is very long (main memory latency of 300 cycles according to our evaluation). We introduced a mechanism in our runahead simulator to eliminate this undesirable behavior. The mechanism uses a *runahead cause status table* [11] to predict the usefulness of

Table 7 Configuration of three-level cache hierarchy.

L1 I-cache	64KB, 2-way, 32B line
L1 D-cache	64KB, 2-way, 32B line, 2 ports, 2-cycle hit latency, non-blocking
L2 cache	256KB, 4-way, 64B line, 12-cycle hit latency
L3 cache	2 MB, 4-way, 64B line, 36-cycle hit latency

Fig. 16 Histogram of L3 cache miss occurrences for miss intervals (*sphinx3*).

the transition to runahead mode based on the past usefulness of runahead for each L2 missed load, and suppresses the transition to runahead mode if it is predicted to be useless. However, this prediction is difficult, and depending on the program, useless runahead cannot always be eliminated. While the performance of runahead execution is sensitive to the degree of L2 miss clustering, our dynamic instruction window resizing scheme is rather more tolerant.

5.11 The Case of a Three-Level Cache Hierarchy

Recent processors are equipped with three-level cache hierarchies. So far, we have evaluated the case in which the processor has a two-level cache hierarchy. In this section, we evaluate the case with a three-level cache hierarchy. Table 7 lists the configuration of the three-level cache hierarchy; other configuration settings of the processor are identical to those in Table 1.

First, we evaluate whether LLC misses are clustered as shown in Fig. 4. Figure 16 shows the histograms of L3 cache-miss occurrences for miss intervals in *sphinx3*. As confirmed by comparing this figure with Fig. 4, both histograms are very similar. The only slight difference is that the second peak is moved approximately 36 cycles toward the right. This is caused by the timing change due to the addition of the L3 cache hit latency (36 cycles) to the LLC missed load latency. This similarity is unsurprising since the occurrence of LLC misses simply depends solely on the LLC configuration (capacity and associativity); LLC misses do not depend on the caches between the L1 and LLC, because the data are brought to the L1 cache through the LLC at a miss, and thereby the data that are not in the LLC are generally not in the upper-level caches.

Next, we evaluate the performance of our dynamic instruction window resizing. Figure 17 compares the cases with three-level and two-level cache hierarchies. The ver-

Fig. 17 Comparison of the cases with three-level and two-level cache hierarchies.

tical axis is IPC relative to that of each baseline processor. As observed in the figure, the relative performance in the three-level cache hierarchy case is very similar to that in the two-level cache hierarchy case for all programs. Overall, the relative performance in the three-level cache is improved by 1%. This improvement arises from the addition of the L3 cache hit latency to the LLC missed load latency, where MLP exploitation is more effective.

6. Related Work

In this section, we first discuss the approach for exploiting MLP using a large instruction window. We then review schemes for resource resizing, and finally consider MLP exploitation techniques.

6.1 Large Instruction Window

To the best of our knowledge, Cristal *et al.* were the first to suggest that the ability to support many in-flight instructions is very effective in overcoming the memory wall [22]. Unfortunately, they thought (as did many others) that a simple implementation, based on enlarging the window resources, was impractical because of the delay, area, and power overheads. However, the delay issue can be solved by pipelining, and the pipeline delay has little effect on MLP exploitation in memory-intensive programs. Although pipelining causes ILP loss in compute-intensive programs, this can be solved by adaptive resizing of the window resources using our scheme. Additionally, because of the rapidly rising transistor budget and high levels of integration on processor chips, the area overhead has now become acceptable, that is, only 3% of the entire processor chip area as discussed in Sect. 5.5. Moreover, energy efficiency is not a problem; efficiency is 8% better than that of a conventional processor as explained in Sect. 5.4.

6.2 Resizing Resources

Albonesi *et al.* presented a comprehensive survey of studies on resource resizing to improve power efficiency [23]. However, our policy for resource resizing is completely different to that proposed in previous studies. Most policies focus on the demand for resources, whereas our LLC-miss-driven

policy is MLP-aware. In other words, it focuses on which form of parallelism, i.e. MLP or ILP, is most effective in improving performance.

Ponomarev *et al.* proposed a resizing scheme focusing on occupation of the IQ [24]. This scheme shrinks the IQ if the average number of entries occupied by instructions in a certain period is smaller than a predetermined value. Conversely, it enlarges the IQ if the stall cycles caused by a full IQ exceed a predetermined threshold. This scheme is potentially suitable for exploiting MLP by enlarging the IQ, but it lacks a perspective on MLP. In other words, the scheme views the overflow of the IQ simply as an indication that IQ enlargement is beneficial. In fact, because it is generally beneficial for the insertion rate into the IQ to be set greater than the average issue rate, the IQ eventually becomes full, even when no LLC miss occurs. As a result, the scheme enlarges the IQ even in situations where MLP is not exploitable, thus wasting power.

Folegnani *et al.* proposed a resizing scheme for the IQ that deactivates those parts that contribute little to performance [25]. This scheme periodically counts the number of committed instructions in the last section with a predetermined size. If the number is less than the predetermined threshold, the section is deactivated. However, this scheme enlarges the IQ periodically to check whether enlargement is beneficial. Although this scheme has the advantage of directly monitoring the contribution of a particular section to the overall performance, it has no systematic policy for IQ enlargement. Thus, it is difficult for the scheme to adapt to rapid changes in the amount of exploitable MLP and enlarge the IQ in a timely manner.

6.3 Exploiting MLP

Lebeck *et al.* proposed a scheme that efficiently uses the IQ by moving a load causing an LLC miss and the instructions that depend directly or indirectly on it to a special buffer, called the WIB (waiting instruction buffer) [26]. MLP can be exploited in a small IQ. Although the IQ is not enlarged in this scheme, other window resources must be enlarged. The other drawback of this scheme is that to exploit MLP fully, sophisticated compaction logic is required for the IQ to use the sparse vacant entries effectively in the small queue created by the instruction transfer. This compaction logic is extremely complex [27]. Note that the enlarged IQ only requires simple compaction logic, because inefficiency in terms of the capacity is tolerated. Moreover, Lebeck's scheme is less effective if the IQ is implemented by a circular buffer, which is more power-efficient than the shift register implementation. This is because even if an entry in the middle of the queue becomes vacant, it cannot be used for a succeeding instruction; only the tail entry is available for this purpose. In other words, for instruction transfer to the WIB to be useful, the transferring instruction must be at the tail. This situation obviously does not occur very frequently, and thus, the scheme cannot fully exploit MLP. Although we do not detail the implementation of IQ in this

paper, our scheme can be simply applied to a circular buffer implementation. If the IQ is enlarged, a large number of vacant entries are created behind the tail, and thus MLP can be fully exploited.

Srinivasan *et al.* extended the WIB for register files so that they also remain small [28]. In their organization, called *continual flow pipelines*, instructions release the mapped physical registers when they leave the IQ for the WIB, and registers are re-renamed when the instructions are reinserted into the IQ from the WIB. The drawback of this organization is that the releasing and re-naming of registers complicate the renaming hardware. The other drawback is that a large number of physical registers ($\#logical\ registers \times \#checkpoints$) must be reserved to avoid deadlock when reinserting instructions into the IQ from the WIB and re-naming, as described in [28]. For example, if $\#checkpoints = 8$ (i.e., the expected number of instructions between the mispredicted branches for a 512-instruction window is 64), then the number of reserved registers is 512. Although reducing the number of checkpoints also reduces the number of reserved physical registers, it also prevents deep speculation.

Brekelbaum *et al.* proposed a hierarchical IQ with a large pipelined queue and a small non-pipelined queue [29]. Instructions that become ready for issue in the large queue are issued with a pipeline delay, while older instructions that are not ready but are expected to be critical to program execution are moved to the small queue and issued later without any extra delay. The issue of the young non-critical instructions contributes to MLP exploitation, while that of the old critical instructions contributes to ILP. The first drawback of this scheme is that the logic required to move the unready old instructions to the small queue is complex. A second drawback is that the large queue does not contribute to the performance of compute-intensive programs, thereby also wasting power.

Mutlu *et al.* proposed a scheme called *runahead execution* [11]. Because this has been discussed in Sect. 5.10, the explanation is not repeated here. However, we would like to emphasize that runahead execution is only a partial alternative to a large instruction window; it can strictly exploit either ILP or MLP, but not both, because of the small instruction window.

Yamamoto *et al.* proposed a scheme called *two-step physical register deallocation* (TSD), which allows pre-execution for MLP exploitation [30], [31]. This scheme temporarily deallocates a physically register at the rename stage, and enables its reuse by another instruction temporarily. Such instructions are inserted into the issue queue, and wait for a register to become available. While waiting, if their source operands become available, they are issued and executed, causing pre-execution. Although the result is not allowed to be written to the temporarily allocated register during this pre-execution, it is passed to the dependent instructions via the bypass logic. Therefore, pre-execution can be performed continuously. This would enable memory accesses as early as would be possible given an infinite

number of physical registers. Thus, main many memory accesses can be overlapped if the memory accesses miss the LLC, thereby exploiting MLP. Tanaka *et al.* enhanced TSD by applying value prediction to pre-executed instructions to increase the number of pre-executed instructions [32], [33]. Although TSD can exploit MLP without enlarging the register file, the mechanism to inform the instructions waiting in the issue queue of the availability of the destination register complicates the issue queue.

7. Conclusion

In this paper, we proposed a solution to the tradeoff arising from enlarging the instruction window for aggressive out-of-order execution. In this tradeoff, a large window is beneficial for MLP exploitation but is harmful for ILP exploitation, whereas a small window is unable to exploit MLP. The solution we proposed is a scheme called dynamic instruction window resizing, which adaptively enlarges and pipelines the instruction window for MLP exploitation and shrinks and de-pipelines it for ILP exploitation, based on the prediction of the available parallelism, either ILP or MLP. Our scheme is very simple and accommodates existing processor architectures, and is thus very practical.

The results of our evaluations show that the proposed scheme is highly adaptive, and achieves performance that is as good as or better than the best performance achieved in a processor with fixed size resources. According to the results, our scheme achieves 21% better performance, with an extra cost of only 6% of the processor core, or only 3% of the entire processor chip area, compared with that of a conventional processor, thus achieving a significantly better cost/performance ratio, far exceeding that based on Pollack's law. Our scheme also achieves better energy efficiency (1/EDP).

Acknowledgments

The authors wish to thank T. Inagaki for his contributions to our study. This work is supported by a Grant-in-Aid for Scientific Research (C) (No. 25330057 and 25330057) from the Japan Society for the Promotion Science, and the VLSI Design and Education Center (VDEC), the University of Tokyo with the collaboration with Synopsys Inc.

References

- [1] N.P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers," Proc. 17th Annual International Symposium on Computer Architecture, pp.364–373, May 1990.
- [2] J.L. Baer and T.F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," Proc. 1991 Conference on Supercomputing, pp.176–186, Nov. 1991.
- [3] Y. Chou, "Low-cost epoch-based correlation prefetching for commercial applications," Proc. 40th Annual International Symposium on Microarchitecture, pp.301–313, Dec. 2007.
- [4] J.E. Smith and A.R. Pleszkun, "Implementation of precise interrupts in pipelined processors," Proc. 12th Annual International Symposium on Computer Architecture, pp.291–299, June 1985.
- [5] R.M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," IBM Journal of Research and Development, vol.11, no.1, pp.25–33, Jan. 1967.
- [6] Intel, P6 Family of Processors - Hardware Developer's Manual, Sept. 1998.
- [7] Y. Kora, K. Yamaguchi, and H. Ando, "MLP-aware dynamic instruction window resizing for adaptively exploiting both ILP and MLP," Proc. 46th Annual International Symposium on Microarchitecture, pp.37–48, Dec. 2013.
- [8] M. Johnson, Superscalar microprocessor design, Prentice Hall, Inc., Englewood Cliffs, New Jersey, USA, 1991.
- [9] H. Ando, Instruction-Level Parallel Processing – Processor Architecture and Compilers –, Corona Publishing Co., Ltd., Tokyo, Japan, 2005.
- [10] J.L. Hennessy and D.A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers, San Francisco, California, USA, 2011.
- [11] O. Mutlu, J. Stark, C. Wilkerson, and Y.N. Patt, "Runahead execution: An effective alternative to large instruction windows," Proc. Ninth Annual International Symposium on High-Performance Computer Architecture, pp.129–140, Feb. 2003.
- [12] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffner, and M. Tremblay, "Rock: A high-performance Sparc CMT processor," IEEE Micro, vol.29, no.2, pp.6–16, March/April 2009.
- [13] H.Q. Le, W.J. Starke, J.S. Fields, F.P. O'Connell, D.Q. Nguyen, B.J. Ronchetti, W.M. Sauer, E.M. Schwarz, and M.T. Vaden, "IBM POWER6 microarchitecture," IBM Journal of Research and Development, vol.51, no.6, pp.639–662, Nov. 2007.
- [14] K. Yamaguchi, Y. Kora, and H. Ando, "Evaluation of issue queue delay: Banking tag RAM and identifying correct critical path," Proc. 29th International Conference on Computer Design, pp.313–319, Oct. 2011.
- [15] <http://www.mosis.com/>
- [16] <http://www.eas.asu.edu/~ptm/>
- [17] N. Muralimanohar, R. Balasubramanian, and N.P. Jouppi, "CACTI 6.0: A tool to model large caches," HPL-2009-85, HP Laboratories, April 2009.
- [18] <http://www.simplescalar.com/>
- [19] 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. 42nd Annual International Symposium on Microarchitecture, pp.469–480, Dec. 2009.
- [20] M. Yuffe, M. Mehalel, E.K.J. Shor, T. Kurts, E. Altshuler, E. Fayneh, K. Luria, and M. Zelikson, "A fully integrated multi-CPU, processor graphics, and memory controller 32-nm processor," IEEE J. Solid-State Circuits, vol.47, pp.194–205, Jan. 2012.
- [21] F. Pollack, "New microarchitecture challenges in the coming generations of CMOS process technologies," Proc. 32nd Annual International Symposium on Microarchitecture, Nov. 1999.
- [22] A. Cristal, O.J. Santana, F. Cazorla, M. Galluzzi, T. Ramírez, M. Pericàs, and M. Valero, "Kilo-instruction processors: Overcoming the memory wall," IEEE Micro, vol.25, no.3, pp.48–57, May/June 2005.
- [23] D.H. Albonesi, R. Balasubramanian, S.G. Dropsho, S. Dwarkadas, E.G. Friedman, M.C. Huang, V. Kursun, G. Magklis, M.L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P.W. Cook, and S.E. Schuster, "Dynamically tuning processor resources with adaptive processing," Computer, vol.36, pp.49–58, Dec. 2003.
- [24] D. Ponomarev, G. Kucuk, and K. Ghose, "Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources," Proc. 34th Annual International Symposium on Microarchitecture, pp.90–101, Dec. 2001.
- [25] D. Folegnani and A. González, "Energy-effective issue logic," Proc. 28th Annual International Symposium on Computer Architecture,

- pp.230–239, June 2001.
- [26] A.R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, “A large, fast instruction window for tolerating cache misses,” *Proc. 29th Annual International Symposium on Computer Architecture*, pp.59–70, May 2002.
 - [27] J.A. Farrell and T.C. Fischer, “Issue logic for a 600-MHz out-of-order execution microprocessor,” *IEEE J. Solid-State Circuits*, vol.33, no.5, pp.707–712, May 1998.
 - [28] S.T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, “Continual flow pipelines,” *Proc. 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.107–119, Oct. 2004.
 - [29] E. Brekelbaum, J. Rupley, C. Wilkerson, and B. Black, “Hierarchical scheduling windows,” *Proc. 35th Annual International Symposium on Microarchitecture*, pp.27–36, Nov. 2002.
 - [30] A. Yamamoto, Y. Tanaka, H. Ando, and T. Shimada, “Data prefetching and address pre-calculation through instruction pre-execution with two-step physical register deallocation,” *Proc. Eighth Workshop on Memory Performance: Dealing with Applications, Systems and Architectures*, pp.41–48, Sept. 2007.
 - [31] A. Yamamoto, Y. Tanaka, H. Ando, and T. Shimada, “Two-step physical register deallocation for data prefetching and address pre-calculation,” *IPSJ Trans. Advanced Computing Systems*, vol.1, no.2, pp.34–46, Aug. 2008.
 - [32] Y. Tanaka and H. Ando, “Reducing register file size through instruction pre-execution enhanced by value prediction,” *Proc. 27th IEEE International Conference on Computer Design*, pp.238–245, Oct. 2009.
 - [33] Y. Tanaka and H. Ando, “Register file size reduction through instruction pre-execution incorporating value prediction,” *IEICE Trans. Inf. & Syst.*, vol.E93-D, no.12, pp.3294–3305, Dec. 2010.

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.

Yuya Kora received his B.E. and M.E. degrees from Nagoya University, Nagoya, Japan, in 2009 and 2011, respectively. From 2011 to 2013, he was with Rohm Co., Ltd., and he is currently with the Transportation Bureau, Nagoya.

Kyohei Yamaguchi received his B.E. and M.E. degrees from Nagoya University, Nagoya, Japan, in 2011 and 2013, respectively. Since then, he has been with Renesas Electronics Corporation.