MLP-Aware Dynamic Instruction Window Resizing for Adaptively Exploiting Both ILP and MLP

Yuya Kora[1]* Kyohei Yamaguchi[2]† Hideki Ando[3]
[1] Department of Computational Science, Nagoya University
[2,3] Department of Electrical Engineering and Computer Science, Nagoya University
Nagoya, Aichi, Japan
[2] kyoheister@gmail.com [3] ando@nuee.nagoya-u.ac.jp

ABSTRACT

It is difficult to improve the single-thread performance of a processor in memory-intensive programs because processors have hit the memory wall, i.e., the large speed discrepancy between the processors and the main memory. Exploiting memory-level parallelism (MLP) is an effective way to overcome this problem. One scheme for exploiting MLP is aggressive out-of-order execution. To achieve this, large instruction window resources (i.e., the reorder buffer, the issue queue, and the load/store queue) are required; however, simply enlarging these resources degrades the clock cycle time. While pipelining these resources can solve this problem, this leads to instruction issue delays, which prevents instruction-level parallelism (ILP) from being exploited effectively. As a result, the performance of compute-intensive programs is degraded dramatically.

This paper proposes an adaptive dynamic instruction window resizing scheme that enlarges and pipelines the window resources only when MLP is exploitable, and shrinks and de-pipelines the resources when ILP is exploitable. Our scheme changes the size of the window resources by predicting whether MLP is exploitable based on the occurrence of last-level cache misses. Our scheme is very simple and hardware change is accommodated within the existing processor organization, it is thus very practical. Evaluation results using the SPEC2006 benchmark programs show that, for all programs, our dynamic instruction window resizing scheme achieves performance levels similar to the best performance achieved with fixed-size resources. On average, our scheme produces a performance improvement of 21% in comparison with that of a conventional processor, with an additional cost of only 6% of the conventional processor core or 3% of the entire processor chip, thus achieving a significantly better cost/performance ratio that is far beyond the level that can be achieved based on Pollack's law. The evaluation results also show an 8% better energy efficiency in terms of 1/EDP (energy-delay product).

Categories and Subject Descriptors

C.1.0 [**Processor Architectures**]: General

General Terms

Design, Performance

Keywords

Memory-level parallelism, instruction-level parallelism, issue queue

1. INTRODUCTION

In the past, single-thread performance increased according to Pollack's law [21], which states that the performance improves in proportion to the square-root of the processor area. For the past few years, however, this has no longer been the case, and the performance improvement rate has slowed dramatically, despite ever increasing transistor budgets. New architectural techniques that use the increased transistor budget effectively are needed.

One of the reasons for this lack of performance improvement is the memory wall, which is the large speed discrepancy between the processors and the main memory. This severely limits the performance of a computer, because of the long latency in a load if a last-level cache (LLC) miss occurs. Conventional solutions to this problem have involved incorporation of a large cache and a hardware prefetcher [4, 12]. Unfortunately, a large cache is very expensive and several megabytes are not enough. Hardware prefetchers are generally inexpensive and are effective for regular access patterns, but they are ineffective with irregular access patterns. Although prefetchers for irregular access patterns have been reported in the literature, they require a very large correlation table comprising multiple megabytes [7].

Aggressive out-of-order execution is an effective alternative approach to this problem. This method significantly increases the number of in-flight instructions that are supported by the processor through extensive instruction window resources (i.e., reorder buffer (ROB), issue queue (IQ),

*Presently with Transportation Bureau, City of Nagoya
†Presently with Renesas Electronics Corporation

MICRO '46, December 7-11, 2013, Davis, CA, USA.

http://dx.doi.org/10.1145/2540708.2540713.

and load/store queue (LSQ)[1]). This allows parallel memory accesses by executing the cache-miss-causing loads as early as possible, and thereby reducing the effective memory latency. This type of parallelism is called memory-level parallelism (MLP).

One advantage of this approach is that data fetch is carried out by instruction execution, and not by prediction like a hardware prefetch, and thus the data fetch is accurate. Another advantage is that it can be implemented by applying a simple extension to a conventional superscalar processor. However, the downside is that the large resources adversely affect the clock cycle time. Although this can be solved by pipelining the resources, it prevents instruction-level parallelism (ILP) from being exploited effectively, mainly because of the enlarged IQ. Specifically, pipelining of the IQ makes it impossible to issue dependent instructions back-to-back, because the wakeup-select issue loop takes more than a single cycle to complete.

As explained above, there is a tradeoff involved in enlarging and pipelining the window resources for exploitation of ILP and MLP. In other words, large pipelined resources are effective for exploiting MLP, and are thus beneficial for memory-intensive programs or execution phases. However, they are harmful when exploiting ILP, which offers high performance in compute-intensive programs or phases.

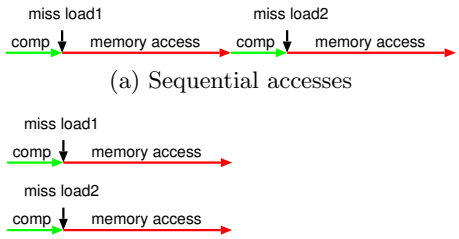
To solve this tradeoff, we propose *dynamic instruction window resizing*, which adapts the window size to the available parallelism (ILP or MLP). In this adaptation, as more exploitable MLP is predicted, the window resources are enlarged further, while the pipeline depth has increased at the same time. Conversely, if the prediction indicates that less MLP should be exploited, i.e., that ILP is more valuable for better performance, the window resources are shrunk, while simultaneously decreasing the pipeline depth. A larger window allows more MLP to be exploited, whereas a smaller window allows more ILP to be exploited. We can predict when MLP is exploitable by the occurrence of an LLC miss. Specifically, if an LLC miss occurs once, we predict that MLP can be exploited for a while thereafter. The rationale of this prediction is that the LLC misses are typically clustered with respect to time, and thus if one miss occurs, more misses are expected to occur soon afterwards. Conversely, we predict that MLP will not be exploitable once the memory latency has lapsed after the last LLC miss.

The remainder of this paper is organized as follows. Section 2 gives an overview of MLP, while Section 3 explains the tradeoffs in the instruction window size. Our dynamic instruction window resizing scheme is proposed in Section 4 and the evaluation results are presented in Section 5. Related work is described in Section 6 and our conclusions are given in Section 7.

2. OVERVIEW OF MEMORY-LEVEL PARALLELISM

Memory-level parallelism (MLP) is the type of parallelism that is associated with main memory accesses. Use of MLP reduces the total main memory access time that is included in the total execution time of a program. Because the time for a single memory access is at present a few hundred processor cycles, this saving is highly significant in memory-intensive programs.

[1]We assume an Intel P6-type architecture [11] 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.



(a) Sequential accesses

(b) Parallel accesses

Figure 1: Time sequence of sequential and parallel main memory accesses.

We explain how the execution time is reduced by exploiting MLP by referring to Figure 1, which shows the time sequence of the events in which two distantly separate loads, `load1` and `load2`, cause cache misses. Note that the light and dark lines (shown as green and red lines, respectively, in the color print) show the computation and the main memory accesses, respectively.

Figure 1(a) illustrates the case where two memory accesses occur sequentially. In this case, the time required for the two memory accesses simply extends the execution time of the program. Conversely, Figure 1(b) shows that `load1` and `load2` are executed in parallel. In this case, the two memory accesses occur in parallel, thereby saving one memory access time.

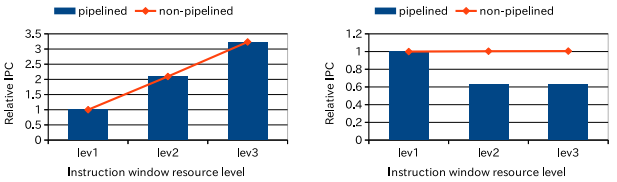
For example, if two-thirds of a program's execution time is used to perform main memory accesses, always parallelizing these two memory accesses (i.e., doubling the MLP) can decrease the overall execution time to two-thirds of the original time ($= \textit{computation time} + \textit{memory access time}/2 = 1/3 + (2/3)/2$), resulting in a speedup of 50%. In this way, if more cache-miss-causing loads can be executed within a short period, then the additional memory accesses can be parallelized, thus further reducing the execution time of the program.

One method to move the execution timing of multiple long-distant loads closer, as shown in Figure 1(b), is to increase the instruction window size. Because the instructions in the instruction window can be reordered, multiple loads can potentially be executed within a short period. The window size is determined by the resources of the ROB, IQ, and LSQ.

3. TRADEOFFS IN THE INSTRUCTION WINDOW SIZE

As described in Section 1, there is the following tradeoff with regard to the size of the instruction window when exploiting ILP and MLP.

(1) A large and pipelined instruction window resource is beneficial for exploiting MLP in memory-intensive programs. However, pipelining the IQ means that it is unable to issue dependent instructions back-to-back, thereby causing a reduction in ILP. This results in performance degradation in compute-intensive programs.



(a) libquantum (memory-intensive) (b) gcc (compute-intensive)

Figure 2: IPC for varying instruction window resource levels (i.e., resource size, pipeline depth).

(2) Conversely, a small and non-pipelined instruction window resource is beneficial for exploiting ILP in compute-intensive programs. However, this makes it difficult to overlap the memory accesses of cache-miss-causing loads, resulting in a deterioration in MLP. This means that the performance of memory-intensive programs is hardly improved.

Figure 2 shows the performance (IPC) for *libquantum* (a memory-intensive program) and *gcc* (a compute-intensive program) as an example to illustrate this tradeoff. The horizontal axis represents the *instruction window resource level*, which represents the size and the pipeline-depth of the various window resources as defined in Section 4. As the level increases, the size also increases and the pipeline depth increases or remains the same. The specific values for each level are listed in Table 2, while the configuration of the evaluated processor is given in Table 1. The bars represent the IPCs relative to those of the processor at level 1 (conventional processor), when the instruction window resource level was varied. The lines represent the relative IPC of an ideal processor, in which the window resources are enlarged but are not pipelined.

The bars in the two figures show the tradeoff, i.e. that the 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. The first is that the deterioration of ILP in the large window resources, which is caused by the pipelining of these resources, hardly affects the performance of *libquantum*, because the memory-access time dominates the execution time. As shown in Figure 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 shown in Figure 2(b), enlarging the window resources does little to increase IPC, even when the adverse effects of pipelining are removed (see the IPC for the ideal processor). That means that a small window size (the size of the IQ and the LSQ are 64, while that of the ROB is 128 at instruction window resource level 1) is sufficient for exploiting ILP.

Detailed evaluation results that highlight this tradeoff are presented in Section 5.3.

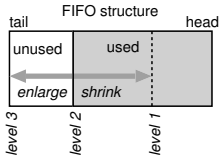


Figure 3: Resizing resources by changing the *level*.

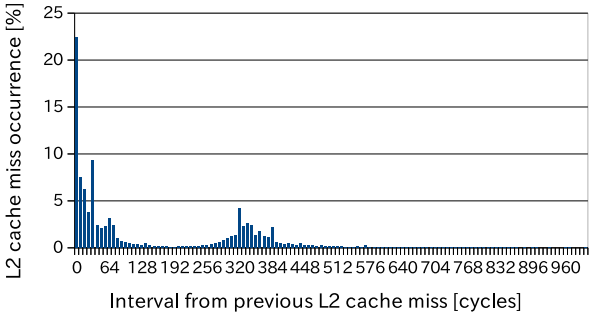


Figure 4: Histogram of L2 cache miss occurrences for miss intervals (*soplex*).

4. MLP-AWARE DYNAMIC INSTRUCTION WINDOW RESIZING

This section proposes dynamic instruction window resizing. In this study, we assume the Intel P6-type architecture to be the base architecture, where the instruction window resources are 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 Figure 3. How the circuit of a resource is pipelined depends on the circuit. In the Appendix, we present an example of how the IQ is pipelined. Other resources, including the IQ with different implementations, can also be pipelined easily. In addition, to prevent an increase in the delay and suppress the power consumed by the circuits in the unused region, the signals propagated to the unused region are gated, and precharging of the dynamic circuits in the unused region is disabled. It takes several cycles to resize resources. We assume 10 cycles are required for the evaluation presented in Section 5. This transition cycle penalty has little effect on performance, according to our evaluation (only 1.3% slowdown, even if the penalty increases to 30 cycles).

Here, we define an *instruction window resource level* (henceforth simply referred to as a *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, then the corresponding size also increases. 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

```
 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:   foreach 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_shinkable(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:   }
```

Figure 5: Algorithm for dynamic resource resizing.

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 the memory accesses decreases because of a phase change in the program execution. Figure 4 (previous page) shows the histograms of the L2 cache miss occurrences for the miss intervals in *soplex* with a bin size of 8 cycles (see the processor configuration in Table 1), as an example. As shown in the figures, the vast majority of the L2 cache misses occur within a short interval, although the average interval in instructions is 147. Also, note that there is another peak at around 300 cycles; this is because the 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 continuously 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 lapsed from the time at which the last L2 cache miss occurred. Specifically, the level of each window resource is reduced by one (if it is currently at level 1, the level does not change). Note that the shrinking of the window resources is delayed until the regions of the ROB, IQ, and LSQ that are to be removed by shrinking simultaneously become vacant.

4.2 Algorithm

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

In a cycle when an L2 cache miss occurs, the window resources are enlarged. This means that the instruction window resource level is increased by one (if it is already at its maximum, it does not change)(line 8). Then, the timing for shrinking the resources, `shrink_timing`, is set to the current cycle plus the memory latency to subsequently find the timing of shrinking (line 9). Also, `do_shrink`, the flag that instructs the shrinking of the resources, is cleared (line 10).

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

If the current resource level is greater than 1 and flag `do_shrink` is set (line 14), then 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 resources are shrunk. In other words, the resource level is decreased by one (line 17). 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. To wait for the instructions in the resources to move forward and for the regions that need to be removed to become vacant, resource allocation at the front-end is stopped (line 21).

Although the processor stalls for a while (we assume a period of 10 cycles in the evaluation presented in Section 5) at the level transition, we omit this from the description of the algorithm for simplicity.

Figure 6 illustrates how the resource level transition occurs, with the maximum level being 3. 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 thus the level is again increased by one to 3. At time *t2*, an L2 cache miss occurs again, but this time the level is not

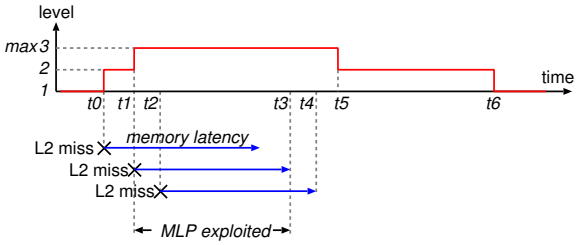


Figure 6: Resource level transitions with L2 cache miss occurrences.

increased, because it is already at the maximum. At time *t4*, the memory latency lapses after the last L2 miss at time *t2*. The level should now be decreased by one, but this is postponed until *t5* because the mechanism must wait for the regions to be removed to become vacant by halting the resource allocation. Another memory latency subsequently lapses, and the level is decreased further by one at time *t6*. At this time, the regions to be removed are vacant, and thus the resources are immediately shrunk. Note that MLP is exploited from time *t1* to *t3* (the memory accesses overlap during this period), while ILP is exploited before *t0* and after *t6*.

5. EVALUATION

We first describe the assumption of the size and the pipeline depth of the instruction window resources for several resource levels in Section 5.1. Then, we evaluate the performance when we introduce dynamic instruction window resizing in Sections 5.2 and 5.3, where 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 prefetcher proposed in [4]. 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 Sections 5.4 and 5.5, the energy efficiency and the cost/performance ratio are evaluated. We then discuss cache pollution because of deep speculation that may be caused by the large window in Section 5.6. Finally, in Section 5.7, we compare the performance of the proposed dynamic instruction window resizing method with *runahead execution*, which is a scheme that effectively exploits both ILP and MLP, and is used in commercial processors (the Sun Rock processor [6] and the IBM Power6 [13]).

5.1 Size and Pipeline Depth of Window Resources

In the following evaluation, we assume that a processor physically has window resources that are four times larger than those in the base processor, and that three levels can be selected with our dynamic window resizing scheme, as shown in Table 2. The physical sizes that we assume here are not unrealistic, when considering current LSI technology; the increased area is only 3% of the area of an entire processor chip of the Intel Sandy Bridge as presented in Section 5.5.

With regard to the pipeline depth of the IQ, we determined it by evaluating the delay by an HSPICE circuit simulation after drawing the layout, based on a study in the literature [25]. In this simulation, we assumed MOSIS design rules [1] for 32nm LSI technology, and used the predictive transistor model [2] 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 is pipelined so that the clock cycle time does not increase. If we let the delay of the IQ at level L be $D(L)$, and then the pipeline depth of the IQ at level L is derived by $\left\lceil \frac{D(L)}{D(1)} \right\rceil$. Note that, as listed in Table 2, the IQ pipeline depths at levels 2 and 3 are the same (2 stages).

The pipeline depth of the ROB is not a concern with regard to allocating and committing, because this does not affect the IPC. However, the pipeline depth for reading the register fields does affect the IPC, because it changes the branch misprediction penalty. We obtained the delay of the register field read using CACTI 6.5 [16], and then determined the pipeline depth.

The pipeline depth of the LSQ is 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 add the assumption of the penalty at the level transition. This transition cycle penalty has little effect on the performance according to our evaluation (only 1.3% slowdown, even if the penalty increases to 30 cycles).

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	2MB, 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	3
IQ	entries	64	160	256
	pipeline depth	1	2	2
ROB	entries	128	320	512
	pipeline depth	1	2	2
LSQ	entries	64	160	256
	pipeline depth	1	2	2
Level transition penalty		10 cycles		

5.2 Environment for Performance Evaluation

We built a simulator based on the SimpleScalar Tool Set version 3.0a [23] to evaluate the performance. The instruc-

Table 3: Benchmark programs and their load latency.

program	type	average load latency (cycles)	category
hmmer	int	15	
libquantum	int	247	
mcf	int	52	
omnetpp	int	42	
xalancbmk	int	74	memory-
GemsFDTD	fp	32	intensive
lbm	fp	14	
leslie3d	fp	72	
milc	fp	12	
soplex	fp	36	
sphinx3	fp	51	
astar	int	7	
bzip2	int	3	
gcc	int	6	
gobmk	int	3	
h264ref	int	3	
perlbench	int	4	
sjeng	int	2	
bwaves	fp	2	compute-
cactusADM	fp	5	intensive
calculix	fp	6	
dealII	fp	2	
gamess	fp	2	
gromacs	fp	5	
namd	fp	3	
povray	fp	2	
tonto	fp	2	
zeusmp	fp	6	

tion 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. The benchmark programs and their average load latency are listed in Table 3. The fourth column categorizes the programs in terms of whether they are memory-intensive or compute-intensive, when the threshold of the average load latency is 10 cycles. This category is used to show the following evaluation results. We simulated 100M instructions after the first 16G instructions were skipped with the *ref* inputs.

In the following sections, we only show the results of selected programs (eight of the memory-intensive programs and six of the compute-intensive programs) to save space. We also show the "GM mem", the "GM comp", and the "GM all", which are the geometric means of all memory-intensive programs, all compute-intensive programs, and all programs listed in Table 3, including the non-selected programs.

5.3 Performance

We evaluated the performances of the following three models:

- **Fixed size model**: The size of the window resources is fixed during execution and the resources are pipelined, as shown in Table 2. Instruction issues from the IQ and the LSQ take multiple cycles of pipelining. Also, an extra branch misprediction penalty is imposed for the extra delay from the enlarged IQ and from reading the enlarged ROB.

- **Dynamic resizing model**: Each of the window resources has a physical maximum size (i.e., the size at resource level 3), but is resized dynamically using our scheme. Both the issue delay and the extra branch misprediction penalty are imposed, as per the fixed model, depending on the pipeline depth of the resources.

- **Ideal model**: The size of each window resource is identical 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 IPCs for resource levels 1 to 3 in the fixed size model (labeled "Fix"). The bar on the right shows the relative IPC for the dynamic resizing model (labeled "Res"). The line shows the relative IPCs 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) present the geometric means in all memory-intensive programs, all compute-intensive programs, and all programs of SPEC2006 that we evaluated, respectively, including the non-selected programs.

The fixed size model achieves the best performance at level 3 for the memory-intensive programs. MLP is exploited aggressively with the large window resources and there is a significant improvement in performance as the resource levels increase.

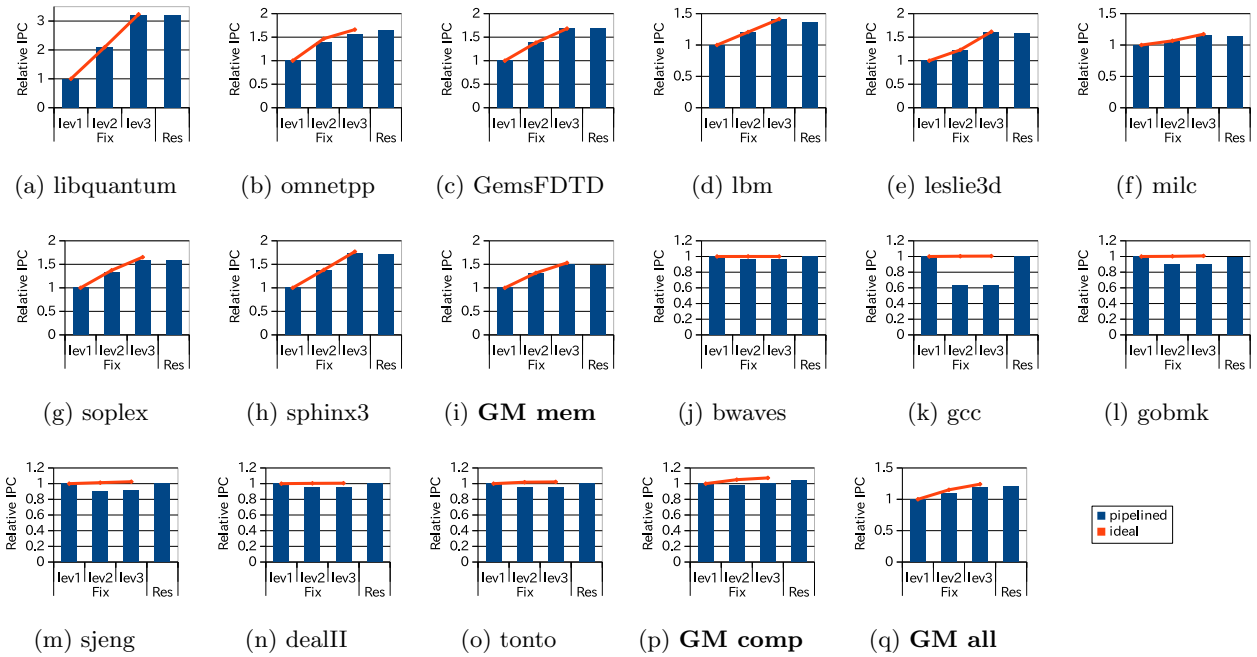
Conversely, for the compute-intensive programs, the performance of the fixed size model is not so sensitive to the level, and even decreases as the level increases in several programs. The deterioration in ILP caused by pipelining of the resources is more severe than the benefits of greater MLP exploitation.

Although the best resource level differs in the fixed size model, depending on the program, the dynamic resizing model achieves a performance that is as good as the best performance for levels 1 to 3 of the fixed size model. This implies good adaptability in our dynamic instruction window resizing scheme. In terms of the geometric mean, the speedup over the base is 48% (GM mem), 4% (GM comp), and 21% (GM all) for all memory-intensive, all compute-intensive, and all programs, respectively.

Compared with the best performance for levels 1 to 3 of the ideal model, which has no drawbacks because of enlargement, there is no significant degradation in the performance of the dynamic resizing model for any program, as we confirmed in Figure 7. In terms of the geometric mean, the dynamic resizing model is inferior to the ideal model by only 3% for the memory-intensive programs, the compute-intensive programs, and for all programs. This further indicates that our dynamic resizing scheme is highly adaptable.

Figure 8 shows the percentages of the cycles where the window resources were configured to particular levels in the dynamic resizing model. Unsurprisingly, resource level 1 is generally selected most in the compute-intensive programs, whereas resource level 3 is generally selected most in the 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 3 in the fixed



(a) libquantum (b) omnetpp (c) GemsFDTD (d) lbm (e) leslie3d (f) milc

(g) soplex (h) sphinx3 (i) **GM mem** (j) bwaves (k) gcc (l) gobmk

(m) sjeng (n) dealII (o) tonto (p) **GM comp** (q) **GM all**

Figure 7: IPC normalized by the base.

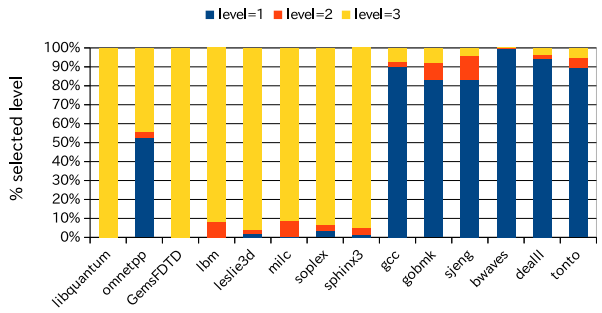


Figure 8: Percentages of cycles where the window resources were configured to particular levels.

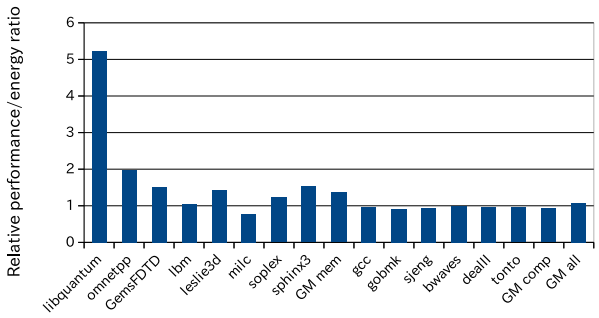


Figure 9: Energy efficiency (1/EDP).

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 mixed well, and the 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 energy (which is proportional to 1/EDP (energy-delay product)), using the McPAT [15] and assuming 32nm LSI technology and a temperature of 350K. Figure 9 shows the results. The vertical axis is the IPC per energy normalized based on that of the base processor.

As observed in the figure, dynamic resizing performs significantly better than the base for the memory-intensive programs. This is because although the large window resources consume more power, the performance is significantly improved, as shown in Figure 7. In particular, the improvement is dramatic in *libquantum* (423%).

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

The average improvements are 36%, −8%, and 8% for the memory-intensive programs, the compute-intensive programs, and for all 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, to that of a single core, and to that of an

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 by Pollack's law*1	3%
	augmented L2 cache	1%

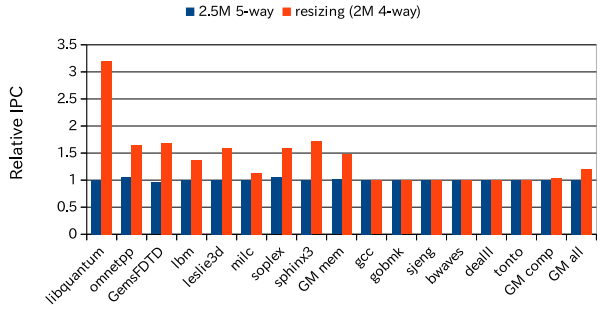


Figure 10: Comparison with the enlarged L2 cache model.

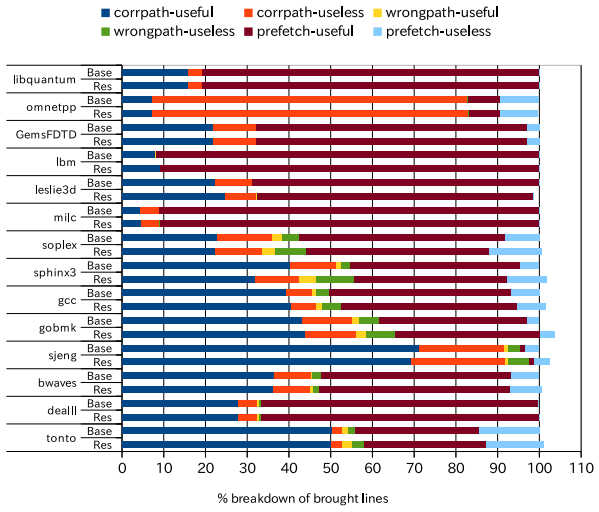


Figure 11: Breakdown of L2 cache lines brought in terms of touched (useful) or not (normalized based on the number of L2 cache lines brought in the base processor).

entire Intel Sandy Bridge chip [26], respectively. The areas of the base core, a single core of the Sandy Bridge, and the entire Sandy Bridge chip are 25mm^2, 19mm^2, and 216mm^2, respectively. Note that the Sandy Bridge has four cores and is fabricated using 32nm LSI technology. In the calculations of the additional costs to the Sandy Bridge chip, we assumed that our scheme would be used with these 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 2MB. 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 a good cost/performance ratio that far exceeds that based on Pollack's law. This indicates that our scheme is significantly 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 of 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, 5-way L2 cache. Because the area of the 2MB, 4-way cache (the base configuration) is 8.6mm^2 (calculated by McPAT), the increased cost of a 2.5MB, 5-way cache is approximately 1.3× greater than the additional cost when using our scheme. The evaluation results shown in Figure 10 confirm that the average IPC of a processor with a 2.5MB, 5-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 Cache Pollution with Deep Speculation

A large window may cause deep speculation. The question is sometimes asked whether loads on mispredicted wrong paths bring many useless (i.e., never touched) data into the L2 cache and, as a result, pollute the cache and waste power with useless replacements.

Figure 11 shows whether the lines brought into the L2 cache were useful or not for each of the base and dynamic resizing models. Each bar, which is normalized based on the number of lines brought by the loads in the base model, is broken into six classes with combinations of each item in the following two groups:

- [*corrpath* | *wrongpath* | *prefetch*]: A line was brought by a load on a correct path, wrong path, or prefetch, respectively.
- [*useful* | *useless*]: A line was touched or NOT touched by a load on a correct path.

First, not many lines are brought by loads on the wrong path. This is because, in general, the average number of instructions between the adjacent mispredicted branches is not very small when compared with the window size, as listed in Table 5, especially in the memory-intensive programs. Second, although the number of useless lines brought by loads on wrong paths and prefetch is higher in the dynamic resizing model than in the base model in several programs, the ratio in comparison to the total number of lines brought in is small. Finally, the increase in the total number of lines brought in the resizing model compared with that in the base model is very small. In conclusion, the cache pollution caused by speculation is limited, and the effect on the power consumption of the useless replacements in the L2 cache is very small.

Although we do not show the results for the L1 D-cache for the space limit, the pollution is very small, as in the L2 cache.

Table 5: Average number of committed instructions between adjacent mispredcted branches.

mem-intensive	# of insts	comp-intensive	# of insts
libquantum	3703704	gcc	5323
omnetpp	178	gobmk	71
GemsFDTD	10064	sjeng	116
lbm	32830	bwaves	169
leslie3d	1608	dealII	1294
milc	3448276	tonto	423
soplex	154		
sphinx3	327		

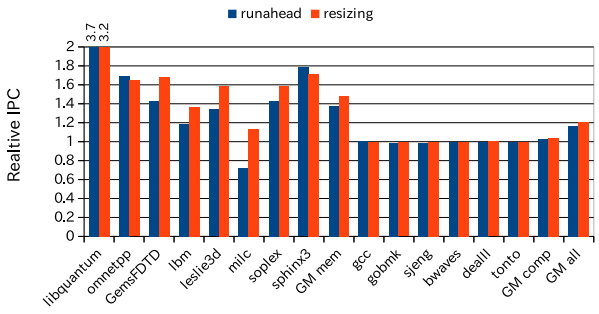


Figure 12: Performance comparison of the dynamic resource resizing scheme with runahead execution.

5.7 Comparison with Runahead Execution

In this section, we compare the performance of *runahead execution* [18] with that of our scheme. Several performance enhancing techniques among those proposed in [17] are introduced into our runahead simulator in addition to the basic scheme for runahead execution.

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 of accommodating existing processor architectures. In fact, it has been adopted in commercial processors, including the Sun Rock processor [6] 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 normal mode is resumed, and execution restarts from the checkpoint. The re-executed load hits the cache this time.

Figure 12 compares the performance of our dynamic resizing scheme with that of runahead execution. The vertical axis represents the IPC normalized based on that of the base processor. The configuration of the runahead execution processor is the same as that of the base processor, except that it has two checkpointing register files for integer and floating-point and a 2-port, 512-byte, 4-way runahead cache for resolution of the memory dependences during the runahead mode, with a configuration that is based on that in [18]. We assume that no cycle penalty is imposed for checkpointing and resume to 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 than runahead execution for memory-intensive and compute-intensive programs on average, respectively. This speedup arises from the difference that runahead execution cannot perform computations while MLP exploitation 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 the computation is almost stalled, 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 the computation.

Note that the IPC for runahead execution is lower than that of the base in *milc*. This is caused by useless runahead occurring, where a few L2 misses occur while in the runahead mode. This is equivalent to stalling execution without MLP exploitation. In general, this situation sometimes occurs in programs where the L2 misses do not occur so frequently and are not clustered. This situation is also disadvantageous to our dynamic instruction resizing scheme, but the performance loss is not so high (ILP is lost) when compared to 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, which uses the *runahead cause status table* [18] to predict the usefulness of the transition to the runahead mode based on the past usefulness of runahead for each L2 missed load, and suppresses the transition to the runahead mode if it is predicted to be useless. However, the prediction is difficult, and useless runahead cannot always be eliminated very well, depending on the programs. While the performance of runahead execution is sensitive to the degree of L2 miss frequency and clustering, our dynamic instruction window resizing scheme is rather more tolerant.

6. RELATED WORK

In this section, we first discuss the approach to MLP exploitation using a large instruction window. We then review schemes for resource resizing, and finally consider the 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 [8]. 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. Also, because of the rapidly rising transistor budget and high levels of integration on processor chips, the area overhead has now become acceptable; the overhead is only 3% of the entire processor chip area, as shown in Section 5.5. Also, the energy efficiency is not a problem; the efficiency is 8% better than that of a conventional processor, as described in Section 5.4.

6.2 Resizing Resources

Albonesi *et al.* presented a comprehensive survey of studies on resource resizing to improve power efficiency [3]. However, our policy for resource resizing is completely different to that proposed in any previous studies in the literature. Most policies focus on the demand for resources. On the other hand, 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 the performance.

Ponomarev *et al.* proposed a resizing scheme that focused on the occupation of the IQ [22]. 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 to be greater than the average issue rate, the IQ eventually become 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 the performance [9]. This scheme periodically counts the number of committed instructions in the last section of a predetermined size. If the number is less than a 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.

Petoumenos *et al.* proposed an IQ resizing scheme that takes into account MLP [20]. Their scheme basically uses an ILP-aware resizing scheme like that proposed in [9], but its resizing decision is overridden by their proposed MLP-aware scheme. Their MLP-aware scheme first measures *MLP-distance*, which is the number of dispatched instructions between LLC missed instructions, and associates this number with the instruction trace in the instruction window. If the trace is fetched again, the scheme obtains the associated MLP-distance, and resizes the IQ to be large so that instructions that are distant with the MLP-distance can be held simultaneously in the IQ. A difference in comparison with our study is that their scheme is history-based, but our scheme takes advantage of the characteristic of LLC miss occurrences and is consequently simpler and cheaper. The other difference is that our study spans a very large pipelined instruction window. In other words, we discussed the trade-off involved in enlarging and pipelining the instruction window. However, their study is limited within the conventional single-cycle instruction window.

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) [14]. MLP can be exploited in a small IQ. However, because the IQ is small, sophisticated compaction logic is required for the IQ to use the existing sparse vacant entries effectively in a queue. This compaction logic is extremely complex. Note that the enlarged IQ either does not require compaction logic or only requires simple compaction logic. Also, instructions must move between the IQ and the WIB, adding pressure to the issue and the dispatch bandwidth.

Srinivasan *et al.* extended the WIB for the register files so that they also remain small [24]. In their organization, called the *continual flow pipeline*, instructions release the mapped physical registers when they leave the IQ for the WIB. The drawback of this organization, besides that stated with regard to the WIB, 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-renaming. For example, if $\#checkpoints = 8$ (i.e., the expected number of instructions between the mispredicted branches for 512-instruction window is 64), then the number of reserved registers is 512. Although reducing the number of checkpoints also reduces the numbers 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 [5]. Instructions that become ready to be issued in the large queue are issued with a pipeline delay, while older instructions that are not ready but that 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, and also wastes power.

Mutlu *et al.* proposed a scheme called runahead execution [18]. Because this has been discussed in Section 5.7, the explanation is not repeated here. However, we would like to emphasize that runahead execution is only a partial alternative to the large instruction windows; it can strictly exclusively exploit ILP or MLP because of the small instruction window.

7. CONCLUSION

In this paper, we proposed a dynamic instruction window resizing scheme that exploits ILP and MLP adaptively during execution, based on prediction of the available parallelism based on the occurrence of LLC misses. Our scheme is

very simple and accommodates existing processor architectures, and is thus very practical. The results of our evaluations show that our scheme is highly adaptive, and achieves performance that is as good 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 area cost of only 6% of a processor core, or only 3% of the entire processor chip area, compared to that of a conventional processor, thus achieving a significantly better cost/performance ratio to far exceed that based on Pollack's law. Our scheme also achieves better energy efficiency (1/EDP).

Acknowledgments

The authors thank our shepherd, Prof. T. M. Conte, and anonymous reviewers for their useful comments. The authors also thank T. Inagaki for contributions to our study. This work is supported by the Ministry of Education, Culture, Sports, Science and Technology Grant-in-Aid for Scientific Research (C)(No. 25330057), and VLSI Design and Education Center (VDEC), the University of Tokyo with the collaboration with Synopsys Inc.

References

[1] `http://www.mosis.com/`.

[2] `http://www.eas.asu.edu/~ptm/`.

[3] D. H. Albonesi, R. Balasubramonian, 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*, 36:49–58, December 2003.

[4] J. L. Baer and T. F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 Conference on Supercomputing*, pages 176–186, November 1991.

[5] E. Brekelbaum, J. Rupley, C. Wilkerson, and B. Black. Hierarchical scheduling windows. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 27–36, November 2002.

[6] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance Sparc CMT processor. *IEEE Micro*, 29(2):6–16, March/April 2009.

[7] Y. Chou. Low-cost epoch-based correlation prefetching for commercial applications. In *Proceedings of the 40th Annual International Symposium on Microarchitecture*, pages 301–313, December 2007.

[8] 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*, 25(3):48–57, May/June 2005.

[9] D. Folegnani and A. González. Energy-effective issue logic. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 230–239, June 2001.

[10] M. Goshima. *Research on high-speed instruction scheduling logic for out-of-order ILP processors*. PhD thesis, Kyoto University, 2004.

[11] Intel. *P6 Family of Processors - Hardware Developer's Manual*, September 1998.

[12] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[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*, 51(6):639–662, November 2007.

[14] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 59–70, May 2002.

[15] 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. In *Proceedings of the 42nd Annual International Symposium on Microarchitecture*, pages 469–480, December 2009.

[16] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A tool to model large caches. HPL-2009-85, HP Laboratories, April 2009.

[17] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 370–381, June 2005.

[18] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An effective alternative to large instruction windows. In *Proceedings of the Nineth Annual International Symposium on High-Performance Computer Architecture*, pages 129–140, February 2003.

[19] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[20] P. Petoumenos, G. Psychou, S. Kaxiras, J. M. C. Gonzalez, and J. L. Aragon. MLP-aware instruction queue resizing: The key to power-efficient performance. In *Proceedings of the 23rd International Conference on Architecture of Computing Systems*, pages 113–125, February 2010.

[21] F. Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, November 1999.

[22] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 90–101, December 2001.

[23] `http://www.simplescalar.com/`.

[24] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, October 2004.

[25] K. Yamaguchi, Y. Kora, and H. Ando. Evaluation of issue queue delay: Banking tag RAM and identifying correct critical path. In *Proceedings of the 29th International Conference on Computer Design*, pages 313–319, October 2011.

[26] 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 Journal of Solid-State Circuits*, 47:194–205, January 2012.

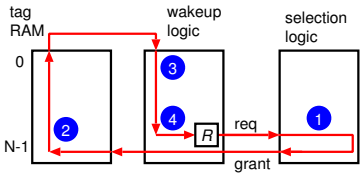


Figure 13: Critical path of issue queue.

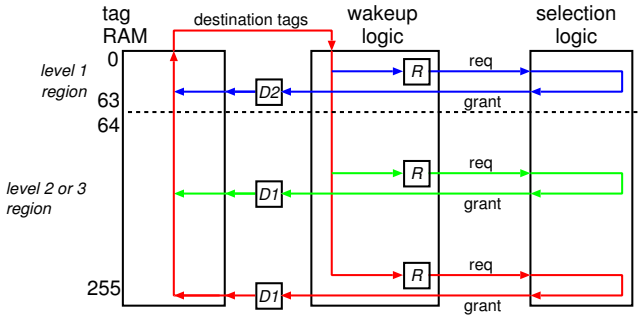


Figure 14: Two-stage pipelining of issue queue.

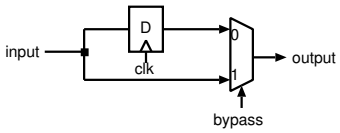


Figure 15: Circuit of *D2*.

APPENDIX

A. PIPELINING ISSUE QUEUE

This section describes how the issue queue is pipelined.

In this study, we assume an IQ with a CAM organization [19]. The IQ comprises the wakeup logic, selection logic, tag RAM, and payload RAM. The wakeup logic is a one-dimensional array, where each entry holds the tags of two source operands and ready flags that indicate a data dependence state (resolved or not) for a corresponding instruction. If both data dependences are resolved, an issue request is sent to the selection logic. The selection logic grants certain requests after considering resource constraints. The grant signals are sent to the payload RAM, which outputs information regarding the issued instructions. The signals are also sent to the tag RAM and the destination tags are read. These tags are broadcast to the wakeup logic to update the ready flags.

A.1 Circuit of Issue Queue

We explain the circuit of the wakeup logic, selection logic, and tag RAM shortly. See [25] for the details.

Each entry of the wakeup logic is composed of 1) SRAM cells that hold source operand tags, 2) comparators of the issue width that compare each source operand tag with a destination operand tag broadcast, 3) OR gates that OR the output of the comparators, 4) SR flip-flops (FFs) for ready bits that hold the output of the OR gate, and 5) an AND gate that ANDs two ready bits and outputs the issue request to the selection logic.

The selection logic we assume is implemented by prefix-sum logic, which calculates the cumulative sum of issue requests [10]. If the $(i-1)$-th output is less than the issue width and the i-th issue request is true, the request is granted. The adder used in the prefix-sum logic is not normal, but is specialized for the selection logic to reduce the delay. The selection logic outputs grant signals of the issue width to the tag RAM.

The tag RAM consists of SRAM without the address decoder. It has multiple ports of the issue width, with grant lines of the issue width per entry from the selection logic directly connected to the wordlines.

A.2 Pipelining

The critical path of the IQ starts at the wakeup logic and proceeds via the selection logic to the tag RAM before returning to the wakeup logic. As shown in Figure 13, the critical path traverses the following path: 1) it starts from the FF denoted as R, which holds the ready bit in the last entry; 2) an issue request signal is sent to the selection logic; 3) an issue grant signal is sent to the tag RAM via the selection logic (mark (1)); 4) a destination tag is read through the bitline of the tag RAM (mark (2)); 5) it is driven from the first to the last entry of the wakeup logic (mark (3)); 6) tag comparison is carried out in the last entry (mark (4)); and 7) the signal reaches the FF with the ready bit.

Figure 14 shows where pipeline registers (*D1* and *D2*) are inserted for the IQ with 256 entries, the size at the level 3, where the IQ is pipelined with two stages. As shown in the figure, the pipeline registers are inserted at the end of the grant signals (immediately before the signals are input to the tag RAM). Here, the FF denoted as *D1* used in the level 2 or 3 region is a normal D-FF, but the FF denoted as *D2* used in the level 1 region is a D-FF with a multiplexer, of which organization is explained later. According to our HSPICE circuit simulation, assuming MOSIS design rules for 32nm LSI technology, and used the predictive transistor model [2], the delays of the two paths, R to *D1* and *D1* to R, which collectively constitute the critical path in the non-pipelined IQ, are 0.78 and 0.96 times the clock cycle time, respectively. Note that the clock cycle time is assumed to be determined by the delay of the IQ with 64 entries, the size at level 1.

As previously described, the *D2* is a D-FF with a multiplexer. This organization is required because the *D2* must work as a D-FF when the instruction window resource level is 2 or 3 (i.e., the pipeline depth is two), but it must work as merely a wire logically when the level is 1 to operate in a single cycle. To be able to configure the circuit between these two modes, the *D2* is organized as shown in Figure 15. If the control signal *bypass* is 0, the *D2* works as a D-FF; if it is 1, the input signal skips the D-FF and just goes to the output.