Performance Improvement by Prioritizing the Issue of the Instructions in Unconfident Branch Slices

Hideki Ando

Nagoya University

ando@nuee.nagoya-u.ac.jp

Abstract—Single-thread performance has hardly improved for more than a decade. One of the largest problems for performance improvements is branch misprediction. There are two approaches to reduce the penalty caused by this. One is to reduce the *frequency* of misprediction, and the other is to reduce the *cycles* consumed because of misprediction. Improving branch predictors is the former approach, and many studies on this topic have been done for several decades. However, the latter approach has been rarely studied. The present paper hence explores the latter approach.

The cycles consumed because of misprediction are divided into the following two parts. The first part is the *state recovery penalty*, which consists of cycles consumed for rolling back the processor state. The second part is the *misspeculation penalty*, which are cycles consumed during useless speculative execution from the fetch of a mispredicted branch until the completion of the branch execution. We focus on reducing the misspeculation penalty. For this, we propose a scheme called *PUBS*, which allows the instructions in *unconfident branch slices* to be issued with highest priority from the issue queue (IQ). Here, a branch slice is a set consisting of a branch and the instructions this branch directly or indirectly depends on, and we call the branch slice unconfident if the associated branch prediction cannot be sufficiently trusted. By issuing instructions in unconfident branch slices as early as possible, the wait cycles of these instructions in the IQ are minimized and thus the misspeculation penalty is minimized. Our evaluation results using SPEC2006 benchmark programs show that the PUBS scheme improves the performance of the programs with difficult branch prediction by 7.8% on average (a maximum of 19.2%) using only 4.0KB hardware cost.

I. INTRODUCTION

Background: The improvement of single-thread performance has been sluggish for more than a decade. The largest reason is that Dennard scaling has ended. During this period, many computer architects have lost interest in core microarchitecture, and few studies on this topic have been done. This academic trend also has made single-thread performance sluggish.

Meanwhile, a new computer device, the smartphone, appeared in 2007. Since then, the smartphone market has rapidly grown, and now has become the largest market in all of computer history; the market size is seven times larger than that of the PC [1]. In smartphones, a short response time is important for good user experience, and thus high single-thread performance is required.

Against this background, restudying the microarchitecture for high single-thread performance has become important.

In particular, studies regarding the issue queue (IQ), which significantly affects performance, are important.

Branch misprediction penalty: In superscalar processors, instructions are speculatively executed based on branch prediction for high performance. However, once a branch is mispredicted, the performance is significantly degraded because the penalty imposed by misprediction is large.

In general, the total branch misprediction penalty during program execution is proportional to the frequency of misprediction and the penalty cycles per misprediction (i.e., branch misprediction penalty). Therefore, there are two approaches to reduce the performance degradation caused by branch misprediction. One approach is to reduce the misprediction frequency. Improving the accuracy of branch prediction is categorized as this approach. Branch predictors have been studied for several decades, but are extensively studied even at present [2], because the performance degradation due to misprediction is still serious, even with a modern branch predictor. The other approach is to reduce the branch misprediction penalty. However, few studies on this approach have been carried out. The present paper proposes a scheme that uses this approach.

The branch misprediction penalty is divided into two penalties: *state recovery penalty* and *misspeculation penalty*. The state recovery penalty is the clock cycles consumed for flushing the pipeline and recovering the processor state. In contrast, the misspeculation penalty consists of the clock cycles uselessly consumed for speculatively executing the instructions succeeding the mispredicted branch. Because these instructions are flushed later when the dependent branch is found to be mispredicted, the execution of these instructions is useless. These useless cycles are those from the fetch of the mispredicted branch until the completion of this branch execution. The pipeline is substantially stalled during these cycles. The paper focuses on reducing this misspeculation penalty.

The misspeculation penalty includes 1) the cycles during which a mispredicted branch flows down the front-end pipeline, 2) the waiting cycles spent in the IQ until the dependence is resolved and the branch is selected to be issued, and 3) the execution cycles of the branch. Given a pipeline structure, 1) and 3) cannot be reduced. However, 2) can be reduced by an architectural scheme for the IQ. The present paper improves performance by reducing the waiting cycles in the IQ by issuing the instructions that belong to *unconfident branch slices* as early as possible. Here, a branch slice is

defined as a set that consists of a branch and the instructions this branch directly or indirectly depends on. An unconfident branch slice is a branch slice with the associated branch that cannot be sufficiently trusted.

The IQ schedules instructions for execution by selecting and issuing instructions from the ready-to-execute instructions every cycle based on the priority of the instructions. The select logic included in the IQ is responsible for this task. Because the critical path of the IQ circuit is one of the critical paths of the processor, this circuit must be simple. Therefore, the select logic considers a simple priority setting policy, which is *position-based priority*, where higher priority is given to instructions that are closer to the head of the queue. This simple priority policy is problematic in terms of reducing misspeculation penalty, because instructions are selected independently of whether the instructions are those in unconfident branch slices or not. For example, if a branch slice is composed of a dependent chain of five instructions, and the issue of each instruction is delayed by an additional one cycle due to issue conflict, the misspeculation penalty is then increased by five cycles. To minimize the misspeculation penalty, the highest priorities must be given to instructions in unconfident branch slices.

Proposal: In the present paper, we propose the following scheme, which we call *prioritizing unconfident branch slices* (PUBS).

- *Predicting unconfident branch slices*: The PUBS scheme links each instruction in a branch slice to the prediction confidence of the branch associated with this branch slice. Links are constructed by tracking backward the dataflow that ends in a branch at the decode stage. By looking up the prediction confidence using the links, each instruction determines whether it belongs to an unconfident branch slice or not.
  To do this, we prepare two tables called *conf_tab* and *brslice_tab*. Each entry of the conf_tab is associated with a branch, and holds a counter that represents the prediction confidence of the corresponding branch. The confidence is learned using past prediction correctness. In contrast, each entry of brslice_tab is associated with an instruction in a branch slice, and holds the pointer to the conf_tab entry of the associated branch.
- *Prioritizing the issue of the instructions in unconfident branch slices*: We reserve a small number of entries at the head of the IQ, called *priority entries*. Instructions predicted to belong to unconfident branch slices are dispatched (i.e., written) to the priority entries. As previously described, instructions in the head entries are given the highest priority for issue. Therefore, these instructions are issued with highest priority, and consequently, the misspeculation penalty is minimized.

The remainder of the paper is organized as follows. Section II defines the misspeculation penalty and branch slices. The PUBS scheme is proposed in Section III. Section IV explains how we reduce the cost for PUBS. The evaluation
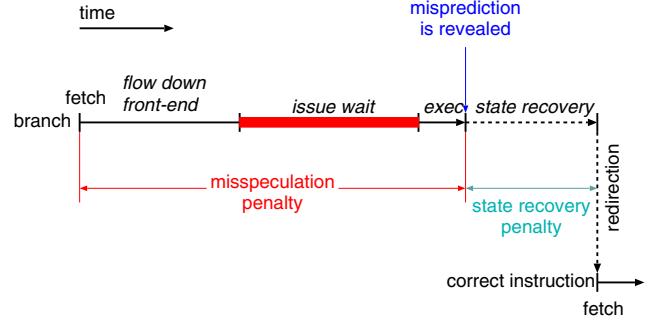

Fig. 1. Timeline of the flow of a mispredicted branch.

results are presented in Section V. Section VI describes related work. Finally, our conclusions are presented in Section VII.

II. DEFINITIONS OF MISSPECULATION PENALTY AND BRANCH SLICES

We define the misspeculation penalty and branch slices in this section.

A. Misspeculation Penalty

Figure 1 shows a chart of the timeline from the cycle where a mispredicted branch is fetched until the cycle where the instruction fetch is redirected toward the correct path. The branch first flows down the front-end pipeline, and is then dispatched to the IQ. In the IQ, the branch waits for several cycles until the instructions this branch directly or indirectly depends on are executed. Finally, the branch is issued, executed, and the misprediction is revealed. Then, the processor state is recovered and the instruction fetch is redirected to fetch the instruction on the correct path.

In this process, we call the *misspeculation penalty* the clock cycles from the fetch until the end of the execution of the branch, while we call the *state recovery penalty* the clock cycles used for recovering the processor state. In this paper, we reduce the misspeculation penalty as much as possible by reducing the waiting cycles in the IQ.

B. Branch and Computation Slices

Branch and computation slices are defined in a dataflow graph. We explain them using the example of the dataflow graph shown in Figure 2, where the only red circle represents a branch and the others are non-branch instructions.

A branch slice is defined as a sub-graph of the given dataflow graph, which includes a branch as a leaf and the instructions the branch directly or indirectly depends on. In the example shown in Figure 2, the sub-graph surrounded by a red dashed line is a branch slice.

In contrast, a computation slice is defined as a sub-graph, which includes an instruction other than a branch as a leaf and the instructions this instruction directly or indirectly depends on. In the example shown in Figure 2, the sub-graph surrounded by a blue dashed line is a computation slice. Although the branch and computation slices are exclusive in
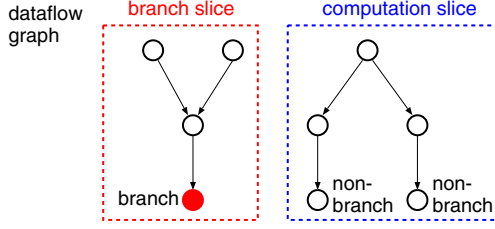
Fig. 2. Example of branch and computation slices.

Fig. 3. Structures for predicting unconfident branch slice instructions.

the example shown in Figure 2, they can overlap. This occurs, for example, if the result of an instruction in a branch slice flows to an instruction in a computation slice. Whether the two slices are exclusive or overlapped, the definitions and our scheme hold.

If the prediction of the branch cannot be sufficiently trusted, the associated branch slice is called an *unconfident branch slice*. If the issue of any instruction in the branch slice with a mispredicted branch is extra delayed, the misspeculation penalty can be increased, degrading the performance.

III. PRIORITIZING UNCONFIDENT BRANCH SLICES

In this section, we propose our scheme, called *prioritizing unconfident branch slices* (PUBS). This scheme is divided into two parts. One is a scheme that predicts whether an instruction belongs to an unconfident branch slice or not, which is described in Section III-A The other part is a scheme that prioritizes the issue of instructions in unconfident branch slices, which is described in Section III-B. Finally, in Section III-C, we discuss possible alternative implementations to PUBS and the adaptation of the PUBS scheme to an IQ organization that is different from that assumed in this section.

A. Predicting Unconfident Branch Slices

We predict whether an instruction belongs to an unconfident branch slice or not as follows:

1) The scheme records the confidence of the prediction of each branch using the past prediction correctness (Section III-A1).
2) The scheme constructs a pointer table, where each pointer links each instruction in a branch slice to the prediction confidence record of the associated branch (Section III-A2).
3) The scheme predicts whether a decoding instruction belongs to unconfident branch slices or not using the table of the pointers described above (Section III-A3).

The major structures used to implement this scheme are two tables, called the *confidence estimation table* (conf_tab) and *branch slice table* (brslice_tab). Figure 3 shows the relationship of the branch slice instructions and these structures. A minor structure, a table called the *define table* (def_tab), is prepared to keep track of instruction dependence relationships (not shown in the figure). We detail the scheme in the following subsections.
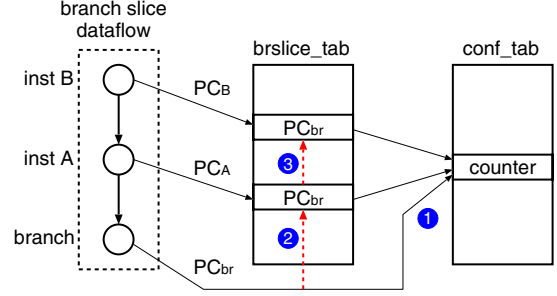
1) Estimation of the Confidence of Branch Prediction: We estimate branch prediction confidence using saturated resetting counters [3]. The conf_tab, where the index is the PC of a branch (PC_{br} in Figure 3), holds counters (mark (1) in Figure 3).

If a branch is executed, and an entry in the conf_tab is not allocated, an entry is allocated to the conf_tab. The counter in the allocated entry is then initialized to the maximum value in the case of correct prediction; otherwise, it is initialized to 0. If an entry is allocated, the counter in the entry is incremented by 1 if the prediction was correct (if the counter already has the maximum value, no action is taken); otherwise, it is reset to 0.

The confidence estimation is carried out as follows. If a branch is decoded, it looks up the conf_tab. If the counter value in the corresponding entry is the maximum value, the prediction of the associated branch is confident; otherwise, it is unconfident.

2) Linking Branch Slice Instructions to a Confidence Counter : We keep track of dataflow to find a branch slice. The def_tab is responsible for this task. The index of the def_tab is the logical destination register number of a decoding instruction, and each entry has the PC of the instruction. If an instruction is decoded, the PC of the instruction is written to the corresponding entry.

We link the instructions in a branch slice to a confidence counter in the conf_tab of the associated branch as follows:

1) If a branch is decoded, the scheme obtains the PCs of the instructions (inst A in Figure 3) that produce the source registers of the branch by looking up the def_tab using the logical source register numbers. Using the obtained PCs (PC_A in Figure 3), the scheme writes the PC of the *branch* (PC_{br}) to the corresponding entries of the brslice_tab (see mark (2) in Figure 3). Now, inst A has been indirectly linked to the confidence counter of the associated branch through the pointer in the brslice_tab.
2) Suppose that instruction A is decoded. The scheme obtains PC_{br} by looking up the brslice_tab using the PC of inst A. It then obtains the PCs of the producer instructions (inst B) by looking up the def_tab. Using the obtained PCs, the scheme writes PC_{br} to the corresponding entries of the brslice_tab (see mark (3) in
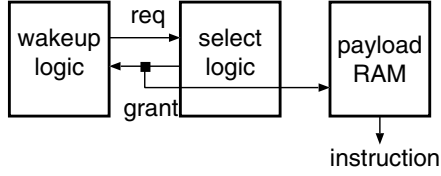
Fig. 4. Organization of the IQ.

Figure 3). Now, `inst B` has been indirectly linked to the confidence counter of the associated branch through the pointer in the brslice_tab.

3) The scheme repeats step 2.

3) Prediction of Unconfident Branch Slice Instructions: Predicting whether a decoding instruction belongs to an unconfident branch slice or not is carried out as follows:

1) If the decoding instruction is a branch, the scheme looks up the conf_tab using the PC. If a confidence counter is obtained and indicates low confidence, the branch belongs to an unconfident branch slice; otherwise (i.e., the confidence counter is not obtained or it indicates the maximum confidence), it does not.

2) If the decoding instruction is a non-branch instruction, the scheme looks up the brslice_tab using the PC. If a pointer is obtained, the scheme then accesses the conf_tab using the obtained pointer. If a confidence counter is obtained and indicates low confidence, the instruction belongs to an unconfident branch slice; otherwise, it does not.

If a pointer is not obtained from the brslice_tab, the instruction does not belong to a unconfident branch slice.

B. Prioritizing the Issue of the Instructions in Unconfident Branch Slices

This section describes how we prioritize the issue of the instructions in unconfident branch slices. Before describing the scheme, we explain the organization of conventional IQs as the base organization of PUBS in Section III-B1. We then explain the scheme in Sections III-B2. Finally, we describe an additional scheme to solve a problem caused by the main part of our scheme in Section III-B3.

1) Organization of IQs: **Overview**: The IQ largely comprises the wakeup logic, select logic, and payload RAM [4]–[6], as illustrated in Figure 4. The issue operation is pipelined: the wakeup and select in the first cycle, and the payload RAM read in the second cycle. The wakeup–select loop in the first cycle is the critical path of the IQ [4], and this loop is not pipeline in general; if pipelined, dependent instructions cannot be issued back-to-back.

There are two types of wakeup logic circuits: content addressable memory (CAM) or RAM [5], [7]. In the CAM type, the wakeup logic is a one-dimensional array, where each entry of the wakeup logic holds the tags of two source and destination operands, and ready flags indicating the data dependence state (resolved or not) for the corresponding

instruction. If both data dependences are resolved, an issue request is output.

In contrast, the RAM type has two matrices for each of two source operands [5]. Each row and column of the matrix is associated with an instruction in the IQ, and each element represents the data dependency between the instructions. In addition to the matrices, there is one row vector of ready flags for each matrix, where each bit corresponds to an instruction in the IQ. The ready flags are set depending on the values of the rows corresponding to the issued instructions. If the two ready bits corresponding to an instruction are set, an issue request is output.

The issue request is sent to the select logic, which grants some requests by considering resource constraints. As a circuit of the select logic, a tree arbiter circuit [8] and prefix-sum circuit [6], [9] are published. When using the tree arbiter circuit, the circuit must be stacked by the number of the issue width [8], [9]. This stacking considerably lengthens the delay of the select logic [8]. In contrast, the prefix-sum circuits does not need to be stacked for multiple issues (i.e., a single circuit is sufficient). Thus, it is much faster than the tree arbiter circuit [9].

The grant signals are sent to the payload RAM, which holds instructions. Instructions are read (issued) from the payload RAM and sent to the function units. The grant signals are also sent back to the wakeup logic. The destination tags corresponding to the grant signals are broadcast to the wakeup logic to update the ready flags in the CAM-type wakeup logic. In the RAM-type wakeup logic, the grant signal of a row reads the same row of the matrices.

Although there are several circuits for wakeup and select logic as described, these are orthogonal to our scheme. In other words, our scheme can be applied to any circuit.

Issue priority the select logic considers: The select logic is an arbiter that grants a maximum of IW requests from a maximum of IQS requests, where IW and IQS are the issue width and issue queue size, respectively. In the arbitration, the requests with higher priority are granted. Here, the priority is not flexible but is fixed with respect to the position of the IQ to make the select logic simple, where the closer to the head the instruction is, the higher the priority is [6], [8], [9]. Otherwise, the delay of the IQ is lengthened (see Section III-C1). Increasing the delay is not acceptable, because the critical path of the IQ is one of the critical paths of processors [8], and thus increasing the delay of the IQ can lengthen the clock cycle time.

Taxonomy in terms of instruction ordering: There is one more issue of note for this study. There are three types of IQs in terms of instruction ordering. The first type of IQ is the *shifting queue*. This type of IQ was used in old processors (e.g., DEC Alpha 21264 [10] two decades ago), where the size of the IQ is small. In the shifting queue, instructions stay physically ordered by age from the head to the tail of the queue. It is widely known that instruction age is highly correlated with instruction criticality in general, because a critical path is composed of a long dependence chain, and

instructions on a critical path thus stay in the IQ for a long time. Old instructions are therefore likely to be those on a critical path. Because the select logic is position based, the priority considered by the select logic is consistent with the criticality order of the instructions in the IQ. Thus, high IPC is achieved compared with the non-age-ordered queue. However, it needs a *compaction circuit* to fill the "holes" created by the instructions that have been issued, while keeping the order of instructions by age. This compaction circuit is very complex, and is inserted into the critical path of the IQ [10]. Thus, the delay of the IQ is significantly increased. The shifting queue is not used in current processors anymore.

The second type of IQ is the *circular queue*, which is composed of a circular buffer. In this queue, instructions stay physically ordered by age like the shifting queue, but it does not have a compaction circuit. Although this queue is simple, unlike the shifting queue, remaining "holes" cause serious capacity inefficiency. This significantly degrades the performance in capacity-sensitive programs. In addition, wrap-around in instruction order occurs, and this reverses the issue priority, further degrading the performance. The circular queue is also not used in current processors.

The last type of IQ is the *random queue*, where instructions are simply dispatched into the "holes". Because "holes" arise randomly over the long term, the order of instructions in the queue becomes random. The random queue is simple and thus the delay is short. However, the issue priority of the instructions is given randomly, because the instruction order is random. Therefore, the IPC is worse than that of the shifting queue.

To mitigate IPC degradation in the random queue, there is a circuit called the *age matrix* [7], [11]. The age matrix is used in parallel with the select logic [12], and selects the oldest ready instruction. Because instruction age is correlated with instruction criticality, as described previously, the age matrix is effective in terms of IPC. The downside is that the delay of the IQ is increased, because the global wires are lengthened by traversing the age matrix. This can increase the clock cycle time. Therefore, effectiveness is determined by balancing the IPC increase and delay increase. The details of the age matrix are described in Section V-G1.

Summary: We have described all the published organizations of IQ used in commercial processors to the best of our knowledge in this section. As described, the shifting or circular queues are not used anymore. Instead, although all processor vendors do not publish their IQ organization, the random queue alone or with an age matrix are used in modern processors [11]–[13]. In this paper, we assume a random queue without an age matrix as the base organization, and compare the performance of a processor with our scheme to this. Regarding the random queue with an age matrix, we compare the IPC, evaluate the delay, and discuss the results in Section V-G.

2) Prioritizing Unconfident Branch Slice Instructions: In this section, we describe how we assign the highest issue priority to the instructions in unconfident branch slices.
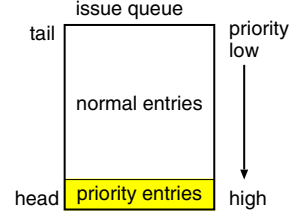

Fig. 5. Priority entries in the IQ.

As described in Section III-B1, instructions that are closer to the head of the IQ are assigned higher issue priority. Therefore, we reserve a small number of entries at the head of the IQ, which can be used by the only instructions in unconfident branch slices, as shown in Figure 5. We call these entries the *priority entries*, and call the remaining entries *normal entries*. The unconfident branch slice instructions in priority entries are issued with the highest priority, and thus misspeculation penalty is minimized.

When an instruction is dispatched (i.e., written) to the IQ, and if it is an instruction in an unconfident branch slice, it is dispatched into one of the priority entries. If there is no available priority entry, the dispatch is stalled (we evaluate whether it is better to stall or not stall dispatching to the normal entries in Section V-C). If the instruction does not belong to an unconfident branch slice, it is dispatched to a normal entry.

To implement this scheme, we divide the free list of the IQ into two lists: one each for the priority and normal entries. When an instruction is dispatched to a priority or normal entry, a free entry number is obtained from the corresponding free list.

3) Mode Switching: According to our evaluation in Section V-C, the optimal number of priority entries is only 6. Despite this very small number of entries, reserving entries can waste the capacity of the IQ, because they are not always full. This can degrade the performance for very capacity-sensitive programs. We have found that such programs are memory-intensive, where memory-level parallelism (MLP) is the most important source for high performance. To exploit MLP as much as possible, as many loads as possible must be issued in a short time. Thus, the capacity of the IQ is important. In addition, reducing branch misprediction penalty is less important in a situation where LLC misses occur frequently, because the LLC miss penalty is huge (hundreds of cycles).

To solve this problem, we introduce a simple mode switching scheme, where the PUBS scheme is enabled or not depending on memory-intensity. We observe last-level cache misses per kilo instructions (MPKI) periodically, and PUBS is enabled if the observed MPKI is less than a predetermined threshold; otherwise, PUBS is disabled. In the disabled periods, there are no priority entries and the IQ is used uniformly. At the instruction dispatch, two free lists (one for the priority entries and the other for the normal entries) are selectively used using a random number, where the selection probability is weighted

86

by the entry ratio. Because of the simplicity, there is no penalty for mode switching.

C. Discussion of the PUBS Implementation

In this section, we discuss a possible alternative implementation of the IQ for the PUBS scheme, and adaptation of PUBS to a distributed IQ instead of the unified IQ we have assumed thus far.

1) Select Logic with Flexible Priority: Given a scheme that marks instructions that are preferable to issue with high priority, a question that may be asked is whether a select logic that considers the marks into the priority can be implemented in a random queue. If it is possible, partitioning the IQ as in the PUBS scheme is unnecessary.

To the best of our knowledge, such a select logic has not been proposed in the literature, and we believe that it would be very difficult to implement without an extraordinary breakthrough. One possible but straightforward implementation is that many IQS-to-1 MUXes are placed between the wakeup logic and conventional position-based select logic. The MUXes are controlled by the marks, the issue requests from the wakeup logic are sorted by the marks by selecting one of IQS requests, and sorted requests are provided to the select logic. Although this circuit can be implemented theoretically, the huge fan-out of request signals and huge fan-in of MUXes significantly increase the delay, and thus it is unpractical.

2) Adapting to the Distributed IQ: In the explanation thus far, we have assumed a unified IQ, where the IQ is shared among function units. This type of IQ is currently used, for example, in the processors of Intel Sandy Bridge, Haswell, and Skylake [14]–[16], and is also used as the main IQ of IBM POWER7 and 8 [13], [17]. In contrast, AMD Zen uses an IQ distributed among integer function units [18], and each function unit thus has a dedicated IQ. The advantage of the unified IQ is capacity efficiency, while that of the distributed IQ is that the select logic is simplified. Although our study does not describe comprehensively which type of IQ is better, our PUBS scheme can be applied to a distributed IQ, where each IQ is partitioned into priority and normal entries.

IV. REDUCING COST

This section addresses the cost reduction of tables def_tab, brslice_tab, and conf_tab.

We organize brslice_tab and conf_tab as set-associative tables. In contrast, we prepare a full size table (i.e., the number of rows is equal to the number of logical registers) for def_tab, because the number of logical registers is small (i.e., 64). Although tagless organization is possible for the brslice_tab and conf_tab, a set-associative organization achieves a better performance according to our preliminary evaluation. In this organization, an entry of def_tab, brslice_tab, and conf_tab
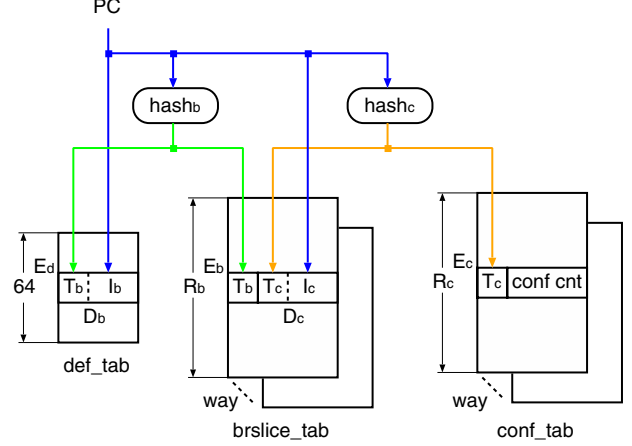

Fig. 6. Implementation of the three tables for PUBS with reduced hardware cost.

(E_d, E_b, and E_c, respectively) is composed of the following fields, respectively (see Figure 6):

$$E_d = D_b$$
$$E_b = T_b \text{ and } D_c$$
$$E_c = T_c \text{ and } confidence \ counter,$$

where D_b and D_c are PCs in the logical explanation of Section III-A, but are data generated from the PCs for implementation, which include an index to the brslice_tab and conf_tab, respectively. In contrast, T_b and T_c are tags for the brslice_tab and conf_tab, respectively.

Regarding D_b and D_c, they comprise the following concatenated information:

$$D_b = T_b \ || \ I_b$$
$$D_c = T_c \ || \ I_c,$$

where I_b and I_c are indices to the brslice_tab and conf_tab, respectively, and the symbol "$||$" represents concatenation.

The number of bits for I_X is determined by the number of rows (i.e., sets) of table X. In other words, it is $log_2 R_X$, where R_X is the number of rows of table X. In contrast, the number of bits for T_X is one for the portion of the PC remaining after eliminating the index portion from the PC in a straightforward implementation. For example, if we determine the number of rows of the brslice_tab to be 128, this then determines $I_b = 7$ and thus $T_b = 55(= 62 - 7)$. As easily found, the number of tag bits is significant, and this becomes a large cost overhead of our scheme.

To reduce this cost, we hash the tag. The hashed tag is generated by a bitwise XOR for each N-bit portion of the tag part of PC, as shown in Figure 7. According to our evaluations, the optimal N values for the brslice_tab and conf_tab that hardly degrade the performance are 8 and 4, respectively. This hashing significantly reduces the cost of the tables.
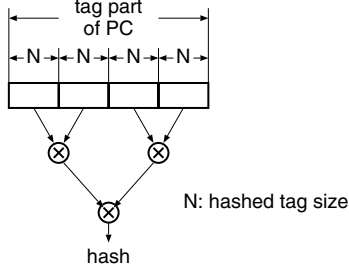
Fig. 7. Generation of hash from the tag part of the PC.

V. Evaluation Results

We first describe the methodology for evaluation in Section V-A. We then compare the performance of the PUBS scheme with the conventional random queue in Section V-B. In Sections V-C and V-D, we evaluate the performance sensitivity to the number of priority entries and the number of the confidence counter bits. We next evaluate the effectiveness of the mode switch in Section V-E. We then evaluate the hardware cost required for PUBS, and compare its performance to that of a processor with an enlarged branch predictor using this cost in Section V-F. We next compare the IPC to a processor with an IQ with an age matrix, and discuss the results in Section V-G. Finally, we evaluate the IPC sensitivity to the size of a processor in Section V-H.

A. Methodology

We built a simulator based on the SimpleScalar Tool Set version 3.0a [19] to evaluate IPC. The instruction set used was Alpha ISA. We used all the programs from the SPEC2006 benchmark suite except *wrf*; this program was 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. Our evaluation focuses on the programs with *difficult* branch prediction (D-BP), because our scheme reduces the misspeculation penalty caused by branch misprediction. The threshold in the difficulty of the branch prediction is 3.0 branch MPKI (mispredictions per kilo instructions). We also briefly show the evaluation results for *easy* branch prediction programs (E-BP) if necessary.

The configuration of the base processor used in the evaluation is summarized in Table I. The number of function units is important in this evaluation, because it is a main cause of issue conflicts. We used the number of function units in an ARM Cortex-A72 [20], which is a state-of-the-art high-performance and very energy-efficient processor used in smartphones and tablets. We also set the sizes of the IQ and reorder buffer to 64 and 128, respectively, because their sizes in the Cortex-A72 are 66 and 128 [20], [21]. We do not choose a PC processor but a mobile processor instead, because the mobile market is the larger than any other market, as described in Section I. The other important configuration is the branch predictor. Although the branch predictor is one of the most confidential components for processor vendors, and

TABLE I
BASE PROCESSOR CONFIGURATION.

Pipeline width	4-instruction wide for each of fetch, decode, issue, and commit
Reorder buffer	128 entries
IQ	64 entries
Load/store queue	64 entries
Physical registers	128(int) + 128(fp)
Branch prediction	34-bit history, 256-entry weight table perceptron, 2K-set 4-way BTB, 10-cycle state recovery penalty on misprediction
Function unit	2 iALU, 1 iMULT/DIV, 2 Ld/St, 2 FPU
L1 I-cache	32KB, 8-way, 64B line
L1 D-cache	32KB, 8-way, 64B line, 2 ports, 2-cycle hit latency, non-blocking
L2 cache	2MB, 16-way, 64B line, 12-cycle hit latency
Main memory	300-cycle min. latency, 8B/cycle bandwidth
Data prefetch	stream-based: 32-stream tracked, 16-line distance, 2-line degree, prefetch to L2 cache

TABLE II
PARAMETERS FOR PUBS.

Brslice_tab	128-set, 8-way, 8-bit hashed tag
Conf_tab	128-set, 8-way, 4-bit hashed tag
Confidence counter	6 bits
Priority entries in IQ	6
Normal entries in IQ	58
Dispatch policy	stall if no priority entry is available for unconfident branch slice instruction
Mode switch	10k-cycle interval, 3.0 LLC MPKI threshold

thus there is little publicly available information about them, AMD has revealed that its state-of-the-art processor, Zen, uses the perceptron branch predictor [18], [22]. Thus, we use it in this evaluation.

We simulated 100M instructions after the first 16B instructions were skipped using the *ref* inputs. The parameters that are specific to the PUBS scheme are summarized in Table II. The performance sensitivity to these parameters is evaluated in the following sections.

B. Performance

Figure 8 shows the speedup over the base. As described in Section V-A, the graph focuses on the results of the programs with D-BP. "GM diff" is the geometric mean of the results in D-BP. For the programs with easy branch prediction (E-BP), we show the only geometric mean ("GM easy").

As shown in the figure, PUBS achieves a 7.8% speedup on GM in D-BP, and no adverse effect is observed in E-BP. The speedups significantly vary depending on the program. The maximum speedup is 19.2% in *sjeng*, while the minimum speedup is 0.3% in *mcf*. This variation arises from 1) how difficult the branch prediction is and 2) which of the branch or computation slices are critical. Regarding 2), the frequency of LLC misses is the strongest factor. The higher this frequency is, the more computation slices become critical, because the LLC miss penalty is very long (300 cycles in our evaluation).
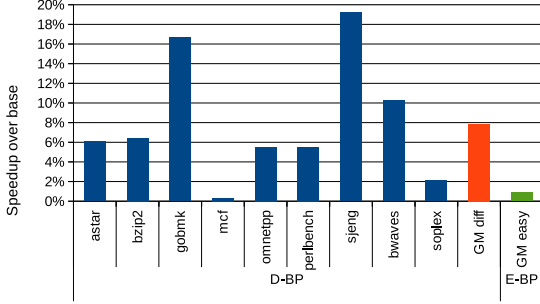
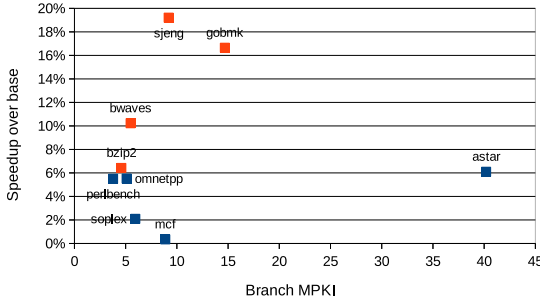Fig. 8.   Speedup of the PUBS over the base.

Fig. 9. Correlation between the speedup, branch MPKI, and memory intensity. Red and blue dots represent data of compute- and memory-intensive programs, respectively.
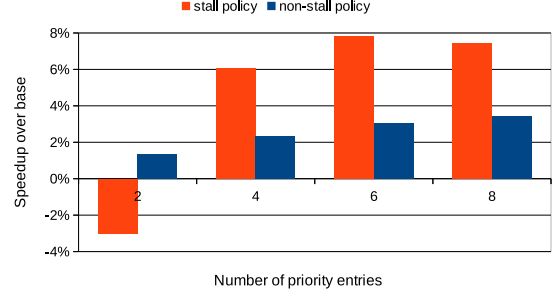

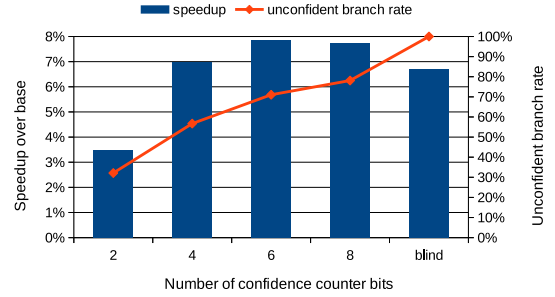Fig. 10.   Average speedup over the base in D-BP when varying the number of priority entries.

Fig. 11.   Average speedup over the base and the unconfident branch rate in D-BP when varying the number of confidence counter bits.

To confirm the reasons for the performance variation described above, we show the correlation between the speedup and branch MPKI in Figure 9.[1] The graph also shows the correlation with memory intensity using colored dots. The red and blue dots represent data of compute- and memory-intensive programs, respectively, where the threshold is 1.0 LLC MPKI.

If we focus on the data of compute-intensive programs (red dots), not surprisingly, the speedup is correlated with the branch MPKI. We also find that the speedup is larger for compute-intensive programs than for memory-intensive programs (blue dots).

C. Sensitivity to the Number of Priority Entries

Figure 10 shows the average speedup over the base in D-BP when varying the number of priority entries. Left and right bars represent speedups for the stall or non-stall policy on dispatch. In the stall policy, the instruction dispatch is stalled (default setting), if no priority entry is available for an unconfident branch slice instruction. In contrast, in the non-stall policy, it is not stalled, but the instruction is dispatched to a normal entry.

[1]The branch MPKI in *astar* is extraordinary large, but this is correct. We have confirmed these data using another simulator (gem5 [23]) and/or comparison with other branch predictors (e.g., gshare, bimode, and tournament predictors).

In the case of the stall policy, insufficient entries for unconfident branch slice instructions frequently stall the dispatch, thus degrading the performance. This situation is clearly seen in the case of two priority entries, where the performance is degraded from that of the baseline. In contrast, if the number of priority entries is excessive, the IQ capacity becomes insufficient for instructions that do not belong to unconfident branch slices. According to the evaluation results, the optimum number of priority entries is 6.

In the case of the non-stall policy, although the dispatch is not stalled even if the priority entry is not available, prioritizing the unconfident branch slice instructions is opportunistic, and is thus carried out only partially. As found in the evaluation results, the negative effect is stronger, and the stall policy is hence better.

D. Sensitivity to the Number of Confidence Counter Bits

Figure 11 shows the average speedup over the base (bar, left Y-axis) in D-BP, when varying the number of confidence counter bits in conf_tab from 2 to 8. The rightmost bar labeled "blind" is the speedup when the prediction of all branches are blindly estimated as unconfident. The figure also shows the ratio of the number of unconfident branches to the total number of dynamic branches (line, right Y-axis).

Basically, as the number of counter bits is increased, the unconfident branch rate is increased because the confidence counter is a resetting counter, and thus the branch confidence
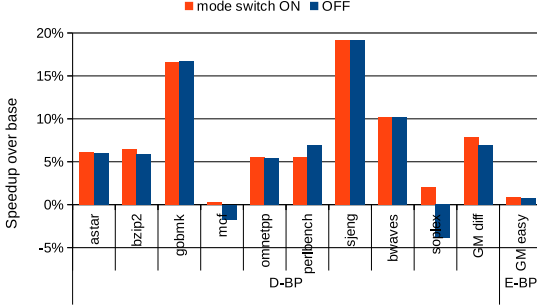
Fig. 12. Average speedup over the base when the mode switch is enabled or disabled.

Fig. 13. Comparison of performance with a processor with a large branch predictor with 8.4KB cost increase.

TABLE III
HARDWARE COST.

H/W	def_tab	brslice_tab	conf_tab	total
cost (KB)	0.1	2.5	1.4	4.0

prediction tends to be unconfident. This increases the coverage of unconfident branch slices, but the accuracy is decreased and it spuriously causes a shortage of priority entries. However, the results indicate that an aggressively estimation of unconfident is more beneficial. The number of optimal counter bits is 6, and the unconfident branch rate is 71% at this number of counter bits.

The "blind" model can eliminate the conf_tab and thus lower cost. However, the speedup is lower than that of the PUBS scheme with the conf_tab. Thus, conf_tab is worth introducing.

E. Effectiveness of the Mode Switch

Figure 12 shows the effectiveness of the mode switch. The left and right bars represent the speedup over that of the base when the mode switch is enabled and disabled, respectively. As described in Section III-B3, the mode switch allows the processor to fully use the capacity of the IQ when exploiting MLP is more beneficial than reducing branch misprediction penalty, while enabling PUBS when reducing branch misprediction penalty is more important.

As shown in the figure, although the performances of most programs are not substantially different when the mode switch is enabled or disabled, and thus the geometric means are also not substantially different, the performance is degraded in *mcf* and *soplex* when the mode switch is disabled. We have found that although the number of reserved priority entries are very small, there are programs that are sensitive to this small inefficiency in the IQ capacity.

F. Hardware Cost

The hardware required for PUBS is mainly three tables, i.e., def_tab, brslice_tab, and conf_tab. Table III shows the hardware cost in KB. As shown in the table, the required hardware cost is only 4.0KB.

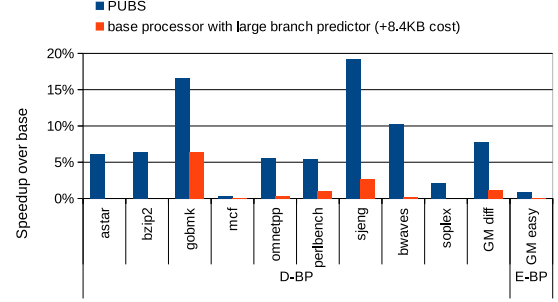Because PUBS aims to reduce branch misprediction penalty, we evaluate the performance of the base processor when we increase the cost of the branch predictor using the cost required for PUBS. Although it is 4.0KB, we increase the cost of the branch predictor by 8.4KB, which is more than double the cost of the default branch predictor. The history length and size of the weight table are 36 and 512, respectively.

Figure 13 shows the results. The left and right bars represent the speedups of PUBS with the default predictor and that of the base with the large branch predictor, respectively. As shown in the figure, the performance increase with the large branch predictor is marginal on average, and thus the performance is much less than that of PUBS. Therefore, PUBS is worth introducing for more reasons than just increasing the branch predictor.

G. Comparison to IQ with the Age Matrix

We have evaluated the performance of the PUBS scheme with a base processor using a random queue without the age matrix thus far. In this section, we compare the IPC in the case with an age matrix, evaluate the increase of the IQ delay when using the age matrix, and finally discuss the results.

1) Age Matrix: As described in Section III-B1, the IPC is degraded in the random queue because of random instruction ordering. To mitigate the IPC degradation, several processors [11]–[13] add a circuit called the age matrix to the IQ.

The age matrix receives the issue requests in parallel with the conventional select logic, as shown in Figure 14(b). It picks a single oldest instruction from ready instructions. This instruction is given the highest priority; the other instructions to be issued are selected using the conventional select logic [12]. Even though the age matrix selects only a single oldest instruction and other instructions are selected randomly, it is effective in terms of IPC.

Each row and column of the age matrix is associated with an instruction in the IQ [7], [11]. Each cell of the matrix holds a single bit representing age ordering information. In each row, the circuit determines whether the input issue request is the
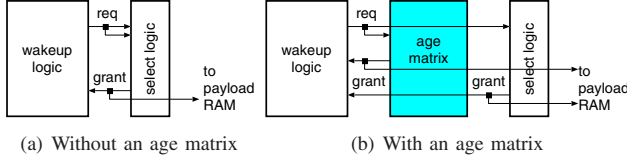
(a) Without an age matrix   (b) With an age matrix

Fig. 14. Organization of the IQ with and without an age matrix. The size of each circuit is scaled in proportion to the actual size.

(a) IPC

(b) Performance assuming the increase of the IQ delay directly increases the clock cycle time

Fig. 15. IPC and performance comparison when the age matrix is introduced.

oldest or not by bitwise ANDing of the row vector of the age matrix with the transposed issue request vector.[2]

Although the age matrix increases IPC, it lengthens the delay of the IQ, because the global wires (request and grant signals) are lengthened by traversing the age matrix, as shown in Figure 14(b). Although the resulting wire delay is significant, the wire delay is difficult to reduce in the modern fine LSI technology in general [24]. To find the increase of the delay, we designed the IQ (the wakeup logic is the CAM type, and the select logic is the prefix-sum circuit) at the transistor level, assuming MOSIS design rules [25]. According to our LSI layout of the IQ, the width of the age matrix is very wide; it is nearly the same width as that of the wakeup logic or as long as 65% of the height of the IQ. See Figures 14(a) and (b) to compare the size of the IQ without the age matrix to that with the age matrix. Note that the size of each circuit is scaled in proportion to the actual size in these figures. We carried out the circuit simulation using HSPICE to evaluate the delay of the IQ, assuming the 16nm predictive transistor model [26] developed by the Nanoscale Integration and Modeling Group of Arizona State University for HSPICE, and the resistance and capacitance per unit length of the wire predicted by the International Technology Roadmap for Semiconductors [27]. Drivers and repeaters were optimally inserted on long wires to reduce the delay, according to experimentation. As a result, we have found that the age matrix increases the delay of the IQ by 13%.
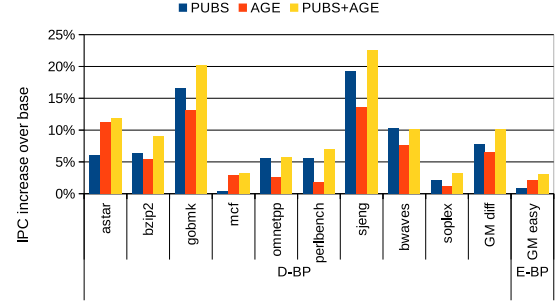
Note that, although we assumed the layout shown in Figure 14(b), we could use another layout where the age matrix and select logic are interchanged. However, the layout in Figure 14(b) was a better choice according to our evaluation.

2) Comparison Results: Figure 15(a) shows the IPC increase over the base for the following three models:
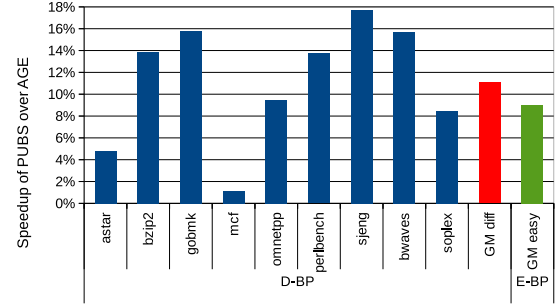
- PUBS: PUBS is introduced.
- AGE: The IQ (random queue) has an age matrix.
- PUBS+AGE: PUBS is introduced and the IQ has an age matrix.

As shown in the figure, the age matrix is effective in terms of IPC. The effectiveness is larger in D-BP than in E-BP (compare "GM diff" with "GM easy"), even though the age matrix does not consider branch misprediction. This infers that branch slices often include an oldest ready instruction. However, the IPC of the AGE is less than that of PUBS in

D-BP (see "GM diff"), although it is slightly more than that of PUBS in E-DP (see "GM easy").

Combining PUBS with AGE further increases the IPC over PUBS. This is because each considers the issue priority from different viewpoints. In other words, PUBS focuses on the criticality related to branch misprediction, while the age matrix considers general criticality. As a result, the gap between AGE and PUBS+AGE becomes significant, where the IPC increase of AGE over the base is 6.5%, while that of PUBS+AGE is 10.2% on average in D-BP.

Note that above results are those with respect to IPC, but not performance. To compare the performance, we must consider the clock cycle time. Because the delay of the IQ is one of the critical paths in a processor, the delay increase can directly lengthen the clock cycle time. As described in Section V-G1, the age matrix increases the delay of the IQ by 13% according to our LSI design. Figure 15(b) shows the performance of PUBS over that of AGE, assuming that the increase of the IQ delay in AGE directly increases the clock cycle time. As shown in the figure, the performances of PUBS over AGE is 11.1% on average in D-BP.

As described in Section V-G1, the benefit of introducing the age matrix is determined by balancing the IPC increase and delay increase. According to our evaluations, the delay increase is larger than the IPC increase, and thus introducing the age matrix is not beneficial if we assume that the delay increase directly lengthens the clock cycle time. Nonetheless,

[2]We have explained the operation based on [11]. A different but essentially the same age matrix circuit is presented in [7].

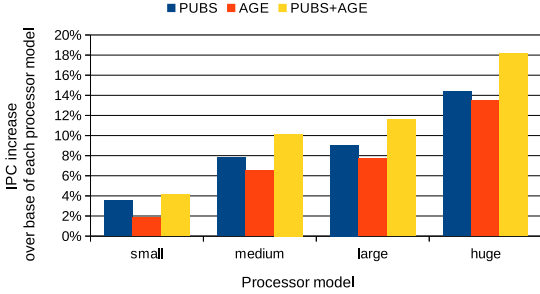| Parameter	Processor model			
	Small	Medium	Large	Huge
Fetch/decode/Issue/ commit width	3	4	6	8
IQ size	32	64	128	256
Load/store queue size	32	64	128	256
Reorder buffer size	64	128	256	512
Physical regs (int+fp)	64+64	128+128	256+256	512+512
Number of iALUs	2	2	3	4
Number of FPUs	1	2	3	3

Fig. 16. IPC increase when varying the size of the processor.

several processor vendors introduce the age matrix. This fact does not immediately imply that our delay evaluation is incorrect, because LSI parameters, including the layout rules, wire capacitance and resistance, and the characteristics of transistors in commercial processors, can be different from the assumptions in our LSI design. Nonetheless, it is a firm fact that the width of the age matrix is wide. Recall that the age matrix is a two-dimensional matrix with the number of rows and columns equal to the issue queue size. Thus, the global wires for the request and grant signals are significantly lengthened by traversing the age matrix, and consequently, the IQ delay is increased. Therefore, the gap in performance between PUBS and AGE is increased more than the gap of the IPC shown in Figure 15(a).

H. Sensitivity to the Size of Processor

In this section, we evaluate the IPC of PUBS, AGE, and PUBS+AGE when varying the size of a processor. In this evaluation, we do not consider increase of the clock cycle time caused by the age matrix. We evaluated four processor models of different sizes, which are shown in Table IV. The medium-sized processor is our default processor. The seven parameters are scaled; the other parameters remain unchanged from the default values. The larger the window size (i.e., the size of IQ, load/store queue, reorder buffer, and register files) is, the more the issue conflicts occur, thereby increasing the effectiveness of PUBS and AGE. In contrast, the more the issue width and the number of function units are increased, the less the issue conflicts occur, thus decreasing the effectiveness of PUBS and AGE.

Figure 16 shows the average IPC increase over the base of each processor model in D-BP. As shown in the figure, the criticality-aware selection schemes (i.e., PUBS and AGE) become more effective as the processor size is increased. Comparing PUBS with AGE, PUBS achieves a higher IPC in any processor model. In addition, PUBS+AGE is more effective than PUBS or AGE alone. These results indicate that the effectiveness of the PUBS scheme is stable if the processor resources are balanced independently of their sizes.

VI. RELATED WORK

The issue queue was extensively studied around 2000. A comprehensive survey was carried out by Abella et al [28].

Butler et al. investigated the effect of several select policies of the IQ, including random selection and selection assigning higher priority to instructions on *branch paths* (which we call branch slices in this paper) [29]. According to their evaluation results, the select policies they investigated deliver almost the same performance for integer programs (SPEC89 benchmark) but have performance differing by up to 20% for floating-point programs. They pointed out that the similar performance arises from the number of ready instructions in a clock cycle being heavily skewed to zero. In contrast, our results are different from their results. These differences arise because Butler et al. assume full function units capable of integer execution, which avoids issue conflicts in the function units. We also confirmed that the number of ready instructions is skewed to zero, but there are still a significant number of clock cycles where the number of ready instructions is more than two. These statistics are highly dependent on the program.

A processor that implemented a strict age-based policy (i.e., shifting queue) is DEC Alpha 21264 [10]. Because the complex compaction operation is inserted into the critical path of the IQ, the shifting queue is practical only in a small IQ (20 entries in Alpha 21264). In current processors with large IQs, it is not used anymore.

Although age is correlated with instruction criticality, it is only a heuristic. Ideally, instructions on a critical path of the dataflow should be selected with high priority for high performance. Fields et al. proposed a scheme that predicts the criticality of an instruction, including the consideration of branch misprediction [30]. However, the scheme is difficult to implement because of its high complexity and large area.

In several modern processors, a random queue with age matrix is used [11]–[13]. Because the position-based select logic cannot implement the age-aware policy in the random queue, the age matrix that selects the oldest ready instruction helps the age-aware selection. An age matrix circuit was proposed in [11], while a similar circuit was presented in [7]. Although the age matrix increases IPC, the delay of the IQ is also increased. To take advantage of the IPC increase, an advanced LSI technology that reduces wire delay is required. However, despite the efforts of LSI fabrication companies, the wire delay increases significantly as the generation of LSI technology advances in general [24], [27], which is a long and firm trend in LSI technology.

The delay of the conventional age matrix can increase the IQ delay. Because the wire delay is dominant in the delay of this circuit, reducing the number of cells that the wires traverse is effective in reducing the delay. This is also helpful to reduce the delay of global wires traversing this circuit. Sassone et al. proposed a scheme that dynamically allocates transposed issue request lines for a group of instructions to reduce the width of the age matrix [7]. The downside of this scheme is that it still requires an arbiter that arbitrates the requests among the instructions in a group. Because the instructions are distributed in the IQ even for a single group (because the queue is a random queue), the wires for the arbiter traverse the IQ vertically. The arbiter delay is thus not trivial, and the effectiveness is consequently reduced.

Speculative precomputation extracts a program slice that includes the instructions that are necessary to compute the outcome of difficult branches, and forking the slice from the original thread as a helper thread in a different context [31], [32]. The precomputation thread forwards the branch outcome to the original thread, avoiding the branch misprediction. While this method is effective, it has a large overhead because of the need for helper threads. Specifically, they consume a part of the processor core resources and this causes conflicts with the original thread in a simultaneous multithreading (SMT) implementation, or consumes the entire core resources when using the cores for helper threads.

VII. Conclusions

Single-thread performance has been hardly improved in this decade. This is a very serious problem, especially in the smartphone era. Now that Dennard scaling has ended, restudying core architecture has become important again. One of the major hurdles for single-thread performance improvement is branch misprediction. Branch predictors that reduce the frequency of misprediction have been studied extensively for several decades. However, reducing the misprediction penalty has been rarely studied. This paper proposes a scheme called PUBS that reduces the misspeculation penalty. More specifically, the waiting cycles of a branch in the IQ, which is included in the misprediction penalty, are reduced by issuing instructions the branch directly or indirectly depends on with the highest priority.

Our evaluation results using SPEC2006 benchmark programs show that PUBS improves the performance by 7.8% on average of the programs with difficult branch prediction (D-BP), with only 4.0KB cost. We also evaluate the performance with respect to a processor with an IQ using an age matrix, and found that the performance of PUBS significantly outweighs that of the processor with the age matrix.

Acknowledgments

References

[1] https://www.gartner.com/newsroom/id/3816763.

[2] "5th JILP workshop on computer architecture competitions," https://www.jilp.org/cbp2016/, May 2016.

[3] E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning confidence to conditional branch predictions," in *Proceedings of the 27th International Symposium on Microarchitecture*, December 1996.

[4] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Quantifying the complexity of superscalar processors," University of Wisconsin-Madison, Tech. Rep. CS-TR-1996-1328, November 1996.

[5] M. Goshima, K. Nishino, T. Kitamura, Y. Nakashima, S. Tomita, and S. Mori, "A high-speed dynamic instruction scheduling scheme for superscalar processors," in *Proceedings of the 34th Annual International Symposium on Microarchitecture*, December 2001.

[6] 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*, October 2011.

[7] P. G. Sassone, J. Rupley II, E. Brekelbaum, G. H. Loh, and B. Black, "Matrix scheduler reloaded," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[8] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[9] M. Goshima, "Research on high-speed instruction scheduling logic for out-of-order ILP processor," Ph.D. dissertation, Kyoto University, 2004.

[10] J. A. Farrell and T. C. Fischer, "Issue logic for a 600-MHz out-of-order execution microprocessor," *Journal of Solid-State Circuits*, vol. 33, issue 5, May 1998.

[11] R. P. Preston, R. W. Badeau, D. W. Bailey, S. L. Bell, L. L. Biro, W. J. Bowhill, D. E. Dever, S. Felix, R. Gammack, V. Germini, M. K. Gowan, P. Gronowski, D. B. Jackson, S. Mehta, S. V. Morton, J. D. Pickholtz, M. H. Reilly, and M. J. Smith, "Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading," in *2002 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, February 2002.

[12] M. Golden, S. Arekapudi, and J. Vinh, "40-entry unified out-of-order scheduler and integer execution unit for the AMD Bulldozer x86-64 core," in *2011 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, February 2011.

[13] B. Sinharoy, J. A. V. Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbacha, T. Karkhanis, and K. M. Fernsler, "IBM POWER8 processor core microarchitecture," *IBM Journal of Research and Development*, vol. 59, issue 1, January - February 2015.

[14] L. Gwennap, "Sandy Bridge spans generations," *Microprocessor Report*, September 2010.

[15] K. Krewell, "Intel's Haswell cuts core power," *Microprocessor Report*, September 2012.

[16] L. Gwennap, "Skylake speed shifts to next gear," *Microprocessor Report*, September 2015.

[17] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. V. Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams, "IBM POWER7 multicore server processor," *IBM Journal of Research and Development*, vol. 55, issue 3, May - June 2011.

[18] D. Kanter, "AMD finds Zen in microarchitecture," *Microprocessor Report*, August 2016.

[19] http://www.simplescalar.com/.

[20] L. Gwennap, "ARM optimizes Cortex-A72 for phones," *Microprocessor Report*, May 2015.

[21] M. Humrick, "ARM Cortex-A72 architecture deep dive," *Tom's Hardware*, January 2016.

[22] D. A. Jiménez, "Dynamic branch prediction with perceptrons," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, January 2001.

[23] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, May 2011.

[24] C. Hou, "A smart design paradigm for smart chips," in *2017 IEEE International Solid-State Circuits Conference, Digest of Technical Papers, Plenary Session*, February 2017.

[25] http://www.mosis.com/.

[26] http://ptm.asu.edu/.

[27] International Technology Roadmap for Semiconductors, (http://www.itrs2.net/).

[28] J. Abella, R. Canal, and A. González, "Power- and complexity-aware issue queue designs," *IEEE Micro*, September-October 2003.

[29] M. Butler and Y. Patt, "An investigation of the performance of various dynamic scheduling techniques," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992.

[30] B. Fields, S. Rubin, and R. Bodík, "Focusing processor policies via critical-path prediction," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, June 2001.

[31] C. B. Zilles and G. S. Sohi, "Execution-based prediction using speculative slices," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[32] A. Roth and G. S. Sohi, "Speculative data-driven multithreading," in *Proceeding of the 7th Annual International Symposium on High Performance Computer Architecture*, January 2001.