Fetch Gating Control through Speculative Instruction Window Weighting

Hans Vandierendonck[1] and André Seznec[2]

[1] Ghent University, Department of Electronics and Information Systems/HiPEAC, B-9000 Gent, Belgium, hvdieren@elis.ugent.be
[2] IRISA/INRIA/HiPEAC Campus de Beaulieu, 35042 Rennes Cedex, France, seznec@irisa.fr

Abstract. In a dynamic reordering superscalar processor, the front-end fetches instructions and places them in the issue queue. Instructions are then issued by the back-end execution core. Till recently, the front-end was designed to maximize performance without considering energy consumption. The front-end fetches instructions as fast as it can until it is stalled by a filled issue queue or some other blocking structure. This approach wastes energy: (i) speculative execution causes many wrong-path instructions to be fetched and executed, and (ii) back-end execution rate is usually less than its peak rate, but front-end structures are dimensioned to sustained peak performance. Dynamically reducing the front-end instruction rate and the active size of front-end structure (e.g. issue queue) is a required performance-energy trade-off. Techniques proposed in the literature attack only one of these effects.
In this paper, we propose Speculative Instruction Window Weighting (SIWW), a fetch gating technique that allows to address both fetch gating and instruction issue queue dynamic sizing. A global weight is computed on the set of inflight instructions. This weight depends on the number and types of inflight instructions (non-branches, high confidence or low confidence branches, ...). The front-end instruction rate can be continuously adapted based on this weight. SIWW is shown to perform better than previously proposed fetch gating techniques. SIWW is also shown to allow to dynamically adapt the size of the active instruction queue.

1 Introduction

Dynamic reordering superscalar architectures are organized around an instruction queue that bridges the front-end instruction delivery part to the back-end execution core. Typical performance-driven designs maximize the throughput of both the front-end and the back-end independently. However, it has been noted [2,3,12] that such designs waste energy as the front-end fetches instructions as fast as it can up to the point where the back-end fills up and the front-end necessarily stalls. All of this fast, aggressive work may be performed at a lower and more power-efficient pace, or it may even turn out to be unnecessary due to control-flow misspeculation.

Historically, the first step to lower the front-end instruction rate relates to fetching wrong-path instructions. By assigning confidence to branch predictions, Manne *et al* [14] gate instruction fetch when it becomes likely that fetch is proceeding along the wrong execution path. However, with the advent of highly accurate conditional [7,10,16] and indirect branch predictors [5,17], the impact of wrong-path instructions on energy decreases [15].

Besides fetching wrong-path instructions, it has been shown that the front-end flow rate may well exceed the required back-end rate [2,3]. Closely linked to this flow-rate mismatch is a mismatch between the required issue queue size and the available issue queue size. Consequently, fetch gating mechanisms are combined with dynamic issue queue adaptation techniques to increase energy savings [3].

This paper contributes to this body of work by presenting a new fetch gating algorithm built on these principles. Our fetch gating algorithm simultaneously tracks branch confidence estimation and the set of already inflight and unissued instructions. In effect, it modulates branch confidence estimation by issue queue utilization: as issue queue utilization is higher, uncertainty in the control-flow speculation weighs stronger to limit the front-end flow rate. Hereby, our technique avoids both wrong-path work and it matches the front-end flow rate to the back-end flow rate.

To illustrate the advantage of our technique, let us consider the two following situations. In example (A), 50 instructions, a low-confidence and 6 high-confidence branches have already been fetched. In example (B), 5 instructions and a single low-confidence branch have been fetched. If the next instruction is a low-confidence branch then a fetch gating control mechanism based only on branch confidence estimation and boosting [14] will take exactly the same decision for the two situations. A simple analysis shows that for (A), delaying the next instruction fetch for a few cycles (but maybe not until the low confidence branch resolves) is unlikely to degrade performance while for (B), delaying it is very likely to induce a few cycles loss if the two low-confidence branchs are correctly predicted.

The first contribution of this paper is Speculative Instruction Instruction Window Weighting (SIWW). Instead of only considering confidence on inflight branches for controlling fetch gating, we consider the overall set of already inflight and unissued instructions, i.e. the speculative instruction window. SIWW tries to evaluate whether or not the immediate fetch of the next instruction group will bring some extra performance. When the expected benefit is low, then fetch is gated until the benefit has increased or a branch misprediction has been detected. This expected performance benefit increases when (i) branch instructions are resolved or (ii) instructions execute and the number of unissued instructions in the issue queue drops. Our experiments show that fetch gating based on SIWW easily outperforms fetch gating schemes based on confidence boosting [14], fetch throttling [1] as well as issue queue dynamic sizing techniques [3].

A second contribution of this paper is to show that fetch gating control through SIWW can be efficiently implemented without any extra storage ta-

ble for confidence estimation. Current state-of-the-art branch predictors such as O-GEHL [16] and piecewise linear branch prediction [10] provide a confidence estimate for free. We show that this estimate does not work very well for fetch gating control through boosting, but it works well with SIWW fetch gating and instruction queue dynamic sizing.

The remainder of the paper is organized as follows. Section 2 quickly reviews related work on fetch gating or throttling and dynamic sizing of instruction queues. Section 3 describes our proposal to use SIWW for fetch gating. Our experimental framework is presented in Section 4. Section 5 presents the performance of SIWW and compares it to confidence boosting, the previous state-of-the-art approach. Finally, Section 6 presents possible future research directions and summarizes this study.

2 Related Work

Gating the instruction fetch stage on the first encountered low-confidence branch results in significant performance loss. By delaying gating until multiple low-confidence branches are outstanding – a technique called *boosting* – it is possible to limit this performance loss while still removing extra work [14].

Fetch throttling slows down instruction fetch by activating the fetch stage only once every N cycles when a low-confidence branch is in-flight [1]. This reduces the performance penalty of pipeline gating, but sacrifices energy reduction by allowing additional extra work.

Decode/commit-rate fetch gating is an instruction flow-based mechanism that limits instruction decode bandwidth to the actual commit bandwidth [2]. This technique saves energy even for correct-path instructions, as only the required fetch bandwidth is utilized.

Buyuktosunoglu *et al* [3] combine fetch gating with dynamic issue queue adaptation in order to match front-end and back-end instruction flow rates and to match the issue queue size to its required size. They propose to gate instruction fetch based on the observed parallelism in the instruction stream. Fetch is gated during one cycle when instruction issue occurs mostly from the oldest halve of the reorder buffer and the issue queue is more than half full.

Several techniques to dynamically adapt the issue queue size have been proposed in the literature. Folegnani and Gonzalez [6] divide the reorder buffer into portions of 8 instructions. The reorder buffer grows and shrinks by units of a portion. The reorder buffer is dimensioned by monitoring the number of instructions that issue from the portion of the reorder buffer holding the youngest instructions.

Just-in-time (JIT) instruction delivery [12] applies a dynamic reconfiguration algorithm to adapt the reorder buffer size. It determines the smallest reorder buffer size that yields a performance degradation less than a preset threshold.

In [4], the issue queue is also divided into portions of 8 instructions but the queue size is determined by its average utilization over a time quantum.

3 Speculative Instruction Window Weighting

Instead of only considering confidence on inflight branches for controlling fetch gating, we consider the overall set of already inflight instructions, i.e. the speculative instruction window. Speculative Instruction Window Weighting (SIWW) tries to evaluate whether or not the **immediate** fetch of the next instruction group will bring some performance benefit. Our thesis is that this benefit decreases with the number of already inflight instructions, with the number of branches and with the quality of the branch prediction (i.e. with the confidence in the predictions). The performance benefit may also depend on the precise type of already inflight instructions and parameters such as latency (e.g. divisions, multiplications, loads that are likely to miss), etc.

For this purpose, a global Speculative Instruction Window (SIW) weight is computed on the overall set of unexecuted inflight instructions. The SIW weight is intended to "evaluate" the performance benefit that immediately fetching new instructions would deliver.

The SIW weight is constantly changing. It increases when instructions are fetched and it decreases as instructions are executed. When the SIW weight exceeds a pre-set threshold, instruction fetch is halted. As soon as the SIW weight drops below the threshold, the instruction fetch is resumed (Figure 1).

3.1 Computing the SIW Weight: Principle

The SIW weight is computed from the overall content of the instruction window. To obtain a very accurate indicator, one should take into account many factors, such as dependencies in the instruction window, instruction latency, etc. However, realistic hardware implementation must also be considered. Therefore, we propose to compute the SIW weight as the sum of individual contributions by the inflight instructions. These contributions are determined at decode time.

As an initial implementation of SIWW, we assign a SIW weight contribution to each instruction by means of its instruction class. The instruction classes and SIW weight contributions used in this paper are listed in Table 1.

The weight contributions reflect the probability of a misprediction. Thus, low-confidence branches are assigned significantly higher weight contributions than high-confidence branches. High-confidence branches are assigned higher weight contributions than non-branch instructions because high-confidence branches too are mispredicted from time to time. Return instructions have a small weight contribution because they are predicted very accurately.

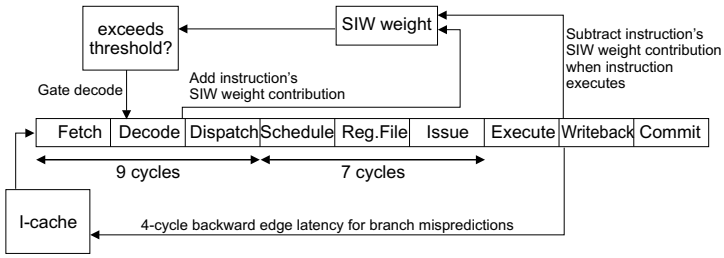

Fig. 1. Block diagram of a pipeline with speculative instruction window weighting.

Table 1. SIW weight contributions.

Instruction type	Contrib.
high-confidence conditional branches	8
low-confidence conditional branches	40
returns	8
high-confidence indirect branches	8
low-confidence indirect branches	40
unconditional direct branches	1
non-branch instructions	1

Unconditional direct branches have the same weight contributions as non-branch instructions as their mispredict penalty is very low in the simulated architecture. Mispredicted targets for unconditional direct branches are captured in the decode stage. Fetch is immediately restarted at the correct branch target.

The weight contributions depend on the accuracy of the conditional branch predictor, branch target predictor and return address stack and may have to be tuned to these predictors. The weight contributions describe only the speculativeness of the in-flight instructions and are therefore independent of other micro-architectural properties.

3.2 A Practical Implementation of SIW Weight Computation

In principle, the SIW weight is computed from the old SIW weight by adding the contributions of all newly decoded instructions and substracting the contributions of all executed instructions. Initially, the SIW weight is zero. However, when a branch misprediction is detected, the SIW weight represents an instruction window where some instructions that have been fetched before the mispredicted branch are still to be executed. Restoring the SIW weight to its correct value while resuming instruction fetch after the mispredicted branch would require to retrieve the contributions of these instructions and perform an adder tree sum.

To sidestep a complex adder tree, we approximate the SIW weight by setting it to zero on a misprediction. The SIW weight then ignores the presence of unexecuted instructions in the pipeline. However, the SIW weight contribution of these instructions may not be substracted again when they execute. To protect against substracting a contribution twice, we keep track of the most recently recovered branch instruction. Instructions that are older (in program order) should not have their SIW weight contribution substracted when they execute.

Experimental results have shown that this practical implementation performs almost identical to the exact scheme. In most cases, when a mispredicted branch is detected, the instruction window will be largely drained, causing the exact SIW weight to drop to the range 20–50 (compare this to the SIW threshold of 160). Most of these remaining instructions are executed before the first corrected path instructions reach the execution stage. At this time, the approximate SIW weight is already very close to its maximum value, minimizing the impact of the temporary underestimation.

3.3 Dynamically Adapting the SIW Weight Contributions

The weight contributions proposed in Table 1 are based on the prediction accuracy of particular types of branches (low-confidence vs. high-confidence, conditional vs. indirect, etc.). However, the prediction accuracy varies strongly from benchmark to benchmark, so the weight contributions should reflect these differences. To improve the SIWW mechanism, we investigated ways to dynamically adapt the weight contributions based on the prediction accuracy.

We dynamically adjust the weight contribution of each instruction class in Table 1 where the baseline contribution differs from 1. Each contribution is trained using only the instructions in its class. The contribution is increased when the misprediction rate is high (high probability of being on the wrong execution path) and is decreased when the misprediction rate in its instruction class is low. To accomplish this, we use two registers: a p-bit register storing the weight contribution and a $(p+n)$-bit register storing a counter. Practical values for p and n are discussed below.

The counter tracks whether the weight contribution is proportional to the misprediction rate. For each committed instruction in its class, the counter is incremented with the weight. If the instruction was mispredicted, it is also decremented by 2^p. Thus, the counter has the value $c - f2^p$ where c is the current weight contribution and f is the misprediction rate. As long as the counter is close to zero, then the contribution is proportional to the misprediction rate. When the counter deviates strongly from zero, then the weight contribution needs adjustment. When the counter overflows, the weight contribution is decremented by 1 because it was higher than the misprediction rate. When the counter underflows, the weight contribution is incremented by 1. At this point, the counter is reset to zero to avoid constantly changing the weight contribution.

When computing the overall SIW weight, the weight contributions for branch instructions are no longer constants but are read from the appropriate register.

The values for p and n used in this paper are 7 and 8, respectively. Note that the size of the counter (n) determines the learning period. In total, we need 5 7-bit registers, 5 15-bit registers and a small number of adders and control to update these registers. This update is not time-critical because these registers track the average over a large instruction sequence and change slowly over time.

3.4 Selecting the SIW Threshold

The SIW threshold remains fixed. Selecting the SIW threshold involves a trade-off between reducing wrong-path instructions (smaller thresholds) and execution speed (larger thresholds). The SIW threshold also depends on the weight contributions: larger weight contributions lead to a larger SIW weight, so to gate fetch under the same conditions a larger SIW threshold is required too. Finally, the SIW threshold depends on branch prediction accuracy too. We analyze SIWW using multiple SIW thresholds in order to quantify this trade-off.

4 Experimental Environment

Simulation results presented in this paper are obtained using sim-flex[3] with the Alpha ISA. The simulator is modified and configured to model a future deeply pipelined processor (Table 2). The configuration is inspired by the Intel Pentium 4 [8], but at the same time care is taken to limit the extra work that the baseline model performs for wrong-path instructions. Amongst others, we use conservative fetch, decode and issue widths of 4 instructions per cycle because this is a good trade-off between power consumption and performance and it is a more realistic number if power efficiency is a major design consideration.

Gating control resides in the decode stage because the instruction type, confidence estimates and SIW contributions are known only at decode. To improve the effectiveness of the gating techniques, the fetch stage and the decode stage are simultaneously gated.

Two different branch predictors are considered in this study: gshare and O-GEHL. These predictors feature 64 Kbits of storage. For gshare, we considered 15 bits global history, a JRS confidence estimator [9] with 4K 4 bit counters and 15 as the confidence threshold. Power consumption in the JRS confidence estimator is modeled by estimating power dissipation in the JRS table. For O-GEHL, we simulated the baseline configuration presented in [16]. As in [11], we use the update threshold as the confidence estimator for O-GEHL. If the absolute value of the weight sum is above the update threshold then we classify the branch as high confidence, and low-confidence otherwise. We call this self confidence estimation as in [11]. Self confidence estimation consumes no additional power.

A cascaded branch target predictor [5] is implemented. Confidence is estimated as follows. Each entry is extended with a 2-bit resetting counter. The counter is incremented on a correct prediction and set to zero on an incorrect prediction. An indirect branch is assigned high confidence when the counter is saturated in the highest state.

We simulate SPEC CPU 2000 benchmarks executing the reference inputs.[4] Traces of 500 million instructions are obtained using SimPoint[5].

5 Evaluation

Table 3 displays the characteristics of our benchmark set considering gshare and OGEHL as branch predictors. Column CND and IND represents the misprediction rate in mispredicts per 1000 instructions for conditional branches and indirect branches. Notwithstanding high prediction accuracy, the extra fetch work (EFW) represents between 15.5% and 93.6% extra work on the SPECint benchmarks when using the O-GEHL predictor. The SPECfp benchmarks exhibit less than 10% extra fetch work. Using gshare instead of O-GEHL as branch

[3] `http://www.ece.cmu.edu/~simflex`

[4] Our simulation infrastructure cannot handle the perlbmk inputs, so we resort to the SPEC'95 reference scrabble input.

[5] `http://www.cs.ucsd.edu/~calder/SimPoint/`.

Table 2. Baseline Processor Model

Processor core	
Issue width	4 instructions
ROB, issue queue	96
Load-store queue	48
Dispatch-execute delay	7 cycles
Fetch Unit	
Fetch width	4 instructions, 2 branches/cycle
Instruction fetch queue	8 instructions
Fetch-dispatch delay	9 cycles
Cond. branch predictor	gshare or O-GEHL
Return address stack	16 entries, checkpoint 2
Branch target buffer	256 sets, 4 ways
Cascaded branch target predictor	64 sets, 4 ways, 8-branch path history
Memory Hierarchy	
L1 I/D caches	64 KB, 4-way, 64B blocks
L2 unified cache	256 KB, 8-way, 64B blocks
L3 unified cache	4 MB, 8-way, 64B blocks
Cache latencies	1 (L1), 6 (L2), 20 (L3)
Memory latency	150 cycles

predictor reduces the overall base performance by 5.65%. It also induces more work on the wrong path: the average extra instruction fetch work is increased from 39.2% to 52.4%.

Table 3. Characteristics of the benchmarks

	IPC	CND	IND	EFW	EPI	ORA
O-GEHL	1.77	2.76	0.33	39.2%	18.6	17.5
gshare	1.67	4.23	0.38	52.4%	18.9	17.4

In Table 3, we also illustrate performance as instruction per cycle (IPC) and power consumption as energy per instruction (EPI) using the SimFlex technological parameters. EPI is represented for the base configuration and an Oracle configuration assuming that fetch is stopped as soon as a mispredicted branch is decoded. The overall objective of fetch gating in terms of power consumption can be seen as reducing as much as possible the extra EPI over the oracle configuration while inducing performance loss as small as possible compared with the base configuration.

5.1 Analysis of SIWW

We compare SIWW with pipeline gating by boosting the confidence estimate and by throttling. We measure the benefits of pipeline gating using extra work metrics [14], i.e. the number of wrong-path instructions that pass through a pipeline stage divided by the number of correct-path instructions.

First, Figure 2 compares the three gating techniques on a per benchmark basis on configurations favoring a small performance reduction rather than a large extra fetch reduction. The O-GEHL predictor is used here. SIWW (label "SIWW+CE") incurs less performance degradation than boosting: 0.31% on average compared to 0.79% for boosting. Furthermore, extra fetch work is reduced

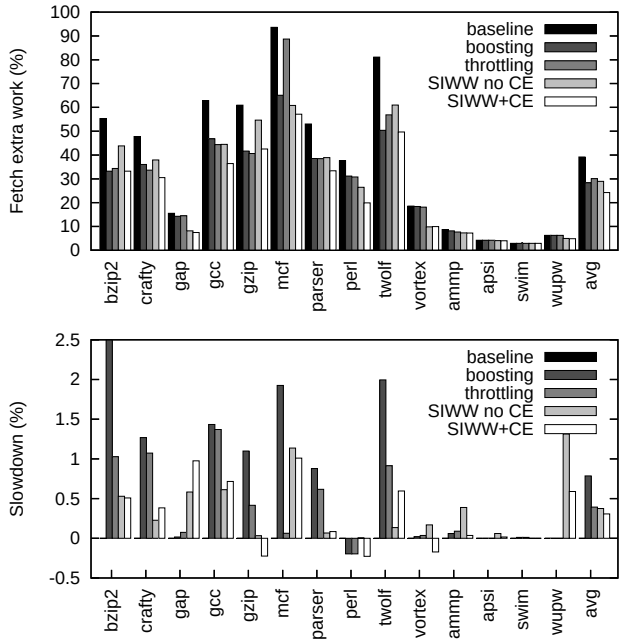

Fig. 2. Comparison between SIWW and boosting for the O-GEHL predictor. The boosting level is 2 low-confidence branches. Throttling fetches only once every two cycles when a low-confidence branch is inflight. The SIWW threshold is 224 ("SIWW no CE") or 160 ("SIWW+CE").

from 39.2% to 24.2% for SIWW vs. 28.4% for boosting level 2. In total, SIWW removes 38.1% of the extra fetch work.

Throttling is known to perform better than boosting. When a low-confidence branch is inflight, fetch is activated once every two cycles. This improves performance slightly over boosting at the expense of a little extra fetch work. However, throttling may be very ineffective for particular benchmarks, e.g. mcf, where hardly any improvement over the baseline is observed.

SIWW works correctly even without using any confidence estimator We run an experiment without using any confidence estimator i.e. assigning the same weight to each indirect or conditional branch. 16 was the assigned weight. On Figure 2, the considered SIW threshold is 224. This configuration of SIWW ("SIWW no CE") achieves average extra fetch work and slowdown very similar to throttling. This is explained by the fact that there is still a correlation between the number of inflight branches and the probability to remain on the correct path. Since the weight of a branch is higher than the weight of a non-branch instruction, SIWW enables fetch gating when the number of inflight branches is high.

SIWW allows to fully exploit the self-confidence estimator Figure 3 illustrates SIWW versus boosting when varying boosting levels and SIWW thresholds. Decreasing the boosting level to 1 significantly decreases the performance by

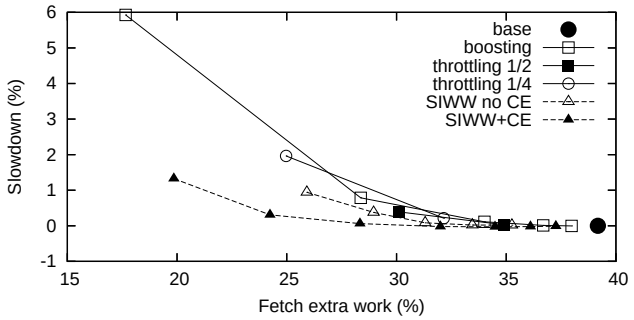

Fig. 3. Varying boosting levels (1 to 5), throttling parameters (threshold 1 and 2, frequency 1/2 and 1/4) and SIWW (thresholds "SIWW no CE" 192 to 320, "SIWW+CE" 128 to 256).

5.9% and reduces the extra fetch work from 39.2% to 17.7%. Therefore with fetch gating based on confidence, the designer has the choice between a limited extra work reduction, but small performance reduction with boosting level 2 or higher, or a large performance degradation, but also a larger extra work reduction with no boosting. This limited choice is associated with intrinsic properties of the self confidence estimator. Manne et al. [14] pointed out that a good trade-off for a confidence estimator for fetch gating based on boosting is high coverage (SPEC) of mispredicted branches (e.g. 80% or higher) and medium *predictive value of a negative test* (PVN)(10-20%). The self-confidence estimator for O-GEHL exhibits medium SPEC and PVN metrics. The SPECint benchmarks show SPEC values in the range of 40%–57% and PVN values above 30%, except for the highly predictable benchmarks gap and vortex.

On the other hand, this property of the self confidence estimator is not a handicap for SIWW. In addition to providing better performance-extra work trade-off than boosting or throttling, SIWW offers the possibility to choose the SIW threshold level in function of the desired performance/extra work trade-off. For instance, with SIWW theshold 128, one sacrifices 1.3% performance but reduces the extra fetch work from 39.2% to 19.9%.

SIWW Works for all Branch Predictors In order to show that fetch gating control through SIWW works for all branch predictors, we analyze SIWW assuming a gshare branch predictor.

Figure 4 shows that SIWW performs better than boosting and throttling both in terms of fetch extra work and in terms of slowdown. For instance SIWW removes more than half of the extra fetch work at a slowdown of 2.1% while boosting level 3 only removes 43% of the extra fetch work but involves a slowdown of 2.6%.

5.2 Dynamic Adaptation of SIW Weight Contributions

The SIWW mechanism is improved by dynamically adapting the SIW weight contributions depending on the predictability of branch conditions and targets in each benchmark. We find that dynamically adapting the SIW weight yields only

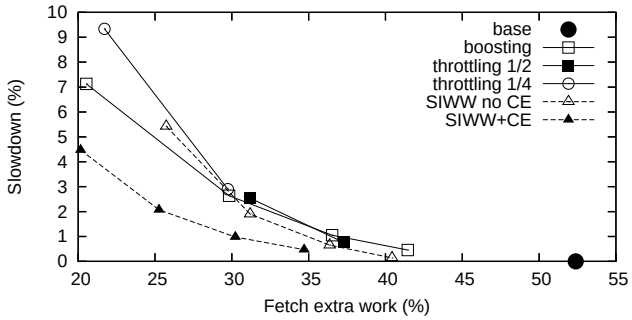

Fig. 4. Gshare branch predictor. Varying boosting levels (2 to 5), throttling parameters (threshold 1 and 2, frequency 1/2 and 1/4) and SIWW thresholds (both cases 128 to 224).

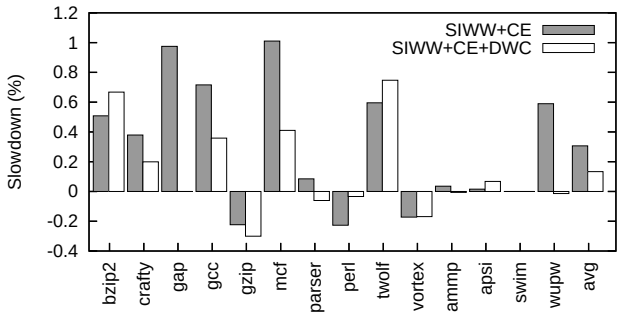

Fig. 5. Slowdown obtained with dynamic weight contributions (DWC). SIW thresholds are 160 (SIWW+CE) and 96 (SIWW+CE+DWC). Both schemes have almost equal extra work metrics. The O-GEHL predictor is used.

small reductions in extra work. On the other hand, it is useful to limit slowdown because decreasing the SIW weight contribution for highly predictable instruction classes avoids unnecesary fetch gating. The lower slowdown also translates into energy savings.

Figure 5 compares SIWW with fixed weight contributions from the previous section (SIWW+CE) to SIWW with dynamically adapted weight contributions. The dynamic adaptation algorithm is effective in reducing slowdown. E.g. slowdown is reduced to almost zero for gap and wupwise. The slowdown is reduced from an average of 0.31% to 0.13%. We found that the extra work metrics change little, e.g. extra fetch work reduces slightly from 24.2% to 24.0%. However, we will see in the following section that dynamic adaptation is effective at reducing energy consumption.

Analysis of the trained weight contributions shows large variations across benchmarks. Furthermore, the difference between the weight contribution for low-confidence branches and high-confidence branches also varies strongly. This difference is small when the PVN of the confidence estimator is small. Then, low-confidence branches are still likely to be predicted correctly and are assigned a lower weight contribution.

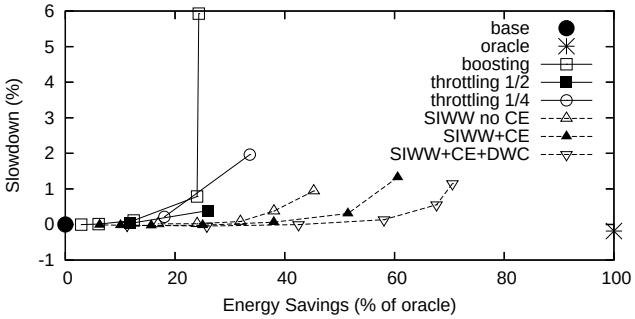

Fig. 6. Reduction of EPI relative to the oracle scheme; the O-GEHL predictor is used.

We attempted to select fixed weight contributions based on the trained values. This, however, yields only small benefits over the fixed weight contributions used throughout this paper.

5.3 SIWW control for fetch gating is Energy-Efficient

Figure 6 illustrates the trade-off between the EPI (energy per committed instruction) reduction and the slowdown. The graph shows the baseline architecture without fetch gating and an oracle fetch gating scheme that gates fetch for all mispredicted instructions. The oracle scheme defines an upper bound on the *total* energy savings obtainable with fetch gating, which is 5.8% in our architecture. The fact that this upper bound on *total* energy savings is quite low simply means that we did a good job at selecting a baseline architecture that is already power-efficient. This is achieved by limiting fetch and issue width to four instructions, by using the highly accurate O-GEHL predictor and by adding history-based branch target prediction.

Within this envelope, SIWW is most effective in realizing an energy reduction. The three variations of SIWW reduce energy in the range of 40–70% for a limited slowdown ($< 1\%$). Previously known techniques, such as throttling and pipeline gating realize no more than 26% of the envelope for the same slowdown.

Note that boosting with level 1 does not save more energy than boosting level 2, this is due to a particularly high loss of performance on a few benchmarks where both performance and power consumption are made worse.

5.4 SIWW and Flow Rate Matching

In the previous sections, we evaluated SIWW for the purpose of gating-off wrong-path instructions. Using the same parameters as in Figure 2, Figure 7 illustrates that SIWW reduces the activity in all pipeline stages. The reduction of activity in the execute stage is small. However, SIWW exhibits the potential to reduce power consumption in the schedule, execute and wake-up stages as the occupancy of the reorder buffer is strongly reduced compared to the baseline, up to 27% for gcc. This property can be leveraged to reduce power further by dynamically scaling the size of the reorder buffer or the issue queue [3,6].

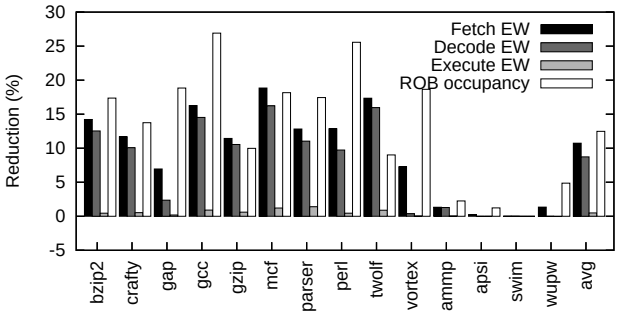

Fig. 7. Reduction of fetch, decode and execute work and reduction of reorder buffer occupancy, using the O-GEHL predictor.

We compare SIWW to PAUTI [3], a parallelism and utilization-based fetch gating mechanism. We assume a non-collapsing issue queue in our baseline processor because of its energy-efficiency [6]. SIWW is not dependent on the issue queue design but PAUTI is specified for a collapsing issue queue [3]. We adapted PAUTI to a non-collapsing issue queue in the following way. During each cycle, PAUTI decides on gating fetch for one cycle based on the issued instructions. If more than half of the issued instructions are issued from the oldest half of the issue queue and the number of issuable instructions in the issue queue exceeds a preset threshold, then fetch is gated during one cycle. Otherwise, fetch is active. We use an issue queue fill threshold of 48 instructions.

The energy-efficiency of a flow-rate matching technique (e.g. PAUTI or SIWW) is amplified by dynamically adapting the issue queue size [3]. The issue queue is scaled using a method we refer to as ADQ [4]. The usable issue queue size is determined at the beginning of a time quantum (e.g. 10000 cycles), depending on the average issue queue utilization during the previous quantum. If the average occupation is less than the usable queue size minus 12, then the usable queue size is reduced by 8. If the average occupation exceeds the usable queue size during the last quantum minus 8, then the usable queue size for the next quantum is increased by 8. The thresholds are chosen such that reducing the issue queue size further would cause an unproportionally large slowdown.

A different trade-off between slowdown and energy consumption is obtained depending on the configuration of the fetch gating scheme (issue queue fill threshold for PAUTI or SIWW threshold). Figure 8 shows that, regardless of the configuration, the SIWW methods achieve higher energy savings for the same slowdown. Analysis shows that SIWW and PAUTI achieve their energy savings in different areas. SIWW removes more fetch stage energy while PAUTI removes more issue stage energy. Total energy savings however, average out to the same values for PAUTI and SIWW with fixed weight contributions when slowdown is restricted to 1% (energy savings are 4.88% and 4.94%, respectively). SIWW with dynamic weight contributions obtains a significantly higher energy reduction (6.5% of total energy) because it removes more fetch extra work than SIWW with fixed weight contributions and it allows for almost the same reduction in the issue queue size as PAUTI.

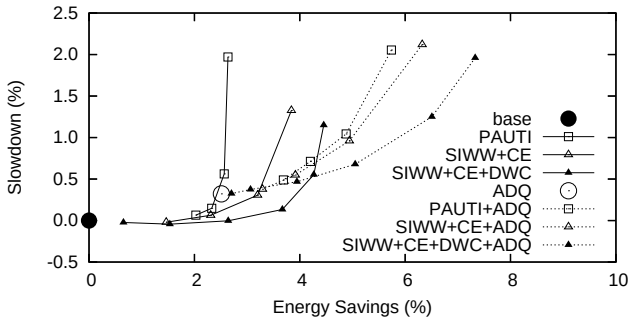

Fig. 8. Energy reduction vs. slowdown for several configurations of each scheme. The issue queue fill thresholds for PAUTI are 40, 48, 56 and 64. The SIWW thresholds are 128, 160, 192 and 224 with fixed weights and 72, 80, 96, 112, 128 and 160 with dynamically adapted weights.

6 Conclusion

Fetch gating improves power-efficiency because of (i) eliminating energy consumption on wrong-path instructions and (ii) matching the front-end instruction rate to the back-end instruction rate.

Previous proposals for wrong-path fetch gating relied only on branch confidence estimation, i.e. counting the number of inflight low-confidence branches. These proposals were not taking into account the structure of the remainder of the speculative instruction window (number of instructions, number of inflight high-confidence branches, ...). SIWW takes this structure into account and therefore allows more accurate decisions for fetch gating. Fetch gating control through SIWW allows to reduce extra work on the wrong path in a more dramatic fashion than fetch gating through confidence boosting and throttling.

Fetch gating mechanisms have been proposed that focus on matching the front-end and back-end instruction flow-rates, neglecting to filter out wrong-path instructions. The SIWW method combines both: by weighting control transfers heavily, wrong-path instructions are gated-off and the front-end flow rate is limited during phases with many hard-to-predict control-transfers.

Future directions for research on SIWW include new usages of SIWW, e.g., optimizing thread usage in SMT processors. We have shown that SIWW limits resource usage by wrong-path instructions, which is very important for SMT processors [13]. Furthermore, by setting a different SIW threshold per thread, different priorities can be assigned to each thread.

Acknowledgements

Hans Vandierendonck is a Post-doctoral Research Fellow with the Fund for Scientific Research-Flanders (FWO-Flanders). Part of this research was performed while Hans Vandierendonck was at IRISA, funded by FWO-Flanders. André Seznec was partially supported by an Intel research grant and an Intel research equipment donation.

References

1. J. L. Aragón, J. González, and A. González. Power-aware control speculation through selective throttling. In *HPCA-9: Proceedings of the 9th international symposium on high-performance computer architecture*, pages 103–112, Feb. 2003.
2. A. Baniasadi and A. Moshovos. Instruction flow-based front-end throttling for power-aware high-performance processors. In *ISLPED '01: Proceedings of the 2001 international symposium on low power electronics and design*, pages 16–21, Aug. 2001.
3. A. Buyuktosunoglu, T. Karkhanis, D. H. Albonesi, and P. Bose. Energy efficient co-adaptive instruction fetch and issue. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 147–156, June 2003.
4. A. Buyuktosunoglu, S. E. Schuster, M. D. Brooks, P. Bose, P. W. Cook, and D. H. Albonesi. A circuit level implementation of an adaptive issue queue for power-aware microprocessors. In *Proceedings of the 11th Great Lakes Symposium on VLSI*, pages 73–78, Mar. 2001.
5. K. Driesen and U. Holzle. The cascaded predictor: Economical and adaptive branch target prediction. In *Proceeding of the 30th Symposium on Microarchitecture*, Dec. 1998.
6. 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.
7. H. Gao and H. Zhou. Adaptive information processing: An effective way to improve perceptron predictors. In *1st Journal of Instruction-Level Parallelism Championship Branch Prediction*, page 4 pages, Dec. 2004.
8. G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 5(1), 2001.
9. E. Jacobsen, E. Rotenberg, and J. Smith. Assigning confidence to conditional branch predictions. In *MICRO 29: Proceedings of the 29th Annual ACM/IEEE International Conference on Microarchitecture*, pages 142–152, Dec. 1996.
10. D. Jiménez. Piecewise linear branch prediction. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 382–393, June 2005.
11. D. A. Jiménez and C. Lin. Composite confidence estimators for enhanced speculation control. Technical Report TR-02-14, Dept. of Computer Sciences, The University of Texas at Austin, Jan. 2002.
12. T. Karkhanis, J. Smith, and P. Bose. Saving energy with just in time instruction delivery. In *Intl. Symposium on Low Power Electronics and Design*, pages 178–183, Aug. 2002.
13. K. Luo, M. Franklin, S. S. Mukherjee, and A. Seznec. Boosting SMT performance by speculation control. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01)*, Apr. 2001.
14. S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: speculation control for energy reduction. In *ISCA '98: Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 132–141, June 1998.
15. D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan. Power issues related to branch prediction. In *HPCA-8: Proceedings of the 8th iInternational Symposium on High-Performance Computer Architecture*, pages 233–246, Feb. 2002.
16. A. Seznec. Analysis of the O-GEometric History Length branch predictor. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 394–405, June 2005.
17. A. Seznec and P. Michaud. A case for (partially) TAgged GEometric history length branch prediction. *Journal of Instruction-Level Parallelism*, Feb. 2006.