Power-aware branch predictor update

A. Baniasadi

Abstract: Designers have invested much effort in developing accurate branch predictors. To maintain accuracy, current processors update the predictor regularly and frequently. Although this aggressive approach helps to achieve high accuracy, for a large number of branches, quite often, updating the branch predictor unit is unnecessary as there is already enough information available to the predictor to predict the branch outcome accurately. Therefore, the current approach appears to be inefficient since it results in unnecessary energy consumption. The author introduces the power-aware branch predictor update (PABU). PABU uses a simple power efficient structure to identify well behaved accurately predicted branch instructions. Once such branches are identified, the predictor is no longer accessed to update the associated data. The key to the success of the proposed technique is a power efficient method that can effectively identify such branches. The author exploits branch instruction behaviour to identify such branch instructions. He shows that it is possible to reduce the number of predictor updates considerably without losing performance. The technique is evaluated by studying energy and performance tradeoffs for SPEC2000 benchmarks. It is shown that the technique can reduce branch prediction energy consumption considerably for both floating point and integer benchmarks. This comes with a negligible impact on performance.

1 Introduction

The goal of this work is to reduce branch predictor energy consumption without compromising accuracy and hence overall performance. Reducing branch predictor energy consumption is important for two reasons: (i) branch predictors already account for a considerable fraction of on-chip dynamic power dissipation; (ii) their power is bound to increase as further improvements in prediction accuracy may call for even larger and more complex branch predictors. The trivial option of reducing energy by using smaller predictors is not acceptable as that would lead to unacceptable accuracy and hence performance degradation. In fact, maintaining and if possible improving prediction accuracy is necessary since it may become essential for future processors to look further in the instruction stream in order to tolerate slower main memories and deeper pipelines. The trend towards larger, more accurate but more energy demanding branch predictors is exemplified by the Alpha processor family: The Alpha EV6 [1] that was released in 1997 used 36 kbits in its branch predictor, while Alpha EV8 [2], which was planned for release after 2001, used 352 kbits – an almost ten-fold increase.

Most state-of-the-art branch predictors use variations of a branch target buffer (BTB [Note 1]) [3] for target address prediction and of the combined branch predictor [4] for direction prediction. Accordingly, we focus on this organisation in this work.

The key opportunity for reducing power in state-of-the-art branch predictors lies in that they were developed with accuracy, speed, complexity and cost as the primary considerations. This tradeoff favours uniformity. Naturally, existing predictors treat all branches uniformly performing the same number of steps per branch. These steps are powerful enough to capture the behaviour of as many branches as possible. However, as we show in this work, quite often, not all steps are needed to accurately predict the branch outcome for all branches. The goal of this work is to exploit this variation and reduce predictor power by selectively updating the predictor for well behaved branches.

Understanding how we can reduce branch predictor power requires a closer look at the underlying principle of operation for state-of-the-art predictors. Specifically modern predictors are history-based. They record branch behaviour implicitly assuming that past behaviour is a good indicator of future behaviour. Processors fill in the branch predictor tables as soon as the outcome of a branch instruction is decided, effectively learning how the specific branch behaves (this is called the learning phase). Then, provided that the branch exhibits relatively stable behaviour, highly accurate prediction is possible simply by looking up the previous outcomes.

A key motivating observation for this work is that there are many well behaved branches for which, once the learning phase is over, the collected information stays virtually unchanged for a period much longer than the learning phase. We refer to this post-learning phase as the steady state phase. Existing designs continuously update all predictor structures even in the steady state. This is unnecessary. By detecting well behaved branches it should be possible to avoid updating the predictor.

We introduce the power-aware predictor update (PABU) to reduce power dissipation while maintaining performance. PABU uses a power efficient structure, referred to as the PABU-filter, to identify well behaved branches that are in their steady state. PABU exploits branch instruction steady state behaviour and temporal locality. Once a branch is

IEE Proceedings online no. 20045117

doi: 10.1049/ip-cdt:20045117

Paper received 6th September 2004

The author is with the Electrical & Computer Engineering, University of Victoria, 3800 Finnerty Rd., Victoria, British Columbic, Canada V8P 5C2

E-mail: amirali@ece.uvic.ca

Note 1: Line prediction is a similar in nature alternative to the BTB.

deemed as one in its steady state, updating the predictor is avoided.

Specifically, by using PABU, while maintaining performance cost within 0.1%, we can:

- reduce the number of BTB updates by up to a maximum of 82%
- reduce the number of predictor buffer updates by up to a maximum of 80%
- reduce branch prediction energy consumption for a variety of predictor sizes by up to 50%.

Previous study has suggested techniques that reduce power dissipation in the entire processor (e.g. voltage and frequency scaling). However, improving power dissipation in future designs is most likely to follow an incremental path. This may require revisiting different processor components and optimising them for better power efficiency. In a conventional processor each component's power dissipation accounts for only a fraction of the total power dissipation. Nonetheless, revisiting traditional designs and developing power-aware components provides a reliable opportunity for reducing power in future processors. To facilitate this goal, power-aware components should achieve considerable local savings while imposing very little performance cost.

Note that PABU introduces new hardware which will impose overhead. In general, adding new hardware may reduce energy, if the new addition saves energy spent elsewhere. As such, PABU will only be justified if savings exceed the overhead.

In this work we are concerned with power reduction techniques at the architectural level that are complementary to low-level circuit techniques. Our goal is to revisit existing architectural decisions, revising where necessary so that energy is used more efficiently.

2 Power-aware predictor update

To enhance performance, many processors use the branch predictor aggressively. While this has resulted in high branch prediction accuracy, unfortunately it comes with extra power dissipation. By using WATTCH [5] we estimated that the power consumed by a 32k-entry combined branch predictor and a 1k-entry, 4-way, BTB of a 4-way issue, 64-entry window superscalar processor is 11.6% of the total processor switching power. Moreover, branch predictor accuracy impacts the overall processor power dissipation. Reportedly, replacing complex power hungry branch predictors with small and simple ones will result in higher overall power dissipation due to an increase in the number of mispredicted instructions executed [6]. Accordingly, in this work we focus on low power and highly accurate predictors.

PABU relies on the following three observations:

1. At fetch, many modern processors access the BTB using the instruction address of the fetched instruction – a possible branch – to index the buffer. If the branch address is found in the BTB, we know the predicted instruction address at the end of the fetch cycle, which is at least one cycle earlier than for a branch prediction buffer. This requires storing taken branch addresses and their target address in the buffer. Accordingly, we update the BTB frequently and as soon as we know the target address and are certain that the branch is taken. However, our study on SPEC2000 benchmarks shows that more than 99% of the BTB updates are unnecessary since the associated branch address is already stored in the BTB. In other words, since many branches appear frequently, all but the first update are unnecessary as long as the branch is not evicted from the buffer. However, many processors (e.g. [7]), update BTB for every taken branch to ensure that the branch information is stored.
2. Several high-performance designs (e.g. [8, 9]) use the combined predictor, which is one of the most accurate predictors [4], to find out the branch outcome. Combined predictors use three underlying subpredictors. Two of the subpredictors produce predictions for branches. They are typically tuned for different branch behaviours. The third subpredictor is the selector and keeps track of which of the two subpredictors works best per branch. Typical configurations use bimodal predictors for one of the subpredictors and the selector, and a pattern-based predictor like gshare for the last subpredictor. Bimodal predictors capture temporal direction biases (e.g. mostly taken) in branch behaviour and have short learning times. However, they cannot capture complex repeating branch behaviours that do not exhibit a direction bias. Pattern-based predictors like gshare can successfully capture repeating direction patterns and correlated behaviours across different branches. However, these predictors require longer learning times and have larger storage demands. By using a selector, combined predictors offer the best of both underlying subpredictors: fast learning and prediction for branches that exhibit temporal bias, and slower but accurate prediction for branches with repeatable direction patterns. The subpredictors use saturating counters to record information. A typical mechanism in such predictors is to increment the associated counter if the branch is taken and decrement it if it is not taken. To reduce the number of bits required, small counters (e.g. 2-bit counters) are used. Once the counter value has reached the maximum (e.g. three), taken branch outcomes will no longer increment the counter. Similarly, once the counter has reached its minimum, not taken branch outcomes will not change the predictor state. In other words, the counter is not decremented past the minimum, nor it is incremented past the maximum. Later, the counters are probed to predict the branch outcome. If the counter value is more than a threshold (e.g. one for 2-bit counters) the branch is predicted taken. Otherwise, the branch outcome is predicted to be not taken.

This approach, while providing accurate predictions, is inefficient since many (i.e. $\sim$ 90%, as we show later) of the subpredictor updates are unnecessary as they attempt to increment/ decrement a counter that is already saturated to the maximum/minimum. For example, in the case of loops with more than hundreds of iterations, only the first and last few predictor updates provide useful information. The rest of the updates do not change the state of the counters associated with the loop branch instruction.

We categorise predictor buffer updates into three major groups. The first group includes those updates that do not change the predictor state. An example is an update caused by a taken branch whose associated counter is already saturated to the maximum value. We refer to such updates as noneffective updates (NEUs). The second group of updates includes those that change the predictor state but not the immediate outcome. An example of such updates is an update caused by a taken branch whose associated counter is already more than the threshold (e.g. the counter value is two) but less than the maximum value (e.g. three). While such updates do not change the immediate predictor outcome they may impact future decisions. We refer to such updates as probably effective updates (PEUs). The third group of updates includes those that change both the

predictor state and immediate outcome. An example of such updates is an update causing a counter to pass the threshold. We refer to such updates as effective updates (EUs).

Figure 1 reports the percentage of each kind of update for SPEC2000 benchmarks and for all three predictor tables (abbreviations are shown under the 'Ab.' column in Table 1). In this Figure (and for all Figures presented in this paper) the first four benchmarks from the left are floating point benchmarks while the rest of the benchmarks are integer. We also report harmonic average measurements for floating point and integer benchmarks separately to provide deeper insight. For each benchmark, bars from left to right report relative distribution for NEUs, PEUs and EUs. On average, 87% and 97% of the updates appear to be noneffective (NEUs) for integer and floating point benchmarks, respectively. This reaches a minimum of 78% for *wlf* and a maximum of 99.8% for *mes*. On average, and for integer benchmarks, only 1.8% and 3.6% of the total number of updates are PEUs and EUs, respectively. For floating point benchmarks, on average, EUs and PEUs account for less than 1%. We conclude from Fig. 1 that only a very low percentage of predictor buffer updates contribute to performance. On average, 87% and 97% of the updates result in power dissipation without contributing to performance for integer and floating point benchmarks, respectively.

3. We have observed that branch instructions show strong temporal locality. That is, during short periods of time, there is a small set of branches that account for the vast majority of predictions [10]. Therefore, it may be possible to use a small structure to store information regarding the branch instructions being executed during short intervals.

PABU uses the discussed phenomena to identify and eliminate unnecessary branch predictor updates. To do so and to take advantage of temporal locality, PABU uses the 256-entry PABU-filter presented in Fig. 2 (in Section 3.4.2 we will discuss how alternative filter sizes impact PABU).

For each dynamic branch instruction, we access the PABU-filter prior to the original predictor to decide if updating the branch predictor is unnecessary. If so, we save power by not accessing the branch predictor structures. Otherwise, we access the original predictor. We store well behaved branch instructions that are in their steady state in the PABU-filter. We do so by taking into account the counters associated with the branch in the branch predictor buffer. If all three predictor counters are strongly biased (i.e. they are saturated), we allocate an entry in the PABU-filter.

Every filter entry includes an address field and a two-bit hint field. PABU uses the latter to record whether a branch is already in the BTB, and if the branch outcome was predicted accurately last time encountered. As presented in Fig. 2, the following information is stored in the PABU-filter:

The first field, Branch PC, is used to store the branch address whose associated counters are saturated.

The second field, BTB, is a single bit used to record if the branch is already stored in the BTB. When we find a well behaved branch present in the BTB we set this bit to one. Later, and before updating the BTB, we check this bit to decide whether updating the BTB is necessary.

The third field, valid, is initially (i.e. when the branch is stored in the filter) set to zero. An accurately predicted branch sets the 'valid' bit to one. If the valid bit associated with the branch is 0, none of the predictor accesses is avoided.

PABU avoids updating all three tables in the predictor buffer if: (*a*) the branch is found in the filter (i.e. associated counters are saturated) and (*b*) the valid bit is 1 (i.e. the branch was predicted accurately last time encountered).

Similarly, PABU avoids updating the BTB if: (*a*) the branch is found in the filter and (*b*) the valid bit is 1 and (*c*) the BTB bit is set to 1 (i.e. the branch is already in the BTB).

Mispredicted branches, if found in the PABU-filter, are removed from it. This is to ensure that the predictor is regularly updated for problematic branches.

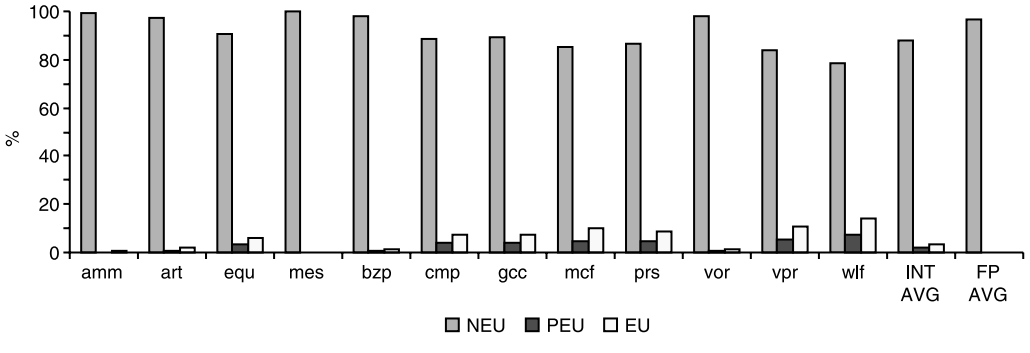

Fig. 1 *Bars from left to right show percentage of NEU, PEU and EUs for a subset of SPEC2k benchmarks*
Abbreviations are shown under the 'Ab.' column in Table 1

Table 1: Benchmark names, abbreviations, BTB (target address) accuracy and branch prediction (direction) accuracy

Program	Ab.	BTB acc., %	BP acc., %	Program	Ab.	BTB acc., %	BP acc., %
ammp	amm	99.3	99.4	*mcf*	mcf	91.8	92.5
art	art	98.2	98.2	*mesa*	mes	99.9	99.9
bzip	bzp	98.7	98.7	*parser*	prs	91.7	93.9
compress	cmp	93.5	94	*vortex*	vor	98.2	99.1
equake	equ	96	96.2	*vpr*	vpr	91.3	91.9
gcc	gcc	90.5	93.9	*wolf*	wlf	86.8	87.7

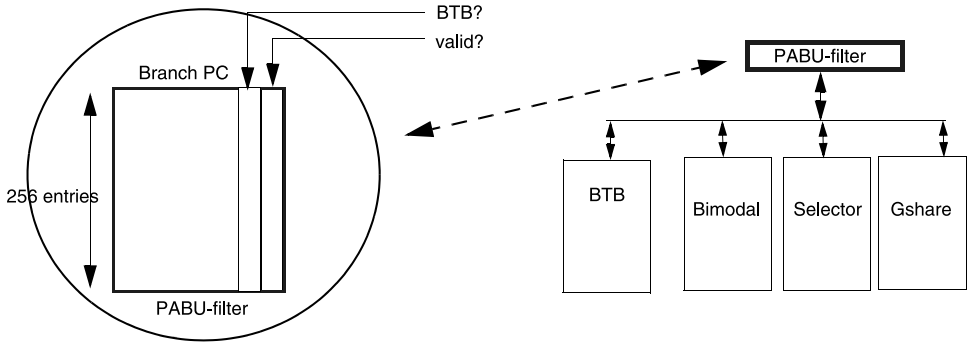

Fig. 2 *PABU-filter*

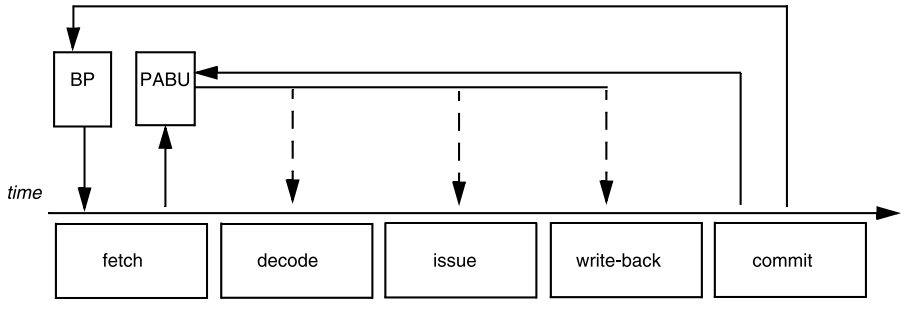

Fig. 3 *Schematic showing PABU operation*

Provided that a sufficient number of well behaved branches are accurately identified, PABU has the potential for reducing branch prediction energy consumption. However, it introduces extra energy overhead and can, in principle, increase overall energy consumption if the necessary behaviour is not there. We take into account this overhead in our study and show that for the programs we studied PABU is robust.

Figure 3 shows the operation of PABU. Initially, we access the branch predictor to see whether the counters associated with the branch are saturated. If that is the case, and if the branch is not already in the filter, we allocate an entry in the PABU-filter. Later, and when the branch outcome is known, we update the PABU-filter. We either set the valid bit or remove the branch from the filter, depending on the prediction accuracy. The decision regarding avoiding the update could be taken anytime from fetch to commit, i.e. when the branch is scheduled to update the predictor. We investigated different timing scenarios where the PABU-filter was probed at decode, issue or write-back. The impact of this variation on performance and energy savings was insignificant.

The timing overhead associated with PABU should not be an issue since the information needed to decide whether the update could be avoided is available far earlier than when the branch would update the predictor.

3 Methodology and results

In this Section, we present our analysis of PABU.

We used programs from the SPEC2000 suite complied for the MIPS-like architecture used by the Simplescalar v3.0 simulation tool set [11], and GNU's gcc complier (flags: -O2-funroll-loops-finline-functions). Table 1 reports the branch prediction and BTB accuracy per benchmark when a 32K-entry branch predictor is used. For brevity we use the abbreviations shown under the 'Ab.' column. We simulated one billion instructions, and picked four floating point (*amm*, *art*, *mes* and *equ*) and eight integer benchmarks to investigate how PABU impacts both integer and floating point benchmarks. We also report harmonic average for both floating point and integer benchmarks. We detail the base processor model in Table 2.

To investigate how PABU impacts energy and performance for different predictor sizes, we study combined predictors with the following configurations:

- 32k-entry gshare, bimodal and selector, 1k, 4-way entry BTB
- 16k-entry gshare, bimodal and selector, 512, 4-way entry BTB
- 8k-entry gshare, bimodal and selector, 256, 4-way entry BTB
- 4k-entry gshare, bimodal and selector, 256, 4-way entry BTB.

The gshare predictor used in all predictors uses 8-bit history. Note that all the tables used to store information, i.e. predictor buffers, BTB and PABU-filter, use a memory core of SRAM cells accessed via row and column decoders.

We used WATTCH [5] for energy estimation. WATTCH is a widely used architectural simulator that estimates CPU energy consumption (both dynamic and leakage). Power estimations are based on a suite of parameterisable power models of different hardware structures and on per-cycle resource usage counts generated through cycle-level simulation. To model a typical modern processor we modelled a 2 GHz superscalar microarchitecture manufactured under a 0.1 μm technology. To estimate the relevant process parameters, we used the process scaling methodology developed for CACTI [12] that is incorporated in WATTCH.

By using WATTCH we have estimated how much energy consumption accessing each of the branch predictor structures requires. This includes the energy consumed to access BTB, local and global predictors and the selector. In Table 3 we report the average energy consumed by each structure separately when each structure is accessed. We also report the relative energy consumed by accessing the PABU-filter. Accordingly, the energy overhead associated

Table 2: Base processor configuration

Branch predictors studied in this work	32k GShare + 32k bi-modal w/ 32k selector, 1024-entry 4-way BTB
	16k GShare + 16k bi-modal w/ 16k selector, 512-entry 4-way BTB
	8k GShare + 8k bi-modal w/ 8k selector, 512-entry 4-way BTB
	4k GShare + 4k bi-modal w/ 4k selector, 256-entry 4-way BTB
Scheduler	64 entries, RUU-like
Fetch unit	up to 4 instr./ cycle. Max 1 branches/cycle 64-entry fetch buffer
OOO core	4-way processors: any 4 instructions/cycle
Func. unit latencies	same as MIPS R10000
L1 - instruction/data caches	64k, 4-way SA, 32-byte blocks, 3-cycle hit latency
Unified L2	256k, 4-way SA, 64-byte blocks, 16-cycle hit latency
Main memory	Infinite, 100 cycles

Table 3: Energy consumption (in nj) for each structure used in 32k, 16k, 8k, 4k-entry predictors used here and the PABU-filter

Energy consumption per access	Lookup	Update
4k-entry local predictor	8.5	11.1
4k-entry global predictor	11.9	15.2
4k-selector	11.9	15.2
8k-entry local predictor	15.3	18.6
8k-entry global predictor	17.4	21.3
8k-selector	17.4	21.3
16k-entry local predictor	24.5	31.3
16k-entry global predictor	32.5	41.5
16k-selector	32.5	41.5
32k-entry local predictor	41.8	53.1
32k-entry global predictor	50.9	62.4
32k-selector	50.9	62.4
1k, 4-way BTB	202.1	248.6
512, 4-way BTB	111.5	136.3
256, 4-way BTB	62.1	76.4
256-entry PABU-filter	1.4	1.8

with the PABU-filter is far less than the energy consumed by BTB or by the predictor buffer. Nevertheless, we will benefit from PABU if the energy saved by eliminating predictor accesses outweighs that consumed by extra PABU accesses. Therefore, through this study we take into account the associated costs.

Unless stated otherwise, power results reported here are based on the nonideal clock-gating style used by WATTCH where gated units dissipate 10% of the maximum power ('cc3'). However, many studies have suggested that the absolute and the relative contribution of leakage power to the total system power is expected to further increase in future technologies because of the exponential increase in leakage currents with technology scaling [13–15]. Therefore, we also report power savings where gated units dissipate 30% of the maximum power later in Section 3.4.1.

3.1 Update and access frequency

In this Section we report how PABU impacts BTB and predictor buffer update and access frequency. We limit our attention to the 32k-entry branch predictor (predictor size will not affect its access frequency but it does impact greatly the energy cost of each access).

Before reporting access and update frequency for a PABU-enhanced processor we report predictor frequency access for a conventional predictor in Fig. 4. We also include how often the PABU-filter is accessed for the PABU-enhanced processor. As reported for both the BTB and the predictor buffer the number of predictor lookups exceeds the number of updates. This is the result of the fact that many instructions are flushed due to branch mispredictions. As such, many branch instructions reading the predictor do not commit and therefore never update the predictor.

In the case of the PABU-filter, however, the number of updates exceeds the number of the lookups. This is due to the fact that for every branch instruction we may update the PABU-filter more than once. For example, we may update the filter initially when we find out that the branch is in steady state, and then remove the branch later on when we realise that it has been mispredicted.

In Fig. 5 we report the reduction in the total number of BTB updates and accesses achieved by PABU. The entire bar shows the reduction in update frequency. The dark portion of each bar reports total access reduction. The maximum update reduction is 82% for *art* and the minimum is 38% for *wlf*. Almost all (i.e. more than 99.5%) of the eliminated BTB updates are unnecessary ones. We have observed that the percentage of useful BTB updates removed is less than 1%. On average, BTB update reduction is 43% and 62% for integer and floating point benchmarks, respectively. On average, BTB access reduction is 16% and 31% for integer and floating point benchmarks, respectively.

In Fig. 6 we report predictor buffer update and access frequency reduction. In Fig. 6*a* bars from left to right report the relative reduction in the total number of updates, NEUs, PEUs and EUs, respectively. On average, we reduce the total number of updates and NEUs by 47% and 53% for integer benchmarks and by 60% and 64% for floating point benchmarks, respectively. PABU does not impact the number of EUs and PEUs considerably.

The fact that we eliminate a considerable number of NEUs with very little impact on the number of EUs is important since it shows that PABU successfully identifies and eliminates unnecessary updates.

In Fig. 6*b* we report the relative reduction in the total number of predictor accesses. On average, total predictor access is reduced by 13% and 26% for integer and floating point benchmarks, respectively.

Note that, among all benchmarks, *wlf* and *art* show the lowest and highest BTB and predictor buffer update frequency reductions, respectively. This is consistent with results reported earlier in Fig. 1 where *wlf* has the lowest number of NEUs while *art* has the highest.

PABU does not eliminate all unnecessary updates, for two reasons. First, due to size restriction, sometimes well

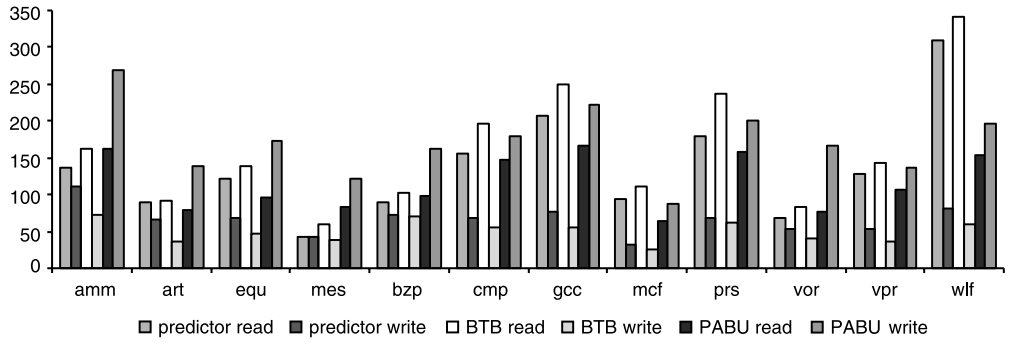

Fig. 4 *Number of times the predictor is read or written*
Bars from left to right report for predictor read, predictor write, BTB read, BTB write, PABU read and PABU write. Numbers are in million instructions

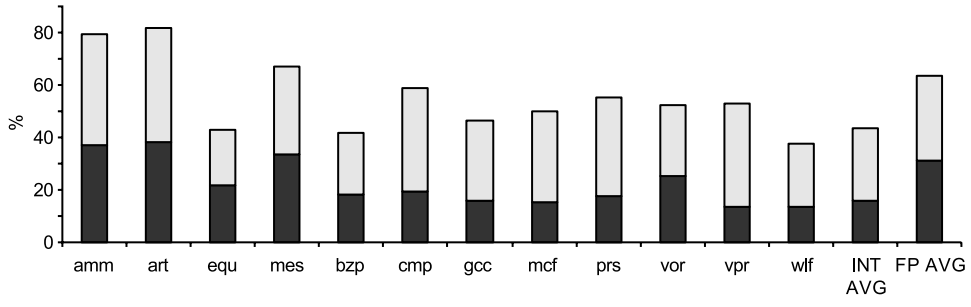

Fig. 5 *Relative reduction in total number of BTB updates (entire bar) and BTB accesses (dark portion) for a PABU-enhanced processor*

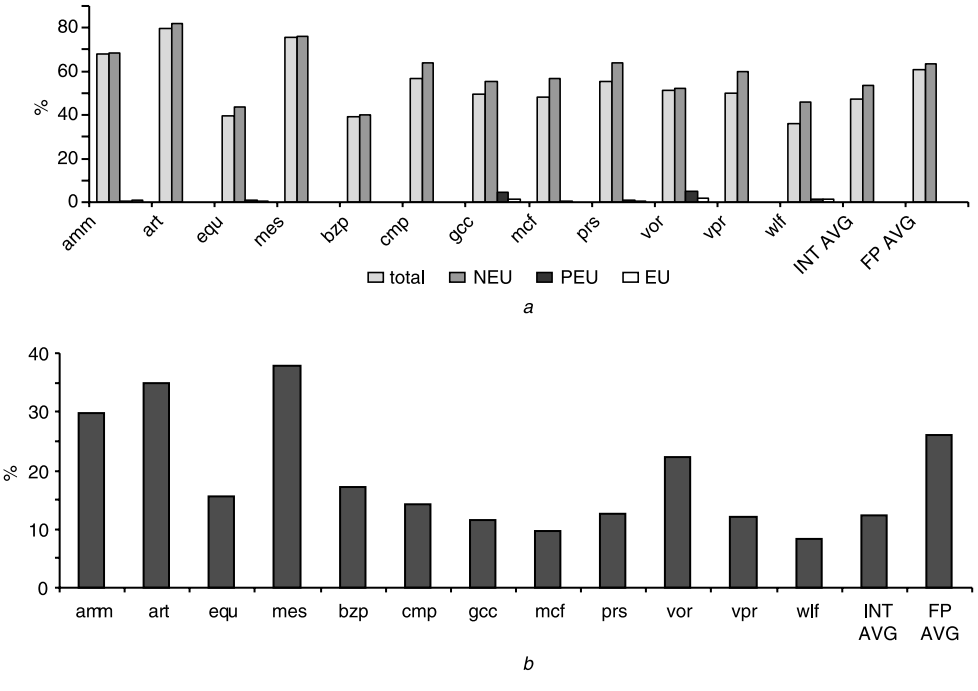

Fig. 6 *Relative reduction in predictor buffer updates and total number of predictor buffer accesses for a PABU-enhanced predictor*
a Updates
b Total accesses
Bars from left to right report for total number of updates, NEUs, PEUs and EUs

behaved branches are evicted from the PABU-filter. Second, we only eliminate updates that are associated with branches that have all three counters saturated. This is only a subset of the NEUs since not all three updates are always unnecessary. In other words, PABU eliminates updates for branches with three unnecessary updates. This leaves out branches with one or two NEUs.

We also conclude from Figs. 5 and 6 that PABU performs more effectively for floating point benchmarks. This is due to the fact that floating point benchmarks have a larger number of well behaved branches (more about this is discussed later).

3.2 Performance

Although we only eliminate predictor updates for branches that are already being predicted accurately, PABU may negatively impact accuracy and hence performance. This could happen since branches may change their behaviour

through time. A typical scenario is that a branch may follow the taken path frequently before changing direction. Under such circumstances, PABU will repeatedly avoid updating the predictor (and follow the taken path) until the mispredicted branch is detected and removed from the filter. Meantime, several mispredictions may take place and impact performance negatively. To determine whether PABU is indeed worthwhile, in Fig. 7 we compare PABU-enhanced processors with conventional processors that do not eliminate any of the predictor updates. Numbers lower than 100% represent slowdowns. Bars from left to right report performance for processors using 32k, 16k, 8k and 4k-entry branch predictors. Maximum performance loss is 0.06% and is observed for *vpr* and for the 8k-entry predictor.

Note that we witness performance improvement for some of the integer benchmarks and for some of the predictor sizes and for one of the floating point benchmarks, i.e. *amm* in the case of the 4k-entry predictor. This may be the result of reducing destructive aliasing as we filter some of the unnecessary updates to the predictor. This especially becomes more important for smaller predictors where the probability of aliasing is higher. On average, integer benchmarks show performance improvement for all predictors but the 32k-entry predictor.

This result is significantly important since it shows that in cases where PABU does degrade performance, the cost is very little. Moreover, for integer benchmarks, where deconstructive aliasing can be critical, PABU can improve performance.

3.3 Energy and power

In this Section we report predictor energy and power measurements. While reducing the number of predictor updates reduces predictors' energy consumption, it could potentially increase the overall energy consumption due to a possible increase in the number of mispredicted instructions. Therefore, we also study overall energy consumption.

Figure 8 reports branch predictor energy reduction and the PABU energy overhead for PABU-enhanced predictors and for different predictor sizes. In Fig. 8*a* we report predictor energy reduction. Note that for every benchmark each bar reports savings for a PABU-enhanced processor compared to a conventional processor that uses a branch predictor similar to that used by the PABU-enhanced processor. Average energy savings are between 20% and 25% for integer benchmarks across all predictor sizes. For floating point benchmarks, average savings are between 23% and 42% for different predictor sizes. As expected, floating point benchmarks show higher savings. This is consistent with results reported earlier where access reduction was higher for floating point benchmarks. Note that predictor energy reduction is higher for smaller

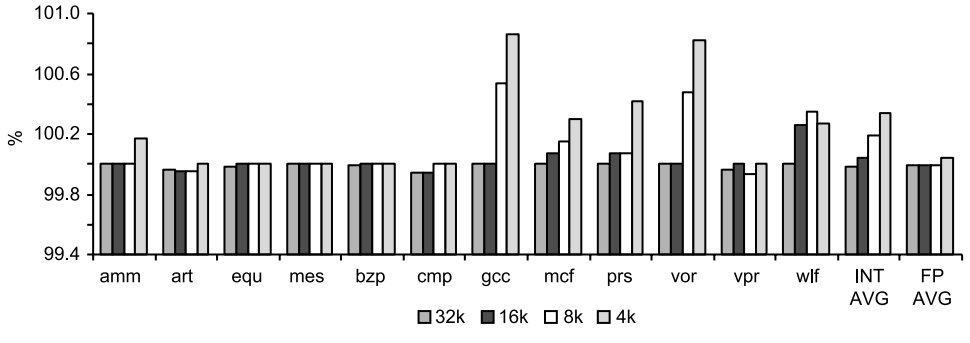

Fig. 7 *Performance for PABU-enhanced processors (higher is better)*
Bars from left to right report performance for processors using 32k, 16k, 8k and 4k-entry branch predictors

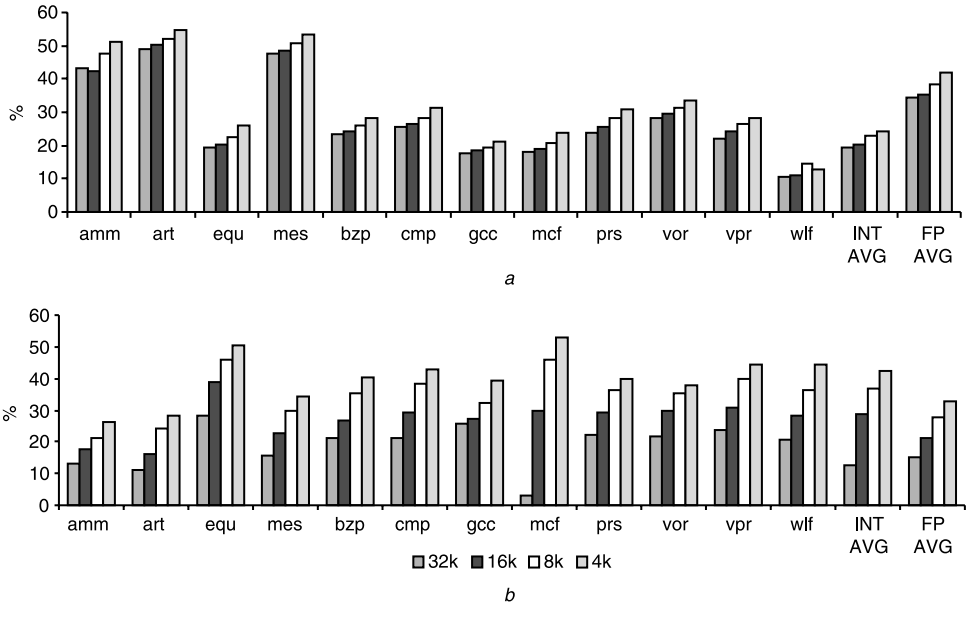

Fig. 8 *Relative branch prediction energy consumption reduction for PABU-enhanced processors and overhead associated with PABU*
a Relative reduction
b Overhead

predictor sizes. This may be due to the fact that for smaller predictors we also reduce the number of mispredicted instruction by reducing deconstructive aliasing. *Mes*, *art* and *amm* (all floating point benchmarks) show higher savings compared to other benchmarks.

To provide better insight, in Fig. 8*b* we report the energy overhead associated with PABU compared with the predictor energy reduction. In other words, the data reported in Fig. 8*b* show how much more our savings would have been if the overhead associated with PABU would not have negated part of our savings. As reported, the smaller the predictor, the more the relative PABU power overhead will be. On average, PABU overhead negates 12%, 28%, 37% and 42% of the savings achieved originally for processors using 32k, 16k, 8k and 4k-entry predictors for integer benchmarks. The PABU overhead negates 15%, 21%, 27% and 33% of the savings achieved originally for processors using 32k, 16k, 8k and 4k-entry predictors for floating point benchmarks.

In Fig. 9 we report overall energy reduction for different predictor sizes. As pointed out earlier, overall savings should be weighted against the negligible performance cost. On average, energy reduction for integer benchmarks is 2.7%, 1.9%, 1.4% and 1.2% for processors using 32k, 16k, 8k and 4k-entry predictors, respectively. Average energy consumption for floating point benchmarks is 3.2%, 2.2%, 1.6% and 1.3% for processors using 32k, 16k, 8k and 4k-entry predictors, respectively. As predictors become smaller they consume a smaller share of the total energy. Therefore, total energy savings start to decline as smaller predictors are used.

Across all predictor sizes, *amm* and *art*, with overall savings more than 5%, show higher savings compared to others.

We conclude from Figs. 8 and 9 that PABU reduces predictor and overall energy consumption across all benchmarks. Also, in relative terms, PABU reduces predictor energy more effectively for smaller predictors. In the meantime, overall energy savings are higher for predictors using larger predictors.

We have observed that power and energy/delay measurements follow the exact trend of energy measurements. This is the result of the fact PABU has negligible impact on runtime; consequently, power (energy over runtime) and energy/delay follow energy. Therefore, we only report energy.

3.4 Sensitivity analysis

In this Section, we investigate the sensitivity of PABU to key parameters. In Section 3.4.1, we investigate our energy

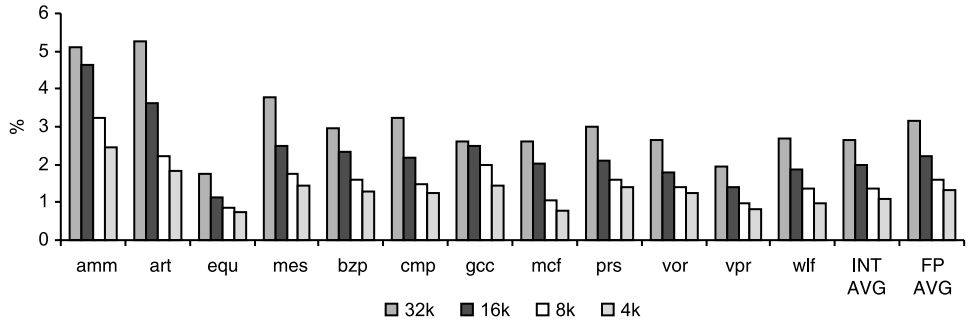

Fig. 9 *Total energy reduction for PABU-enhanced processors*
Bars from left to right report for processors using 32k, 16k, 8k and 4k-entry branch predictors

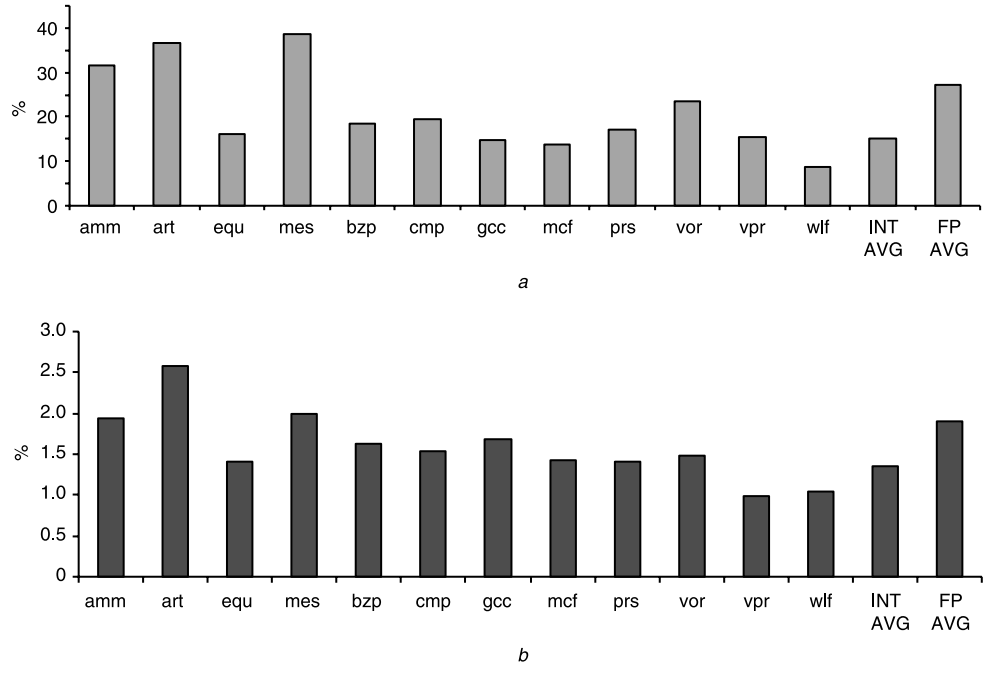

Fig. 10 *Energy savings when leakage power is 30% of total power*
a Predictor energy
b Total energy

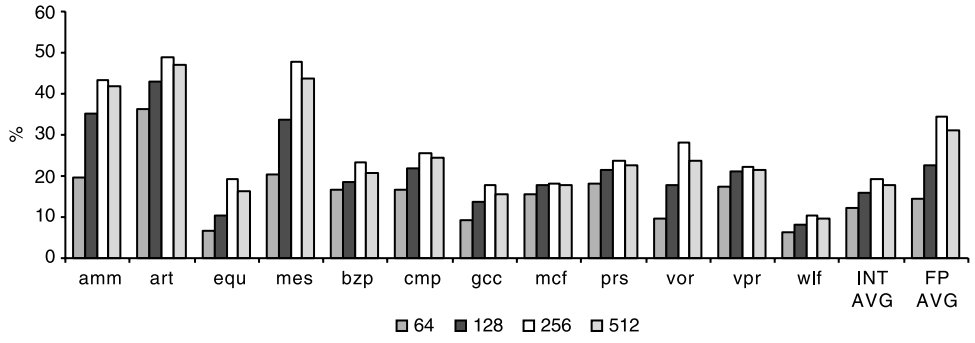

Fig. 11 *Predictor energy reduction for different 1 PABU-filter sizes*

results if gated units dissipate 30% of the maximum power. In Section 3.4.2 we vary PABU-filter size and show how it impacts energy savings.

3.4.1 Leakage power: As many studies have suggested [13–15] in future technologies, the problem of subthreshold leakage power in CMOS circuits will grow in significance. When transistors are switched off, a certain amount of leakage current flows, and this results in leakage power. The leakage current is exponentially dependent on the value of the threshold voltage such that if the threshold voltage is reduced (as it will in future technologies), the leakage current registers.

To study how this will impact our results, we investigate how PABU will perform when gated units dissipate 30% of the maximum power. Accordingly, in Fig. 10 we report predictor and overall energy savings when leakage power is 30% of the maximum power and for a 32k-entry predictor. On average, predictor energy reduction is 15% and 26% for integer and floating point benchmarks, respectively. Average total energy savings are 1.4% and 1.9% for integer and floating point benchmarks, respectively. The fact that PABU aims at reducing dynamic power explains why savings start to decrease as leakage power increases.

3.4.2 PABU-filter size: In this Section we report how changes in the PABU-filter size impact our results. We report predictor energy savings when the filter size is 64, 128, 256 and 512. We do not report performance as it stays virtually intact when the PABU-filter size is changed. Figure 11 reports energy savings for a 32k-entry predictor. As presented energy savings are maximum when a 256-entry filter is used. This is true for both floating point and integer benchmarks. Meantime, floating point benchmarks appear to be more sensitive to filter size compared to integer benchmarks.

4 Discussion

In this Section we review the results and discuss details. We review both floating point and integer benchmarks.

1. Floating point benchmarks include *amm*, *art*, *equ* and *mes*. Among the four floating point benchmarks, *amm*, *art* and *mes* have the highest update frequency, predictor and total energy reduction (see Figs. 6, 8*a* and 9) and therefore benefit most from PABU. A combination of factors explains why the three stand out. First, the three have a high percentage of NEUs (more than 97% as reported in Fig. 1). Therefore there is a large pool of unnecessary predictor updates to pick from. Second, all three benchmarks have a high branch prediction rate (more than 98% as reported in Table 1). This indicates that they have a large number of well behaved branches.

Among the three, *mes* falls behind in total energy reduction (see Fig. 9). To explain this we need to take into account branch instruction frequency. Therefore in Fig. 12 we report the average number of instructions per branch (IPB). Note that high IPB indicates low branch frequency. As presented, IPB is highest for *mes*. In other words, *mes* has the least number of branches. Consequently, for *mes*, a smaller share of total energy is consumed by the branch predictor compared to other benchmarks. This results in lower total energy reduction for this benchmark. Also, *amm* has the lowest IPB, which may explain why *amm* has one of the highest total energy reductions among all benchmarks.

Among the floating point benchmarks, *equ* performs worst. The following explains why *equ* falls behind other floating point benchmarks:

First, *equ* has a lower branch prediction rate (96%) compared to the other three floating point benchmarks. The other three benchmarks have prediction rates higher than 98%.

Second, *equ* has the lowest percentage of NEUs. While 40% of the predictor updates are NEUs for *equ*, for the other three floating point benchmarks more that two-third of the updates are unnecessary.

Third, *equ* has the highest PABU-filter eviction rate. To explain this better, in Fig. 13 we report how often branch instructions are evicted from the PABU-filter. Here we report the percentage of dynamic branches that are evicted or replaced by another branch. As reported, *equ* has the highest rate (62%). Accordingly, as a result of deconstructive interference, many well behaved branches whose associated updates could potentially be eliminated are evicted from the PABU-filter.

2. The main reason explaining why integer benchmarks do not perform as well as the three better performing floating point benchmarks is lower branch prediction rates (see Table 1). All but two of the integer benchmarks, i.e. *bzp* and *vor*, fall behind floating point benchmarks in branch prediction rate.

Integer benchmarks, based on how they benefit from PABU, fall into two categories:

(*a*) The first group includes *bzp*, *vor*, *cmp*, *prs* and *vpr*. This group reduces predictor energy by at least 20% across all predictor sizes. However, they fall behind when compared to the three better performing floating point benchmarks (*amm*, *art* and *mes*) as their energy reductions are lower. A collection of factors including branch prediction rates, low number of NEUs, high IPBs and high PABU eviction rates explain this. For example, *prs*, *cmp* and *vpr* have a lower number of well behaved branches as indicated by their lower branch prediction rate. *Bzp*, on the other hand,

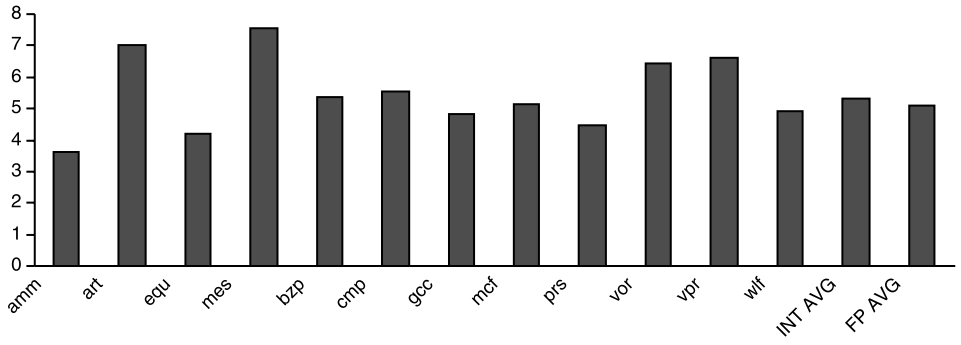

Fig. 12 *Instruction per branch*

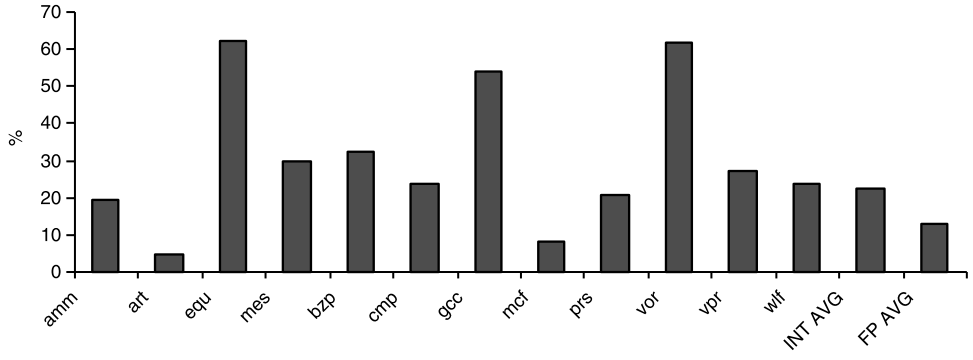

Fig. 13 *How often branches are evicted from PABU-filter*

despite having a high branch prediction rate, comes with a small number of NEUs compared to *amm*, *art* and *mes* (see Fig. 1). Finally, *vor*, again despite having a high branch prediction rate, has a high PABU eviction rate (see Fig. 13). (*b*) The second group of integer benchmarks includes *gcc*, *mcf* and *wlf*. This group has lower savings compared to the first group of integer benchmarks. Again, different factors explain why each benchmark achieves lower energy savings. For example, the fact that *wlf* has the lowest number of NEUs (see Fig. 1) and lower than average branch prediction rate (see Table 1) among all benchmarks may explain why it benefits least from PABU. Low branch prediction rate also explains why *mcf* belongs to this category. *Gcc*, on the other hand, has a higher than average eviction rate (see Fig. 13), which results in less energy saving.

5 Related work

Previous work has introduced Banking [6], Predictor Probe Detection (PPD) [6], Branch Predictor Prediction (BPP) [10] and Branch Predictor Customising [15] as methods to reduce branch predictor energy consumption while maintaining performance.

Banking is a natural solution to multicycle access times for large on-chip structures such as branch predictors. Our solution could be used on top of banking to achieve higher power savings.

PPD aims at reducing the power dissipated during predictor lookups. PPD identifies when a cache line has no conditional branches so that a lookup in the predictor buffer can be avoided. Also, it identifies when a cache line has no control-flow instructions at all, so that the BTB lookup can be eliminated. This is done by storing pre-decoded bits in a structure called the prediction probe detector. PABU is different from PPD since it eliminates unnecessary predictor updates. Also, PABU uses a smaller overhead compared to PPD.

BPP exploits branch instruction behaviour to gate two out of the three subpredictors. In BPP, a buffer is introduced in the fetch stage. Each BPP entry is tagged by the PC of a recently seen dynamic branch and records the subpredictors used by the two branches that followed it in the dynamic execution stream. By comparing PABU to BPP we have observed that PABU achieves higher energy savings with lower energy costs. This is mainly due to the fact that, unlike PABU, BPP focuses on the prediction buffer and does not reduce the energy consumed by the BTB. Also, PABU is different from BPP since it reduces the number of updates, while BPP only focuses on the lookups.

Branch predictor customising [15] applies structure resizing and access gating to create a customised branch predictor. Accordingly, in this technique software is used to exploit on-demand resources utilisation in branch predictors to customise the predictor according to its resource demands. PABU does not use adaptive resizing and depends on steady state branch behaviour which is collected dynamically. Predictor customising on the other hand, uses a profile-based approach. Our solution can be used on top of predictor customising, resulting in further savings.

Using saturating counters as branch confidence estimators was first suggested by Smith [16]. Manne *et al*. [17] suggested the 'both strong' estimation method, which marks a branch as high confidence only if the saturating counters for both gshare and bimodal predictors are in a strong state and have the same predicted direction (taken or not-taken). We extend their technique by marking a branch as one in its steady state phases if all three counters used in the combined predictor are saturated.

6 Conclusion

We presented PABU, a technique for reducing branch predictor energy consumption while maintaining the accuracy advantage of combined branch predictors.

We affirmed that it is possible to significantly reduce branch predictor energy consumption by identifying and eliminating the branch predictor updates that do not contribute to performance.

Our study showed that PABU reduces predictor and overall energy consumption for both floating point and integer benchmarks and across different predictor sizes. We also observed that PABU produces better results for applications with a higher number of well behaved branches.

We have shown that when one considers the overall processor energy consumption, PABU-enhanced processors always consume less energy when compared to ones that use the conventional predictor. Because of the considerable energy savings and the relatively small cost, PABU is an attractive power-aware enhancement for high-performance processors.

7 References

1 Leibholz, D., and Razdan, R.: 'The Alpha 21264: a 500 MHz out-of-order execution microprocessor'. COMPCON, 1997
2 Seznec, A., Lix, S.F., Krishnam, V., and Sazeides, Y.: 'Design tradeoffs for the Alpha EV8 conditional branch predictor'. Proc. 29th Int. Symp. on Computer Architecture, May 2002
3 Perleberg, C.H., and Smith, A.J.: 'Branch target buffer design and optimization', *IEEE Trans. Comput.*, 1993, **42**, pp. 396–412
4 McFarling, S.: 'Combining branch predictors'. Tech. Note TN-36, DECWRL, June 1993
5 Brooks, D., Tiwari, V., and Martonosi, M.: 'Wattch: a framework for architectural-level power analysis and optimizations'. Proc. Int. Symp. on Computer Architecture, 2000
6 Parikh, D., Skadron, K., Zhang, Y., Barcella, M., and Stan, M.R.: 'Power issues related to branch prediction'. Proc. Int. Symp. on High-Performance Computer Architecture, February 2002
7 Diep, T.A., Nelson, C., and Shen, J.P.: 'Performance evaluation of the PowerPC 620 microarchitecture'. Proc. Int. Symp. on Computer Architecture, June 1995
8 Digital Semiconductor. DECchip 21064/21064A Alpha AXP Microprocessors: Hardware Reference Manual, June 1994.
9 Digital Semiconductor. Alpha 21164 Microprocessor: Hardware Reference Manual, April 1995.
10 Baniasadi, A., and Moshovos, A.: 'Branch predictor prediction – a power-aware branch predictor for high-performance processors'. Proc. Int. Conf. on Computer Design, September 2002
11 Burger, D.C., and Austin, T.M.: 'The SimpleScalar tool set, version 2.0', *Comput. Archit. News*, 1997, **25**, (3), pp. 13–25
12 Wilton, S., and Jouppi, N.: 'CACTI: an enhanced access and cycle time model for on-chip caches'. WRL Research Report 93/5, DEC Western Research Laboratory, 1994. SIA. International technology roadmap for semiconductors. Technical report, http://public.itrs.net/
13 Borkar, S.: 'Design challenges of technology scaling', *IEEE Micro*, 1999, **19**, (4), pp. 23–29
14 Kam, T., Rawat, S., Kirkpatrick, D., Roy, R., Spirakis, G.S., Sherwani, N., and Peterson, C.: 'EDA challenges facing future microprocessor design', *IEEE Trans. Comput.-Aided Des.*, 2000, **19**, (12), pp. 1498–1506
15 Huang, M.C., Chaver, D., Pinuel, L., Prieto, M., and Tirado, F.: 'Customizing the branch predictor to reduce complexity and energy consumption', *IEEE Micro*, 2003
16 Smith, J.E.: 'A study of branch prediction strategies'. Ann. Int. Symp. on Computer Architecture, May 1981
17 Manne, S., Klauser, A., and Grunwald, D.: 'Pipeline gating: speculation control for energy reduction'. Proc. Int. Symp. on Computer Architecture, June 1998