SEPAS: A Highly Accurate Energy-Efficient Branch Predictor

Amirali Baniasadi
Electrical and Computer Engineering
University of Victoria
amirali@ece.uvic.ca

Andreas Moshovos
Electrical and Computer Engineering
University of Toronto
moshovos@eecg.toronto.edu

ABSTRACT

Designers have invested much effort in developing accurate branch predictors with short learning periods. Such techniques rely on exploiting complex and relatively large structures. Although exploiting such structures is necessary to achieve high accuracy and fast learning, once the short learning phase is over, a simple structure can efficiently predict the branch outcome for the majority of branches. Moreover, for a large number of branches, once the branch reaches the steady state phase, updating the branch predictor unit is unnecessary since there is already enough information available to the predictor to predict the branch outcome accurately. Therefore, aggressive usage of complex large branch predictors appears to be inefficient since it results in unnecessary energy consumption.

In this work we introduce Selective Predictor Access (SEPAS) to exploit this design inefficiency. SEPAS uses a simple power efficient structure to identify well behaved branch instructions that are in their steady state phase. Once such branches are identified, the predictor is no longer accessed to predict their outcome or to update the associated data. We show that it is possible to reduce the number of predictor accesses and energy consumption considerably with a negligible performance loss (worst case 0.25%).

Categories and Subject Descriptors

C.1.1 [Single Data Stream Architectures] Pipeline processors.

General Terms

Design

Keywords

Power-Aware Branch Prediction, Selective Predictor Access, High-Performance Processors.

1. INTRODUCTION

The goal of this work is to reduce branch predictor energy consumption without harming accuracy and hence overall performance. Reducing branch predictor energy consumption is important for two reasons: First, branch predictors already account for a large fraction of on-chip dynamic power dissipation (as much as 10% [1]). Second, 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 it would lead to unacceptable accuracy and hence performance degradation. In fact, maintaining and if possible improving prediction accuracy is essential for future processors that will be required to look further ahead into the instruction stream in order to tolerate slower main memories and keep deeper pipelines busy. The trend towards larger, more accurate and more energy demanding branch predictors is exemplified by the Alpha processor family: The Alpha EV6 [10] that was released in 1997 used 36Kbits in its branch predictor while Alpha EV8 [16] which was planned for release after 2001 used 352Kbits, an almost ten-fold increase.

Most state-of-the-art branch predictors use variations of the branch target buffer (BTB[1]) [11] for target address prediction and of the combined branch predictor [7,8] for direction prediction. Accordingly, we focus on this organization. The key opportunity for reducing power in state-of-the-art branch predictors lies in that they were developed with accuracy, speed, cost and complexity as the primary considerations. This trade-off favors uniformity. Naturally, existing predictors treat all branches uniformly performing the same actions per branch. These actions are powerful enough to capture the behavior of as many branches as possible. However, it is well understood that some branches are well behaved and as such easier to predict than others. Moreover, some branches are predicted well enough using only one of the underlying predictors of a full-blown combined predictor. The goal of this work is to exploit this variation in predictability in order to reduce predictor power. This is achieved by selectively using simpler and smaller predictors for well behaved branches and by avoiding accesses to all predictors when only one of them is good enough for predicting a specific branch.

We introduce *Selective Predictor Access* (SEPAS), a power efficient technique that exploits branch behavior that reduces predictor energy in two ways. First, by identifying well behaved branches SEPAS uses a tiny structure to predict them accurately and to avoid updating the larger, power-hungry underlying predictor. Second, by identifying branches that are predicted well using only one of the predictors, SEPAS avoids two out of the three predictor accesses. SEPAS is an *architectural* level technique that is complementary to low-level circuit techniques. SEPAS exploits branch instruction steady state behavior and temporal locality in the instruction stream. It is important to note that SEPAS is not a branch predictor cache. A branch predictor cache in this case would simply cache a few entries from the underlying predictors reducing power by filtering some predictor probes and by aggregating predictor updates to the same row into a single writeback update. SEPAS differs from a branch prediction cache in two ways: First, SEPAS completely hides well behaved branches from the underlying predictor (*i.e.*, no predictor or BTB probes or updates occur for such branches). Second, SEPAS selectively

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

ISLPED'04, August 9-11, 2004, Newport Beach, California, USA.

deactivates two out of three predictors when it deems that this will not harm accuracy.

Using a set of the SPEC CPU2000 benchmarks and the WATTCH power models [5] we find that with a worst-case performance penalty of just 0.25% SEPAS can reduce the average number of predictor lookups and updates by 38% and 55% respectively. Since SEPAS introduces an additional structure this may harm overall prediction latency. Accordingly, we study two different possible timing scenarios: accessing SEPAS in-series with or in-parallel to the underlying predictor. We show that while energy savings are higher when SEPAS is accessed in-series with the predictor, if timing restrictions do not allow serial access, by accessing SEPAS in-parallel to the predictor, we still reduce predictor energy considerably as we can selectively shut down part of the predictor accesses (*e.g.*, after bitline precharge and decode).

The rest of this paper is as follows. In section 2 we discuss our motivation. In section 3 we introduce SEPAS and present the details. In section 4 we discuss methodology and results. We study how SEPAS impacts access frequency, performance and energy consumption. In section 5 we review related work. Finally, in section 6 we offer our concluding remarks

2. MOTIVATION

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 behavior implicitly assuming that past behavior is a good indicator of future behavior. 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 behavior 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 rely on continuous usage of all predictor structures even in the steady state. This is unnecessary. By detecting these branches it should be possible to avoid updating the combined predictor. Furthermore, because typical programs exhibit temporal locality in their branch stream, predicting well behaved branches should be possible via a small structure without harming accuracy.Throughout this paper, we refer to branches that are in their steady state phase as *SP branches.* Also we refer to the branches that are in their learning phase as *LP branches.*

Besides the energy inefficiencies in handling branches with a long steady state phase, existing predictors are inefficient in they way they handle another class of branch instructions. Specifically, while some branches do not exhibit long steady state phases their behavior is simple enough to be captured by only one of the two underlying predictors. Identifying such branches would allow us to avoid two out of the three probes thus reducing power even when a branch is not extremely well behaved.

In more detail, the power overheads incurred by existing combined branch predictors are the following:

a) The combined predictor uses three underlying sub-predictors to achieve high prediction accuracy. Two of the sub-predictors produce predictions for branches. They are typically tuned for different branch behaviors. The third sub-predictor is the *selector* and keeps track of which of the two sub-predictors works best per branch. Typical configurations, use *bi-modal* predictors for one of the sub-predictors and the selector, and a pattern-based predictor like *gshare* [8] for the last sub-predictor. The sub-predictors use saturating counters to record information. A typical learning mechanism is to increment/decrement the associated counter if the branch is taken/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, 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. 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 is predicted to be not taken.

This approach, while providing accurate predictions, comes with two major inefficiencies. First, our study shows that more than 92% of the sub-predictor updates are unnecessary as they attempt to increment/decrement a counter that is already saturated to the maximum/minimum. Second, while using multiple large branch predictors helps achieving high performance, for most branches, once they reach the steady state phase, a simple and small predictor can accurately predict the branch outcome.

b) Branch instructions exhibit strong temporal locality. That is, looking over short periods of time, there is a small set of branches that account for the vast majority of predictions. We have observed that on average, about 83% of the branches appear within the last 64 branches fetched.

c) Some branches tend to use the same sub-predictor repeatedly in the combined predictor. On average, 95% of the time, branch instructions use the same sub-predictor they used last time they were encountered.

d) Finally, at fetch we access the BTB to check if it contains the branch address. 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 we are certain that the branch is taken. However, our study shows that more than 99% of the BTB updates are unnecessary since the associated branch address is already stored in the BTB. Nevertheless, many processors, access BTB for every taken branch [9].

Using WATTCH [5] we estimated that the power consumed by a 32k-entry combined branch predictor and a 1k-entry, 4-way, BTB is about 10% of the total processor switching power for a a 4-way issue, 64-entry window superscalar processor. Moreover, branch predictor accuracy impacts the overall processor power dissipation. Reportedly, replacing complex power hungry branch predictors with small and simple ones results in higher overall power dissipation due to an increase in the number of mispredicted instructions executed [1]. Accordingly, in this work we focus on low power and highly accurate predictors. *Therefore, while we studied a variety of approaches we only report those that maintain performance cost within 0.25%.*

3. SELECTIVE PREDICTOR ACCESS

SEPAS aims at identifying SP branches and at eliminating the corresponding predictor accesses. Predictor accesses are either lookups or updates. As branch instructions are encountered the predictor is accessed to speculate the branch's outcome (lookup). Later when the branch outcome is known the predictor is accessed again to store the information (update). We have observed that, for the benchmarks studied here, more than 60% of the predictor accesses are lookups. Note that many branches

do not update the predictor as they are squashed due to earlier mispredictions.

Figure 1 shows the organization of the SEPAS-filter structure. It is easier to reason about SEPAS if we assume that for each dynamic branch instruction, we access the SEPAS-filter prior to the original predictor (we may access them in parallel to avoid increasing latency). Accessing SEPAS-filters allows us to decide if the branch is a well-behaved SP. If so, we save power by not accessing the branch predictor. If the branch is not present in the SEPAS-filter we access the original predictor. If the predictor counters associated with the branch missing from the SEPAS-filter are strongly biased (*i.e.*, all three counters are saturated), we allocate an entry in the SEPAS-filter. In other words, we consider a branch with non-saturated counters to be an LP branch. A branch is removed from the filer either because of limited space or because it is mispredicted.

We used a 256-entry direct-mapped SEPAS-filter after testing many alternatives. We have observed that storing 256 entries provides adequate information with affordable overhead. While smaller filters tend to miss energy saving opportunities, larger filters results in unjustifiable overhead. Every filter entry includes an address field and a five-bit *hint* field. SEPAS uses the latter to record whether a branch is already in the BTB, the last sub-predictor used by the branch, the number of consecutive times the branch was predicted taken and if the branch outcome was predicted accurately. As shown in figure 1, the following information is stored in the SEPAS-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 an SP branch present in the BTB we set this bit to one. The third field, "*sub-predictor*", is a single bit that is used to store the last sub-predictor (gshare or bimodal) used by the branches that are already stored in the filter. While a branch is in the learning phase, we probe the selector to pick either gshare or bimodal. Once the branch reaches the steady state phase, the selected sub-predictor is stored in the "sub-predictor" bit for future reference (0 for bimodal, 1 for gshare). The fourth field, *"taken"*, is a two bit counter used to count the number of consecutive times an SP branch is predicted taken. Every taken branch increments the associated counter while a single not taken branch will reset the counter to zero. Consequently, as we explain later, we eliminate predictor lookups only for branches that steadily follow the taken path. The fifth 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 are avoided. If sufficient number of SP branches are accurately identified, SEPAS can reduce branch prediction energy consumption. However, it introduces energy overhead and can potentially increase overall energy consumption. We take into account this overhead and show that for the programs we studied SEPAS is robust.

3.1 Eliminating Unnecessary Lookups

SEPAS eliminates lookups as follows:

(1) When a branch reaches the steady state phase, quite often (*e.g.*, in loops with high number of iterations), it follows the taken direction repeatedly. Accordingly, in this mechanism we remove lookups for repeatedly taken branches.

While combined predictors probe three entries per branch, for such well-behaved branches, the three-entry overhead is unnecessary. In order to eliminate unnecessary predictor lookups, we check the SEPAS-filter. If *a)* the branch is found in the filter and *b)* the branch has been taken for three times consecutively (*i.e.*, taken counter is "11") and *c)* it has been predicted accurately (*i.e.*, valid

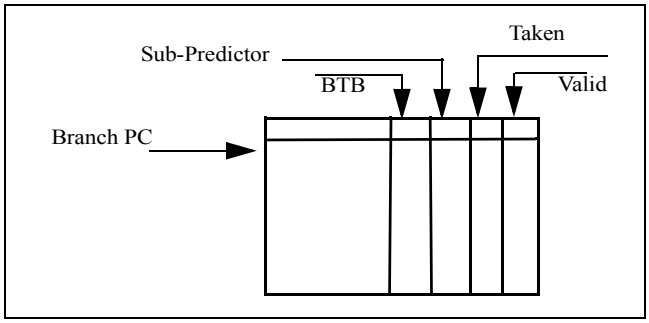

Figure 1: SEPAS-filter.

is 1) then we assume that the branch is taken and do not probe the branch predictor. Alternatively, if the branch is missing from the filter or the valid bit is zero or the taken counter is less than 11 we probe the original combined predictor. We refer to this mechanism as the *complete lookup elimination (CLE)*.

Note that while we focus on taken branches in this mechanism, an alternative is to eliminate lookups for repeatedly not-taken branches. We do not investigate this alternative since not-taken branches are less frequent than taken branches.

(2) We observed that 95% of the time, branch predictors use the same sub-predictor they used last time. Consequently, for 95% of the predictor lookups, two of the three accesses could be avoided if information regarding the previously used sub-predictor is exploited. In this method, as an SP branch (*i.e.*, a branch with all three counters saturated) is predicted, the sub-predictor used to make the prediction is stored in the associated "sub-predictor" bit in the SEPAS-filter. The next time the branch is encountered, if the branch is in the filter and has been predicted accurately (*i.e.*, "valid" is 1) then we assume that the "sub-predictor" bit is a reliable pointer to the appropriate sub-predictor to probe. We refer to this method as the *partial lookup elimination (PLE)*.

We find CLE to be a more efficient approach compared to PLE as it a) eliminates more unnecessary lookups (as we show later) and b) could potentially reduce predictor latency as it eliminates probing all sub-predictors.

3.2 Eliminating Unnecessary Updates

We observed that more than 99% of the BTB updates and 92% of predictor buffer updates are unnecessary. SEPAS aims at identifying and eliminating such updates. Accordingly, we avoid updating the predictor buffer if: *a)* the branch is found in the filter (*i.e.*, the associated counters are saturated) and *b)* the valid bit is 1 (*i.e.*, the branch was predicted accurately last time encountered). Similarly, we avoid 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 (meaning, the branch is already in the BTB).

Intuitively, we avoid updating the predictor if there is already enough information available to the predictor to predict the branch outcome and target accurately. Note that the required information is available as early as the fetch-stage. Therefore, the decision regarding whether the update associated with a branch should take place can be made far earlier than when the branch reaches the commit stage.

3.3 Latency Considerations

The timing overhead associated with SEPAS during an update should is not an issue. This is due to the fact that the information needed to decide if the update can be avoided is available as early as the fetch-stage (which is far earlier than when the branch would update the predictor). However, SEPAS can potentially increase

lookup latency. The key issue is whether there is sufficient time available in the corresponding pipeline stage to so that SEPAS can be accessed in-series with the original predictor (this would result in maximum power savings). We consider the following two timing scenarios: a) Serial Scenario: SEPAS is fast enough so accessing the SEPAS-filter prior to the branch predictor does not impact the front-end latency. b) Parallel Scenario: SEPAS can not be serially accessed within the same cycle as the combined predictor. Therefore, at lookup, to avoid an increase in fetch latency, we access the SEPAS-filter and the predictor in parallel. By using CACTI [12] we have estimated that under these assumptions SEPAS can abort a 32k-entry predictor (the largest we considered) access after the decode. Accesses to smaller predictors can be aborted only after bitline discharge. So, if we must avoid increasing front-end latency at all costs, SEPAS can still reduce power by terminating some probes early enough.

4. METHODOLOGY and RESULTS

In this section, we present our analysis of SEPAS. We report predictor access frequency in 4.1. We report performance results in section 4.2. We report energy measurements in section 4.3. As our study shows that power measurements follow the same trend of energy measurements, we do not report power in the interest of space.

We used programs from the SPEC CPU2000 suite compiled for the MIPS-like architecture used by the Simplescalar v3.0 simulation tool set [13]. We used GNU's gcc compiler (flags: -O2 –funroll-loops –finline-functions). Table 1 reports the branch prediction accuracy per benchmark for a 32k-entry combined predictor (this includes all control flow instructions and takes into account not only direction prediction but also target prediction). In the interest of space, we use the abbreviations shown under the "Ab." column. We simulated half a billion instructions. We detail the base processor model in table 2.

Table 1: Benchmarks and control flow prediction accuracy

Program	Ab.	BP Acc.	Program	Ab.	BP Acc.
ammp	amm	99%	***mesa***	mes	99%
bzip	bzp	98%	***parser***	prs	91%
gcc	gcc	85%	***vortex***	vor	93%
mcf	mcf	92%	***vpr***	vpr	91%

Table 2: Base processor configuration.

Scheduler	64 entries, RUU-like
Fetch Unit	Up to 4 instr./cycle. Max 2 branches/cycle 64-Entry Fetch Buffer
Load/Store Queue	32 entries: 2 loads/stores per cycle Perfect disambiguation
OOO Core	any 4 instructions / cycle
Func. Unit Latencies	same as MIPS R10000
L1 - Inst, /Data Caches	64K, 4-way SA, 32-byte blocks, 3 cycle hit
Unified L2	256K, 4-way SA, 64-byte blks, 16-cycle hit
Main Memory	Infinite, 100 cycles

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

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

The gshare predictor used in all predictors uses 8-bit history. We used WATTCH [5] for energy estimation. To estimate the relevant process parameters, we used the process scaling methodology developed for CACTI [12] and that is incorporated in WATTCH. Note that all the tables used to store information, *i.e.*, predictor buffers, BTB and the SEPAS-filter, use a memory core of SRAM cells accessed via row and column decoders.

4.1 Access Frequency

In figure 2 we report how SEPAS reduces the number of predictor lookups and updates. For this experiment we limit out attention to 32k-entry branch predictors (predictor size does not affect access frequency but it does impact greatly the per access energy cost). In figure 2, bars from left to right report relative reduction in: (1) the number of lookups achieved by CLE alone, (2) the number of lookups achieved by using both CLE and PLE, (3) the updates, and (4) the BTB accesses. On average we reduce the number of lookups and updates by more than 30% and 50% respectively. While both CLE and PLE contribute to elimination of predictor lookups, CLE alone can eliminate the majority of the unnecessary lookups detected by SPEAS.

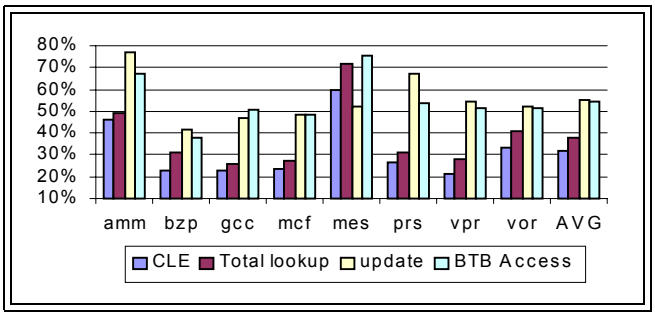

Figure 2: a) Access frequency reduction. Bars from left to right show CLE lookups, total lookups, updates, and BTB accesses.

4.2 Accuracy and Performance

SEPAS can negatively impact accuracy and hence performance. Accordingly, in figure 3 we report performance with SEPAS relative to the baseline processor. Numbers lower than 100% represent slowdowns. Bars from left to right report performance for a 4-way processor using 32k, 16k, 8k and 4k-entry branch predictors. As reported, performance slowdown is below 0.25% across all benchmarks for all predictors. The maximum performance loss of 0.23% is observed for *mcf* and for the 32k and 16k predictors. This result demonstrates that the performance cost associated with SEPAS is negligible.

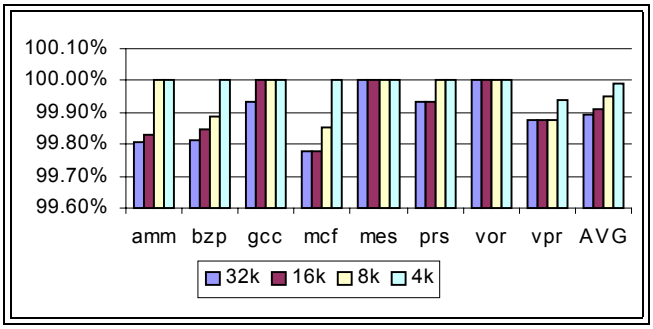

Figure 3: Performance for a SEPAS-enhanced processor compared to a processor using a conventional combined branch predictor (higher is better).

4.3 Energy

An obvious alternative way of avoiding unnecessary updates is to check the predictor entry before updating. In this solution, the predictor will be updated only if the new value is different from the old one. We refer to this solution as the *Update on Change* (UOC) policy. While we assume this does not impact performance and access frequency, it will impact our energy savings. Therefore in this section we compare energy both to the original base case and the OA case.

In figure 4 we report predictor energy savings and a breakdown of energy reduction compared to processors using each of the update policies and under each timing scenario (discussed in 3.3). In parts (a) and (b) we compare to the BASE and the UOC machine respectively. BASE is the processor that always updates its predictors while UOC exploits the *Update on Change* policy.

In both 4(a) and 4(b) bars from left to right report for processors with 32k, 16k, 8k and 4k-entry predictors. Each bar has three parts and reports energy savings and breakdown for the serial scenario. Under the serial scenario, the entire bar represents total savings. Meantime the lowest part represents savings achieved by eliminating updates. The remaining (*i.e.*, the sum of the middle and highest part), represents lookup energy savings.

We use only two of the three parts (*i.e*., the lowest and the middle part) in each bar to report savings for the parallel scenario. Under this scenario, the highest part is irrelevant. Similar to the serial scenario, the lowest part reports update energy savings. However, only the middle part represents lookup savings. Consequently, total energy savings is represented by the sum of the middle and the lowest part.

In figure 4, one can look at the highest part in each bar as the relative increase in energy savings if we switch from the parallel scenario to the serial scenario.

4.3.1 Energy Savings Over BASE

Comparing to the BASE machine in part (a) we observe:

Serial Scenario: On average, predictor energy reduction is almost 35% for all predictors. Two thirds of the savings is the result of eliminating updates while the remaining one third is the result of lookup elimination.

Parallel Scenario: On average, predictor energy reduction is about 30% for all predictors. Approximately 75% of the savings is the result of eliminating updates while the remaining 25% is the result of lookup elimination.

4.3.2 Energy Savings Over UOC

Comparing to the UOC machine in part (b) we observe:

Serial Scenario: On average, predictor energy reduction is almost 25% for all predictors. About 40% of the savings is the result of eliminating updates while 60% is the result of removing lookups.

Parallel Scenario: On average, predictor energy reduction is between 18% and 21% for different predictors. Maximum and minimum share of updates is 64% (in the 4k-entry) and 50% (in the 32k-entry predictor).

4.3.3 Energy Savings Discussion

Note that in figures 4(a) and 4(b) while the update savings remains the same under both scenarios, the lookup fraction is lower under the parallel scenario. This is the result of the fact that the scenarios impact lookups and not updates.

As reported in figure 4, *amm* and *mes* benefit a lot more than the other benchmarks from SEPAS. There are two reasons why this is the case: First, we have observed that *amm* and *mes* have high percentage of useless updates (more than 97% as our study shows). Therefore there is a large pool of unnecessary predictor updates to pick from. Second, both benchmarks have a high branch prediction accuracy and a large number of well-behaved branches.

In figure 5 we report total energy reduction. In both figures the lower part of each bar represents SEPAS energy savings compared to the UOC machine while the entire bar shows our saving compared to the BASE machine. In figure 5 we assume the serial and parallel scenarios respectively and observe the following:

Under the serial scenario (part (a)) and compared to the BASE, average total energy reduction (entire bar) is 3.4%, 2.2%, 1.4% and 1.2% for processors using 32k, 16k, 8k and 4k-entry predictors respectively. When comparing to UOC (lower bar), this is 2.1%, 1.4%, 0.9% and 0.8% for processors using 32k, 16k, 8k and 4k-entry predictors respectively.

Under the parallel scenario (part (b)) and compared to the BASE, average total energy reduction is 2.3%, 1.6%, 1% and 0.8% for processors using 32k, 16k, 8k and 4k-entry predictor. When comparing to the UOC machine (lower bar), average total energy reduction is 1.7%, 0.9%, 0.6% and 0.5% for processors using 32k, 16k, 8k and 4k-entry predictors.

Overall energy savings results are consistent with the predictor energy savings, *i.e.*, energy reduction is higher under the serial scenario compared to the parallel scenario. However, total energy savings start to decline as smaller predictors are used. As predictors become smaller they consume a smaller share of the total energy.

Note that for *mes*, while the reduction in predictor energy is highest among all benchmarks, we do not observe a considerable reduction in overall energy. Our study shows that *mes* has the lowest number of branches among all benchmarks. Consequently, for *mes*, a smaller share of total energy is consumed by the predictor compared to other benchmarks. This results in lower total energy reduction. Also, we have observed that *amm* has the highest number of branches among all benchmarks which could explain why *amm* has the highest energy savings.

Considering the low performance cost associated with SEPAS our savings are more than what could be achieved by frequency and voltage scaling.

5. RELATED WORK

Several previous studies have proposed filters to reduce branch predictor complexity, delay, and destructive aliasing [2,3,4]. Our work differs in that we study the power aspect of exploiting such filters. Chang *et. al* [2], suggested identifying easily predictable branches and inhibiting the pattern history table for these branches to reduce table interference. Eden *et. al* [4] introduced YAGS to reduce aliasing in the pattern history table. While both [2] and [4] focus on the branch predictor buffer we also include the BTB. Jimenez *et. al* [3] suggested using an overriding branch predictor which provides two predictions, one faster and one slower and less accurate consecutively, to reduce delay. Our method takes into account branch confidence to avoid complete access of both predictors. Moreover we also study how this can eliminate updates. *Predictor Probe Detection (PPD)* [1] and *Branch Predictor Prediction (BPP)* [6] are methods to reduce branch predictor energy consumption. 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. SEPAS is different from PPD as it a) eliminates unnecessary predictor updates and b) has a smaller overhead compared to PPD. BPP exploits branch instruction behavior to gate two out of the three sub-predictors during lookup [6]. By comparing SEPAS to BPP we have observed that SEPAS achieves higher energy savings

with lower performance cost. This is mainly due to the fact that SEPAS reduces the number of updates and also the energy consumed by BTB. We estimate confidence using all three counters used in the combined predictor. Branch Predictor Customizing [17] uses software to apply structure resizing and access gating to create a customized branch predictor. We do not use adaptive resizing and depend on dynamically collected branch behavior. SEPAS can be used on top of Predictor Customizing resulting in further savings. Hu *et. al* suggested strategies to reduce leakage energy in branch predictors [19]. We focus on dynamic power.

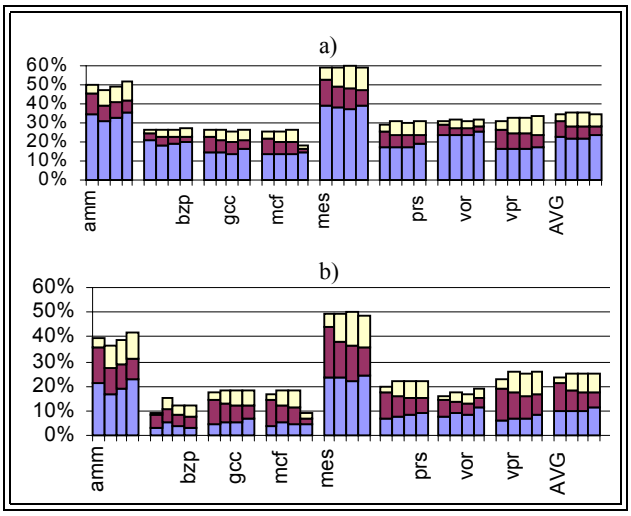

Figure 4: Predictor energy reduction vs. a) BASE and b) UOC machines. Bars from left to right report for 32k, 16k,8k and 4k-entry predictors. We show total savings (entire bar), energy saved by removing updates (lowest part), energy saved by removing lookups in the serial scenario (sum of the middle and the highest part), energy saved by eliminating lookups in the parallel scenario (middle part).

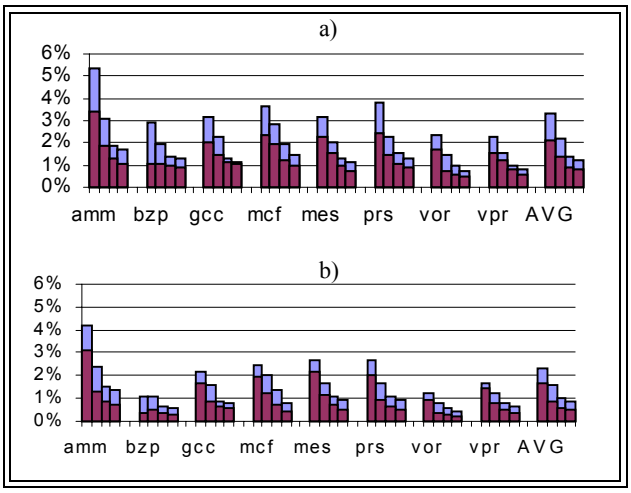

Figure 5: Total energy reduction vs. the BASE machine (entire bar) and the UOC machine (lower bar) for a) serial scenario. b) parallel scenario. Bars from left to right report for 32k, 16k, 8k and 4k-entry branch predictors.

6. CONCLUSION

We presented SEPAS as a highly accurate and energy efficient branch predictor. We demonstrated that it is possible to significantly reduce branch predictor energy consumption by identifying and eliminating the branch predictor accesses that do not contribute to performance. We have shown that when one considers the overall processor energy consumption, SEPAS-enhanced predictors always consume less energy compared to the conventional combined predictor. Because of the considerable energy savings and the very small cost, SEPAS is an attractive power-aware enhancement for modern processors.

References

[1] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M.R. Stan. Power Issues Related to Branch Prediction. *In Proc. of International Symposium on High-Performance Computer Architecture,* Feb. 2002.

[2] P.Chang, M.Evers and Y.N. Patt, Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference. *In Proc. of International Conference on Parallel Architectures and Compilation Techniques.* 1996.

[3] D.A.Jimenez, S.W.Keckler, C. Lin, The Impact of Delay on the Design of Branch Predictors. *In Proc. of International Symposium on Microarchitecture,* Dec. 2000.

[4] A.N.Eden and T.Mudge, The YAGS Branch Prediction Scheme, *In Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, Nov.1998

[5] D. Brooks, V. Tiwari M. Martonosi, Wattch: A Framework for Architectural-Level Power Analysis and Optimizations, *In Proc. of International Symposium on Computer Architecture*, 2000.

[6] A. Baniasadi, A. Moshovos, Branch Predictor prediction a Power-Aware Branch Predictor for High-Performance Processors. *In Proc. of International Conference on Computer Design,* Sep. 2002.

[7] P.-Y. Chang, E. Hao, and Y.N. Patt. Alternative Implementations of Hybrid Branch Predictors. *In Proc. of the 28th Annual International Symposium on Microarchitecture*. Dec. 1995.

[8] S. McFarling. Combining Branch Predictors. Tech. Note TN-36, DECWRL, Jun. 1993.

[9] T.A. Diep, C. Nelson, and J.P. Shen. Performance Evaluation of the PowerPC 620 microarchitecture. *In Proc. of International Symposium on Computer Architecture,* Jun. 1995.

[10] D. Leibholz and R. Razdan. *The Alpha 21264: A 500 MHz Out-of-order Execution Microprocessor.* COMPCON97, 1997.

[11] C.H. Perleberg and A.J. Smith. Branch Target Buffer Design and Optimization. *In IEEE Transactions of Computers,* April 1993.

[12] S. Wilton and N. Jouppi. An Enhanced Access and Cycle Time Model for On-chip Caches. In *WRL Research Report 93/5, DEC Western Research Laboratory,* 1994.

[13] D. C. Burger and T. M. Austin. The SimpleScalar tool set,version 2.0. *Computer Architecture News*, 25(3):13–25, Jun. 1997.

[14] Digital Semiconductor. *DECchip 21064/21064A Alpha AXP Microprocessors: Hardware Reference Manual*, Jun. 1994.

[15] Digital Semiconductor. *Alpha 21164 Microprocessor: Hardware Reference Manual*, Apr. 1995.

[16] A.Seznec, S.Felix, V.Krishnam and Y.Sazeides. Design Tradeoffs for the Alpha EV8 conditional branch predictor. *In Proc. of the 29th International Symposium on Computer Architecture,* May 2002

[17] M.C Huang, D. Chaver, L. Pinuel, M. Prieto, F. Tirado. Customizing The Branch Predictor To Reduce Complexity And Energy Consumption, *In IEEE micro.* Sep. 2003

[18] J.E. Smith. A Study of Branch Prediction Strategies. *In Annual International Symposium on Computer Architecture*, May I981

[19] Z. Hu, P. Juang, K.Skadron, D.Clark and M. Martonosi. Applying Decay Strategies to Branch Predictors for Leakage Energy Savings. *In Proc. of International Conference on Computer Design*, Sep. 2002