Branch Prediction On Demand: an Energy-Efficient Solution*

Daniel Chaver, Luis Piñuel,
Manuel Prieto, Francisco Tirado
Dpto. Arquitectura de Computadores
Universidad Complutense
Madrid, Spain
{dani02, lpinuel, mpmatias, ptirado}@dacya.ucm.es

Michael C. Huang
Dept. of Electrical & Computer Engineering
University of Rochester
Rochester, New York
michael.huang@ece.rochester.edu

ABSTRACT

High-end processors typically incorporate complex branch predictors consisting of many large structures that together consume a notable fraction of total chip power (more than 10% in some cases). Depending on the applications, some of these resources may remain underused for long periods of time. We propose a methodology to reduce the energy consumption of the branch predictor by characterizing prediction demand using profiling and dynamically adjusting predictor resources accordingly. Specifically, we disable components of the hybrid direction predictor and resize the branch target buffer. Detailed simulations show that this approach reduces the energy consumption in the branch predictor by an average of 72% and up to 89% with virtually no impact on prediction accuracy and performance.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Adaptable architectures*

General Terms

Design, Experimentation, Performance

Keywords

Adaptive, Branch Prediction, Profiling

1. INTRODUCTION

Branch prediction is a central piece of technology in exploiting instruction-level parallelism. Modern high-end processors use an array of tables for branch direction and target prediction [13]. These tables are quite large in size (352K bits total for the direction predictor alone in Alpha EV8) and they are accessed every cycle resulting in significant energy consumption - sometimes more than 10% of the total chip power. While high accuracy is essential for high performance *and* energy efficiency, always using the maximum configuration of the branch predictor regardless of the demand is not energy efficient. Consider an array-based floating-point application with very regular control flow: a simple branch predictor or even simply predicting all branches taken could be just as accurate as a complex hybrid predictor. Therefore, we propose to bring the general principle of "on-demand resource allocation" to branch predictor and dynamically adjust the strength of the predictor.

Adaptive branch prediction incurs two types of overhead, reconfiguration overhead and energy waste due to increased misspeculation when using weaker predictor configurations. To reduce such overhead, we adopt a feedback based approach. We partition an application into smaller units called *modules*; characterize their branch prediction demand by profiling; and instrument the application to reconfigure the predictor at runtime. The benefits of this approach are twofold: 1, since characterization and decision making are done off-line, little runtime overhead is incurred; 2, the chosen configuration is highly efficient, because the demand characterization is very accurate, thanks to behavior repetition of modules.

At the circuit level, there are several existing techniques ([1, 15]) to reconfigure cache-like structures to reduce energy consumption. In this paper, we do not attempt to explore all possibilities of existing circuit-level reconfiguration techniques on branch predictor. Rather, to demonstrate the effectiveness of our overall approach to provide on-demand branch prediction capacity, we use two generic, straightforward techniques: a novel access gating technique to reduce ineffectual switchings and table resizing to reduce unnecessarily large capacitative load.

The rest of this paper is organized as follows: Section 2 explains the two different types of predictor adaptations and the profiling methodology, Section 3 discusses the experimental setup , Section 4 shows the evaluation of our proposal, Section 5 discusses related work, and finally Section 6 summarizes.

2. ON-DEMAND BRANCH PREDICTION

2.1 Branch Predictor Reconfiguration

In this paper, we explore two different categories of reconfiguration techniques on the branch predictor: access gating and structure resizing. First, we propose a novel access gating technique for the hybrid direction predictor. Second, we leverage existing resizing techniques for the branch target buffer.

2.1.1 Adaptive Hybrid Predictor through Access Gating

Our baseline direction predictor is a de-aliased hybrid direction predictor: *2Bc-gskew-pskew* [14]. Like in many other hybrid predictors, different prediction tables are accessed and the resulting predictions go through, sometime a series of, majority voting. For certain branches, a vote from a specific prediction table or tables

*This research was supported in part by the Spanish research grant TIC 2002-00750

ISLPED'03, August 25–27, 2003, Seoul, Korea.

can be more accurate than the majority vote. Meta tables are designed to decide which vote sustains.

To reduce energy waste in accessing multiple tables, we propose to disable tables that are not contributing much to improve the prediction accuracy. We decompose the baseline predictor into three components as shown in Figure 1: the *gskew* (**G**) component, the *pskew* (**P**) component, and the *bimodal* (**B**) component. We use two control signals $\overline{GEN}$ and $\overline{PEN}$ to disable (gate accesses to) the gskew and pskew component respectively. The signals can be wired to a special control register and set or cleared using special register load instructions. Once a subset of tables are disabled through this mechanism, they remain inactive until being enabled again. When a certain component is disabled, related meta tables may become irrelevant and can be disabled as well. For example, when $\overline{PEN}$ is asserted, the *pskew* component and metatabe $Meta_2$ are gated and the rest of the tables essentially form a *gskew* predictor.

Notice that we do not gate the bimodal component because it is also used for majority voting in the other two components, and does not consume significant amount of energy. Moreover, having only two control signals simplifies the circuitry.

In theory, it is possible to gate accesses to branch prediction tables for every prediction. The meta tables can be used to allow accesses only to the tables whose prediction will be selected. This saves the energy wasted in accessing the other tables whose predictions will be disregarded anyway. In practice, this does not work well since it sequentialize accesses to the meta table and the prediction tables, thus slows down branch prediction significantly.

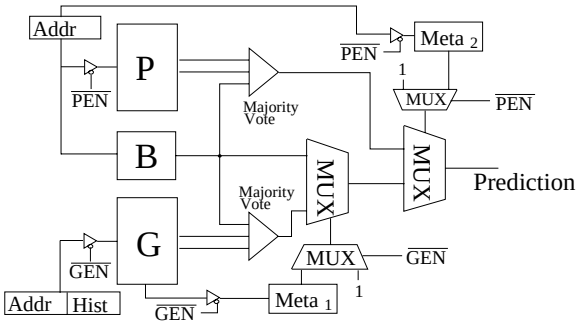

Figure 1: Adaptive hybrid predictor with controls for gating. For brevity, certain structures, such as the local history table for the *pskew* component, are not shown.

2.1.2 Adaptive BTB through Dynamic Resizing

To have accurate branch *direction* predictions is not enough: without the target address, instruction fetch can not proceed after predicted taken branch. Branch target buffer (BTB) helps provide the target address quickly. Increasing the size and associativity of BTBs helps reduce conflict and capacity misses. However, for certain applications, large structures can be a waste. For example, *compress* in the SPEC95 suite has only 46 static branch instructions [13]. Therefore, we propose to resize BTB on the fly. BTB is very similar to a normal data cache in terms of organization and access. There are several schemes in literature discussing how to resize caches and the circuitry to perform the resizing. The main idea of such techniques is to exploit sub-array partitioning, already present for performance reasons, to selectively disable portions of the cache with minimum hardware support. Current proposals for resizable caches differ in the kind of partitioning they are based on: bitline (selective-sets [15]) and wordline (selective-ways [1]). The effectiveness of these two approaches depends on the particular organization of the cache structure. In our case, the baseline BTB is 2-way set associative with 2048 sets (a total of 4096 entries), and segmented into 8 sub-arrays along the bitlines ($N_{dbl} = 8$), according to cache timing model Cacti [12]. Thus, we can scale down the size to 256-set/2-way configuration via selective-sets or to 2048-set/1-way via selective-ways if we use pass-gates to divide wordlines.

BTB resizing is triggered by instructions writing to special control registers. In the context of data caches, resizing entails the need for flushing or other mechanisms to maintain coherence. In our case this is not necessary. The reason is quite simple: the information stored in any prediction table only affects performance and energy, not the correctness of the program execution.

2.2 The Profiling Approach

Adaptive architecture is a promising technology to meet applications' diverse and dynamic resource demand in an efficient manner. Managing adaptation of branch predictor, is a challenging task. Cost and benefit have to be balanced carefully. In particular, we can only switch to a less power-hungry branch predictor configuration if the switch causes minimum degradation in prediction accuracy. This is because the processing of wrong path instructions causes potentially much more energy waste in other parts of the processor than saved in the branch predictor. Another issue is that any extra hardware to keep track of and to predict branch prediction demand will by itself consume extra energy which directly cuts into the savings. Consequently, in this work, we use a profiled-based feedback mechanism to estimate branch prediction demand without incurring runtime overhead.

We statically partition the code into smaller sections, called modules. A module is the smallest unit to apply branch prediction reconfiguration. The reason to tie branch predictor reconfiguration to static code is because intuitively, the code has a large bearing on the branch prediction demand. After all, instruction address is used, exclusively or inclusively, to index almost every table. Also, runtime characteristics of branches (and thus the appropriate predictors for them) do not change much. Certain branches are strongly biased, others correlate with other branches. Finally, our prior research [8] has shown that tying behavior prediction, and thus adaptivity control, to the position of the code is, in general, more effective than time-based prediction and control mechanisms.

The granularity of a module is also important. If a module is too fine-grained, the reconfiguration overhead at runtime will be large, reducing the energy savings. If it is too coarse-grained, it can contain smaller units with different demand on the branch predictor. In this paper, we select subroutines of a program as modules. To reduce switching overhead, and focus only on important subroutines, we use two thresholds in selecting subroutines: average length per invocation, Th_{grain} and total execution time weight, Th_{weight}. In this paper, these two thresholds are set to $1\mu s$ and 5% respectively.

Once the boundaries of the modules are identified, we carry out profiling to determine the demand of prediction resources for each module using a training input. To do this, we use a straightforward approach, running the application multiple times to directly measure the performance and energy effect of different predictor configurations for each module. During this profiling stage, the energy and performance metrics can be obtained via a range of methods: software instrumentation, simulation or using hardware counters. We assume that changing branch predictor configuration for one module does not affect another module. When collecting profiling information for one module, we use the default configuration for the rest of the modules.

Based on the profile, we can select the best configuration for each module such that overall the application achieves most energy

savings without slowing down over a predefined limit. This can be done by using a greedy algorithm solution for the knapsack problem [5]. In this case, the tolerable performance degradation is the capacity of the knapsack, while the total energy savings is the value to maximize. This is similar to the algorithm that we used previously [8]. Once the configuration for each module is chosen, we instrument the binary to carry out the reconfiguration at runtime. This approach incurs very little runtime overhead.

3. EXPERIMENTAL SETUP

We evaluate our proposal on a simulated generic out-of-order processor loosely modeling MIPS R10000 [16]. The baseline branch predictor used is a *2Bc-gskew-pskew* predictor configured with two 4K-entry meta tables, a 4K-entry bimodal table, a gskew component consisting of two 4K-entry global history tables that use 10 bits of global history, and a pskew component consisting of a 1K-entry local history table (8-bit wide) with two 2K-entry PHTs.

Table 1 shows some parameters of the simulated system. We use a MINT-based execution driven simulator [9] that models the contention and occupancy of all resources as the evaluation tool. The simulator incorporates Wattch [4] to evaluate energy consumption. Our simulator fully accounts for all mis-speculation induced overhead.

Processor	
Core: 1 GHz Out-of-order	Branch units: 1
Issue width: 6	Branch penalty: $\geqslant$ 8 cycles
I-window size: 96	RAS entries: 32
Load/store units: 2	BTB entries: 4096
Int,FP units: 5,4	BTB assoc: 2
Pending loads,stores: 16,16	Predictor: *2Bc-gskew-pskew*
Caches	**Bus & Memory**
L1: 32 KB 2-way LRU	FSB freq: 333 MHz
L1 OC,RT: 1,3 ns	FSB width: 128 bits
L1 line size: 32 B	Mem: 2-channel Rambus
L2: 512 KB 8-way pseudo LRU	DRAM bandwidth: 3.2 GB/s
L2 OC,RT: 4,12 ns	Mem RT: 108 ns
L2 line size: 64 B	
I-cache: 32 KB 2-way 32 B line	

Table 1: System configuration. RAS, OC, and RT stand for return address stack, occupancy, and contention-free round trip time from the processor, respectively.

To evaluate our proposal on different types of applications, we select a set of 8 applications that include 5 SPECint applications (181.mcf, 186.crafty, 197.parser, 252.eon, and 256.bzip2), 2 SPECfp applications (171.swim and 301.apsi), and a multimedia application (mp3dec). For the SPECint and SPECfp applications, we reduce the *ref* input dataset to cut down simulation time. Simulation length ranges from hundreds of millions of cycles to over 2.5 billion cycles.

4. EVALUATIONS

In this section, we first evaluate the effect of adapting the size of BTB in Section 4.1, then we evaluate adaptive hybrid predictor in Section 4.2, we briefly discuss combining the two techniques in Section 4.3, and finally we present some discussions in Section 4.4.

4.1 Adaptive BTB

Adaptive BTB exploits the fact that many BTB entries are underused. Figures 2 and 3 show the relationship between the size of BTB and the miss rate for different applications and different modules inside single applications, respectively.

In Figure 2, we see that for some applications, such as *bzip*, BTB miss rate is almost independent of the size in the range shown,

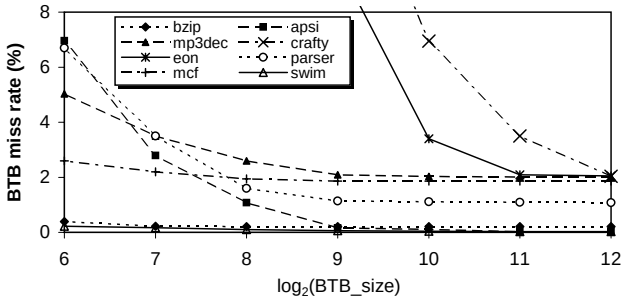

Figure 2: BTB miss rate for different benchmarks using different BTB sizes.

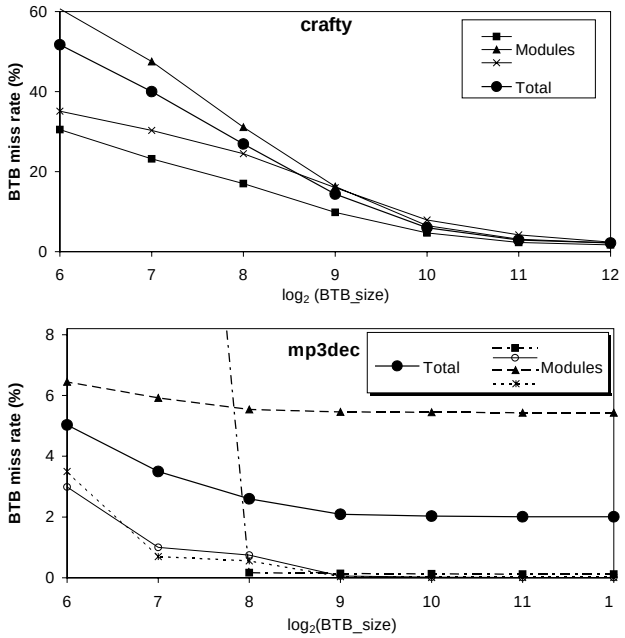

Figure 3: BTB miss rate for different modules and the average of the whole program of two applications (*crafty* and *mp3dec*) with different BTB sizes.

whereas other applications, such as *crafty*, exhibit a significant increase in miss rate for small BTB sizes. In addition, as Figure 3 illustrates, BTB demand for a given application also varies among different modules. This suggests that in some applications many entries of the BTB remain unused for long periods of time. In Figure 4 we can clearly see that while a handful of entries in the BTB are heavily accessed, many others are not.

To gauge an application's BTB size requirement, in the profiling stage, we measure the energy and performance statistics for each module using different BTB sizes. Figure 5 illustrates the results in this stage for *crafty*. For brevity we only show the results of selective-sets scheme on selected modules. The horizontal axis shows the slowdown relative to the program execution time and the vertical axis shows the energy savings relative to total program energy consumption in the processor. We observe the following:

1. For certain modules, moderate BTB resizing can produce relatively significant energy reduction without incurring much slowdown. In fact, BTB resizing can sometimes even improve performance. This may seem counterintuitive, but is possible, since resizing can change BTB entry conflict patterns.
2. Beyond a certain size, further reducing the size of BTB is counterproductive. The reason is straightforward. When the BTB becomes

		apsi		bzip		crafty		eon		mcf		mp3dec		parser		swim	
		train	ref	train	ref	train	ref	train	ref	train	ref	train	ref	train	ref	train	ref
SS	$-\Delta E_{Total}(\%)$	3.55	3.60	8.54	8.62	2.25	2.37	1.60	1.75	7.60	7.72	3.87	3.24	6.54	6.60	1.27	1.17
	$-\Delta E_{BP}(\%)$	57.01	57.83	66.18	65.96	20.13	21.12	43.28	44.80	71.31	72.15	64.60	57.14	57.89	58.41	53.09	49.47
	$\Delta T(\%)$	0.01	0.01	-0.02	0.08	0.50	0.64	0.75	0.97	0.02	0.02	0.38	0.44	0.05	0.08	-0.01	0.00
SW	$-\Delta E_{Total}(\%)$	1.30	1.21	2.16	2.22	0.68	0.55	0.18	0.11	2.22	1.46	1.19	1.14	1.32	1.33	0.50	0.54
	$-\Delta E_{BP}(\%)$	14.99	15.18	15.09	15.18	3.47	3.10	3.97	4.0	14.55	12.04	14.86	15.03	14.17	14.30	13.4	12.8
	$\Delta T(\%)$	0.03	0.03	0.02	0.08	0.09	0.12	0.74	1.08	0.02	0.02	0.31	0.39	0.16	0.21	0.00	0.00

Table 2: Total energy savings and performance degradations obtained using adaptive BTB. SS and SW stand for selective-sets and selective-ways respectively.

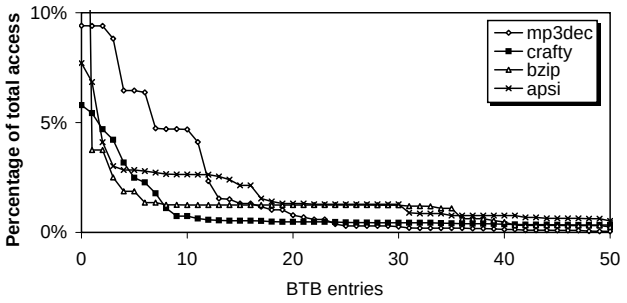

Figure 4: Percentage of total accesses to each entry for a 4096-entry BTB. The entries are sorted by the number of accesses. Only the first 50 entries are shown. For clarity, we only show four applications.

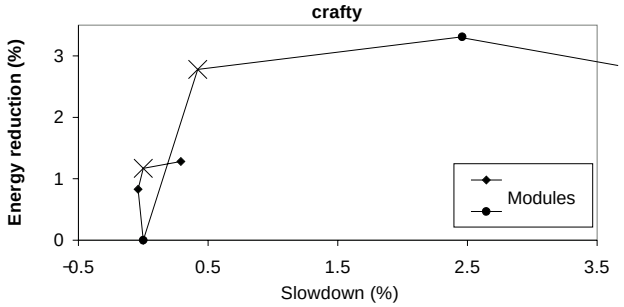

Figure 5: The influence on overall performance and energy consumption by progressive resizing of the BTB sets on different modules in *crafty*. The crosses indicate configurations selected when the tolerable slowdown is set to 0.5%.

too small, slowdown induced energy waste outweighs the energy savings in the BTB.

To demonstrate the effectiveness of BTB resizing, we set up a threshold for tolerable performance degradation: 0.5%. Based on the profile information and this threshold, our off-line decision algorithm chooses the configuration that saves the most energy without going over the threshold for each application, and embed the decision into the application binaries. We then perform two production runs using the training input and the reference input. The results of using this configuration are shown in Table 2.

For each application, the two columns correspond to the two inputs used in the productions runs. Each column contains the relative energy savings in the whole processor ($-\Delta E_{Total}$) and in the branch predictor ($-\Delta E_{BP}$), and the relative increase in total execution time (ΔT).

From the table, we can see that dynamically changing the size of BTB can be profitable: around 20-70% of branch predictor energy and up to 8.6% of total chip energy can be saved with very little performance degradation. We note that the effectiveness of different resizing techniques depends on the particular configuration of the BTB.

4.2 Adaptive Hybrid Predictor

Just as the demand on BTB varies both within and across applications, the demand on the strength of the direction predictor also varies. This is exemplified in Figure 6. In the figure, we show the misprediction rate for a range of predictor configurations: from the most sophisticated full configuration to the simplest bimodal predictor.

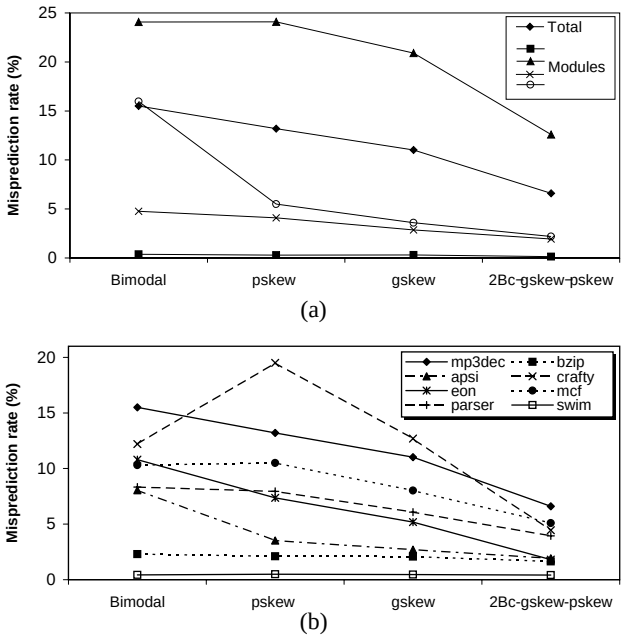

Figure 6: Branch misprediction rate with different branch predictor configurations for each module inside one application *mp3dec* (a), and for all applications (b).

From the figure we can see that, in some cases, certain predictors produce results that are close to the more sophisticated predictors. For example, for *bzip*, the bimodal predictor produces satisfactory results, having a small overall misprediction rate. Experiments show that disabling (gating accesses to) the gskew and pskew components inside the predictor does result in notable energy savings (3% of total processor energy) without noticeable performance degradation for this application. Table 3 summaries performance and energy impact for all applications using adaptive hybrid predictor.

4.3 Combining the Two Adaptations

Finally, we can combine the two techniques to achieve more flexible on-demand branch prediction. Table 4 shows energy savings and performance degradation using both adaptive techniques to save as much energy as possible given a tolerable performance

	apsi		bzip		crafty		eon		mcf		mp3dec		parser		swim	
	train	ref	train	ref	train	ref	train	ref	train	ref	train	ref	train	ref	train	ref
$-\Delta E_{Total}(\%)$	1.84	2.11	2.78	2.95	2.43	2.90	0.92	0.75	3.39	3.29	1.51	1.44	2.20	2.10	0.58	0.56
$-\Delta E_{BP}(\%)$	28.5	32.14	25.32	28.54	21.28	24.95	17.60	17.60	22.67	21.90	24.07	23.45	19.02	18.17	24.22	24.01
$\Delta T(\%)$	0.09	0.12	0.03	0.07	0.32	0.29	-0.09	0.35	-0.02	0.01	0.11	0.19	-0.14	-0.01	0.00	0.00

Table 3: Total energy savings and performance degradations obtained using adaptive hybrid predictor.

degradation limit of 0.5%. In this experiment, we use selective-sets which is more effective for our BTB organization. We see that about 6% on average and as much as 11.5% reduction in processor-wide energy consumption can be achieved.

One key observation from the results presented throughout this section is that using the same profile (based on training input), we obtain very similar results for the two sets of production runs using different inputs. This is in line with our intuition that branch prediction demand largely depends on the code. Also, the results shown in Tables 2, 3, and 4 suggest that the two techniques are largely independent under the tested scenario. Finally we note that the overall energy savings not only depend on the effectiveness of our proposed adaptive system but also depend on the branch prediction demand of the application.

4.4 Discussions

To gain further insight into the validity of our approach, we look at the following issues in more detail.

4.4.1 Invocation Variation

We propose to dynamically adjust branch prediction strength for each module because we hypothesize that branch prediction demand is largely a function of the static code itself. To see if there is a lot of variation of branch prediction demand among the dynamic instances of a module, we show in Figure 7 the standard deviation of BTB hit rate for all applications. We compute the standard deviation of BTB hit rate among all instances of a single module, and then compute the weighted-average of all per-module results. As we can see, in general, the variation is very low. When the configuration is dynamically adjusted, the increase in the standard deviation is also small. Similar results can be shown for the direction prediction. The low standard deviation clearly shows the behavior repetition of modules upon repeated invocations, which in turn suggests that reconfiguring branch predictors based on static code is a viable approach.

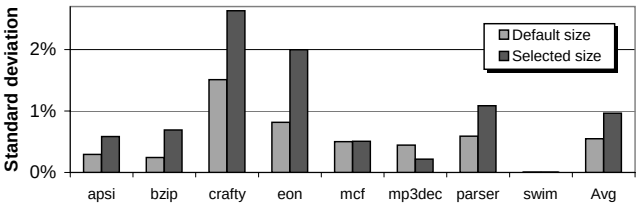

Figure 7: Standard deviation of BTB hit rate for invocations of modules, under the default BTB size and the selected size.

4.4.2 Influence of Different Inputs

Our feedback based approach gauges demand of applications using training inputs and makes decisions based on that. We have already seen some evidence earlier in this section that this is a valid approach: the energy savings and performance impact on two sets of experiments using *train* and *ref* inputs are similar to each other. In Figure 8, we show the difference of branch prediction rate and BTB hit rate between different inputs. We run the same instrumented binaries using two different inputs: the reduced version of *ref* inputs, and the original unchanged *ref* input for 1 billion instructions after the initialization phase. We compare the per-module prediction rates to those obtained using the training input and calculate weighted-average of the absolute differences. As we can see, except for application *mcf* the difference is extremely small. This suggests that different inputs do not change the branch prediction demand, and thus using off-line profiling is a valid way to gauge the demand.

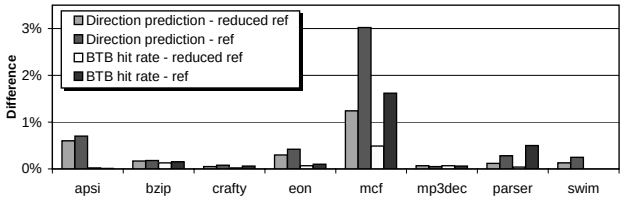

Figure 8: Weighted average of per-module difference in BTB hit rate and branch direction prediction rate between the training input and each of the two production inputs.

The low standard deviation of prediction rates among invocations (Figure 7) and the small difference induced by inputs (Figure 8) support our intuition that branch prediction demand is largely a function of the static code itself.

4.4.3 Interference Between Modules

To reduce the number of profiling experiments, we make a simplifying assumption that the choice of predictor configuration for one module is independent of configurations for other modules. To see if this assumption is reasonable, we compare the BTB hit rate and direction prediction rate of every module under two different cases, where the difference is the predictor configuration for *other* modules. In one case, all other modules keep the default configuration. This is the profile-time scenario. In the other case, all modules choose their selected configuration (to achieve maximum energy savings without passing the 0.5% performance degradation threshold). This is the production-time scenario. To eliminate the influence of different input sets, in both cases, we use the same input – the training input. For each module, we obtain the difference in rates (direction prediction rate and BTB hit rate) between the two cases. In Figure 9, we summarize the data by computing the weighted average of the absolute value of per-module results. As is shown, with the exception of BTB hit rate for application *eon*, the difference between the two cases is negligible, suggesting that the independence assumption is indeed reasonable.

5. RELATED WORK

Some earlier research also looks at energy issues related to branch prediction. Pipeline gating [10] monitors the confidence of predictions for the outstanding branches. When the aggregated confidence is too low, instruction fetch is disabled. The two approaches are orthogonal. While our work tries to reduce energy wasted in the branch predictor itself, their work tries to prevent energy waste in other parts of the processor due to ineffectual branch prediction.

Parikh *et. al.* point out in [11] that modern processors access branch predictors very early in the pipeline stage. This results in

	apsi		bzip		crafty		eon		mcf		mp3dec		parser		swim	
	train	ref	train	ref	train	ref	train	ref	train	ref	train	ref	train	ref	train	ref
$-\Delta E_{Total}(\%)$	5.20	5.33	11.46	11.52	4.83	5.21	2.26	2.28	10.69	10.76	5.06	4.81	8.15	8.21	1.76	1.81
$-\Delta E_{BP}(\%)$	78.91	80.53	89.15	89.48	41.63	43.38	54.07	54.41	74.82	75.52	86.11	82.35	72.41	73.18	74.63	74.98
$\Delta T(\%)$	0.12	0.15	0.00	0.16	0.82	0.96	0.71	1.26	0.03	0.03	0.55	0.69	-0.02	0.11	0.00	0.00

Table 4: Total energy savings and performance degradations obtained using both adaptive BTB (selective-sets) and adaptive hybrid predictor.

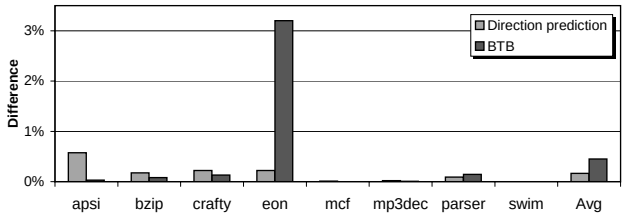

Figure 9: Weighted average of per-module difference between profile-time projected rate (BTB hit rate or branch direction prediction rate) and the production-time measured rate.

many unnecessary accesses. They propose to implement a small, fast structure called *prediction probe detector* to identify chances to avoid or abort branch predictor accesses. While they reduce the number of inconsequential accesses to the predictor, we try to reduce the energy consumption for *every* access, whether useful or not.

Hu *et. al.* let entries that are unused for a long time decay [7]. This reduces leakage energy of branch predictor. This work also exploits the fact that many entries in modern processors' branch predictor tables are underutilized. However, their approach targets leakage energy, while ours targets mainly dynamic energy.

Some other work also proposes to dynamically adjust hardware resources to reduce energy consumption while still meeting application demand. Among other proposals, Albonesi *et. al.* adjust the cache configuration [1, 3], Folegnani and Gonzalez disable empty instruction window entries [6], Bahar and Manne shut down functional units [2]. The concept of these approaches is similar, but the issue of on-demand branch prediction is a bit more tricky. While any adaptation that results in performance degradation runs the risk of increasing energy consumption (due to fixed energy overhead per cycle), adaptation of branch predictor adds an extra source of energy waste: wrong branch predictions introduce useless instructions that will be squashed later. This is not the case for these other adaptations. Moreover, to predict application demand accurately without incurring energy overhead, we adopt a feedback based approach that exploits program behavior repetition at module level, while these related proposals generally use time-based algorithms to control adaptation.

6. SUMMARY

To effectively exploit instruction-level parallelism, high-end processors typically employ sophisticated branch predictors that utilize many large structures. In this paper we have shown that these resources are often underutilized causing unnecessary energy waste. By adapting the size of the branch target buffer and dynamically disabling components of a hybrid predictor, significant amount of energy can be saved with very little performance degradation. We have shown that for a set of eight applications, an average of 71.7% (up to 89.5%) of the energy consumed in the branch predictor or 6.2% (up to 11.5%) processor-wide energy consumption can be saved with negligible performance degradation.

7. REFERENCES

[1] D. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. *Journal of Instruction-Level Parallelism*, 2, 2000.

[2] R. Bahar and S. Manne. Power and Energy Reduction Via Pipeline Balancing. In *International Symposium on Computer Architecture*, pages 218–229, Göteberg, Sweden, June–July 2001.

[3] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In *International Symposium on Microarchitecture*, pages 245–257, Monterey, CA, December 2000.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *International Symposium on Computer Architecture*, pages 83–94, Göteberg, Sweden, June–July 2001.

[5] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw Hill, 1989.

[6] D. Folegnani and A. González. Energy-Effective Issue Logic. In *International Symposium on Computer Architecture*, pages 230–239, Göteberg, Sweden, June–July 2001.

[7] Z. Hu, P. Juang, K. Skadron, D. Clark, and M. Martonosi. Applying Decay Strategies to Branch Predictors for Leakage Energy Savings. In *International Conference on Computer Design*, pages 442–445, Freiburg, Germany, September 2002.

[8] M. Huang, J. Renau, and J. Torrellas. Positional Adaptation of Processors: Application to Energy Reduction. In *International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[9] V. Krishnan and J. Torrellas. A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 286–293, Paris, France, October 1998.

[10] S. Manne, A. Klauser, and D. Grunwald. Pipeline Gating: Speculation Control for Energy Reduction. In *International Symposium on Computer Architecture*, pages 132–141, Barcelona, Spain, June–July 1998.

[11] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan. Power Issues Related to Branch Prediction. In *International Symposium on High-Performance Computer Architecture*, pages 233–244, Cambridge, MA, February 2002.

[12] N. Jouppi S. Wilton. CACTI: an Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.

[13] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor. In *International Symposium on Computer Architecture*, pages 296–306, Anchorage, AK, May 2002.

[14] A. Seznec and P. Michaud. De-aliased Hybrid Branch Predictors. Technical Report No. 3618, Institut National de Recherche en Informatique et en Automatique (INRIA), February 1999.

[15] S. Yang, M. Powell, B. Falsafi, K. Roy, and T. Vijaykumar. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches. In *International Symposium on High-Performance Computer Architecture*, pages 147–157, Nuevo Leone, Mexico, January 2001.

[16] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 6(2):28–40, April 1996.