Power-Aware Branch Prediction: Characterization and Design

Dharmesh Parikh, Kevin Skadron, Yan Zhang, Mircea Stan

Abstract

This paper uses Wattch and the SPEC 2000 integer and floating-point benchmarks to explore the role of branch predictor organization in power/energy/performance tradeoffs for processor design. Even though the direction predictor by itself represents less than 1% of the processor's total power dissipation, prediction accuracy is nevertheless a powerful lever on processor behavior and program execution time. A thorough study of branch predictor organizations shows that, as a general rule, to reduce overall energy consumption in the processor it is worthwhile to spend more *power in the branch predictor if this results in more accurate predictions that improve running time. This not only improves performance, but can also improve the energy-delay product by up to 20%. Three techniques, however, can reduce power dissipation without harming accuracy.* Banking *reduces the portion of the branch predictor that is active at any one time. A new on-chip structure, the* prediction probe detector *(PPD), uses pre-decode bits to entirely eliminate unnecessary predictor and branch target buffer (BTB) accesses. Despite the extra power that must be spent accessing it, the PPD reduces local predictor power and energy dissipation by about 31%, and overall processor power and energy dissipation by 3%. These savings can be further improved by using profiling to annotate branches, identifying those that are highly biased and do not require static prediction. Finally, the paper explores the effectiveness of a previously-proposed technique, pipeline gating, and finds that even with adaptive control based on recent predictor accuracy, pipeline gating yields little or no energy savings.*

Keywords

Low-power design, energy-aware systems, processor architecture, branch prediction, target prediction, power, banking, highly-biased branches, pipeline gating, speculation control.

I. Introduction

This paper explores tradeoffs between power and performance that stem from the choice of branch-predictor organization, and proposes some new techniques that reduce the power dissipation related to prediction without harming performance. Branch prediction has long been an important area of study for microarchitects, because prediction accuracy is a powerful lever over performance. Power-aware computing has also been an important area of study, but until recently was mainly of interest in the domain of mobile, wireless, and embedded devices. Today, however, power dissipation is of interest in even the highest-performance processors. Laptop computers now use high-performance processors but battery life remains a concern, and heat dissipation has become a design obstacle as it is difficult to develop cost-effective packages that can safely dissipate the increasing heat generated by high-performance processors.

While some recent work has explored the power-performance tradeoffs in the processor as a whole and in the memory hierarchy, we are aware of no prior work that looks specifically at issues involving branch prediction except our own [15], [16], [23], [24]. Yet the branch predictor, including the BTB, is the size of a small cache and dissipates a non-trivial amount of power—typically about 7% and as much as 10% of the total processor's power dissipation, according to our simulations—and its accuracy controls the amount of mis-speculated execution and therefore has a substantial impact on energy. For this reason, it is important to develop an understanding of the interactions and tradeoffs between branch predictor organization, processor performance, power spent in the predictor, and power dissipation in the processor as a whole. This paper only examines dynamic power; leakage in branch predictors is discussed in [15] and [16].

Simply trying to reduce the power dissipated in the branch predictor can actually have harmful overall effects. This paper shows that if reducing the power in the predictor comes at the expense of prediction accuracy, hence program performance, this localized reduction may actually *increase* the overall energy dissipated by the processor by making programs run longer. Fortunately, not all the techniques that reduce localized power in the branch predictor suffer such problems. For example, breaking the predictor into banks can reduce power by accessing only one bank per cycle and hence reduces precharge costs, and banking need not have any effect on prediction accuracy. Eliminating unnecessary branch-predictor accesses altogether is an even more direct way to reduce power.

D. Parikh and K. Skadron are with the Dept. of Computer Science, Y. Zhang and M. Stan are with the Dept. of Electrical and Computer Eng., University Of Virginia, 151 Engineer's Way, P.O. Box 400740 Charlottesville, VA, 22904 USA. Phone: (434) 982-2200, Emails: {dharmesh, skadron}@cs.virginia.edu {yz3w, mircea}@virginia.edu.

Overall, there are four main levers for controlling the power related to branch prediction:
1. *Accuracy*: For a given predictor size, better prediction accuracy will not change the power in the predictor, but will make the program run faster, hence reduce *total* energy.
2. *Configuration*: Changing the table size(s) can reduce power within the predictor but may affect accuracy.
3. *Number of Lookups*: Reducing the number of lookups into the predictor is an obvious source of power savings, but it must come with no performance penalty.
4. *Number of Updates*: Reducing the number of predictor updates is another obvious way to reduce power, but is less efficient because mis-speculated computation means that there are many more lookups than updates; because of this aspect we do not further consider updates in this paper.

In this work we also consider modeling issues. Branch predictors try to determine the direction of conditional branches based on the outcome of previous branches. This state must be kept in some sort of on-chip storage. All the tables used to store information—whether caches, branch predictors, or BTBs—consist of essentially the same structure: a memory core of SRAM cells accessed via row and column decoders. Correctly modeling such array structures is important for accurate estimations of performance and power.

A. Contributions

This work uses a modified Wattch 1.02 [3] power/performance simulator to:

- Characterize the power/performance characteristics of different predictor and BTB organizations. As a general rule, to reduce overall energy consumption it is worthwhile to spend *more* power in the branch direction predictor if it permits a more accurate organization that improves running time. This can yield net energy savings as high as 6%.
- Explore the best banked predictor organizations. Banking improves access time and cuts power dissipation at no cost in predictor accuracy. Net savings are less than 1% for the direction predictor, but more can be realized by banking the branch target buffer.
- Propose a new method to reduce lookups, the *prediction probe detector* (PPD). The PPD can use compiler hints and pre-decode bits to recognize when lookups to the BTB and/or direction-predictor can be avoided. Using a PPD cuts power dissipation in the branch predictor by 10–50%, with a net energy savings of 3%.
- Explore specialized treatment for some branches, like those that profiling can identify as having a static bias, in order to minimize the number of branch-predictor lookups, saving as much as an additional 2% of energy.
- Revisit techniques for speculation control via pipeline gating [20]. we show that pipeline gating has little effect for today's more sophisticated and accurate predictors.

Although a wealth of dynamic branch predictors have been proposed, we focus our analysis on a representative sample of the most widely used predictor types: bimodal [29], GAs[1]/gshare [21], [36], PAs [36], and hybrid [21]. We focus mostly on the branch direction predictor that predicts directions of conditional branches. Please note that data for the "predictor power" includes power for both the direction predictor and the BTB, as techniques like the PPD affect both. For a typical branch predictor configuration—a 32 K-entry GAs predictor, for example—80.5% of the power dissipation comes from the BTB, 14% from the direction predictor, and 5.5% from the return-address stack. Despite the small amount of power dissipated in the direction predictor—less than 0.6 W—this paper shows that the choice of direction predictor has a much larger impact on power dissipation and energy efficiency in the rest of the processor than might be suggested by the small share of power dissipated within the predictor. For SPECint, the best direction predictor we examined reduces the processor's overall energy consumption by 10% compared to the worst predictor and, due to improved prediction accuracy, improves the processor's energy-delay product by over 20%.

Our goal is to understand how the different branch-prediction design options interact at both the performance and power level, the different tradeoffs that are available, and how these design options affect the overall processor's power/performance characteristics. Our hope is that these results will provide a road-map to help researchers and designers better find branch predictor organizations that meet various power and performance design goals.

[1] GAg, GAs, PAg, PAs are a common naming scheme for two-level predictors [36]. The first letter indicates global ("G") vs. local ("P"—per-address) history, the "A" indicates the use of two-bit counters, and the last letter indicates whether the PHT is logically a single table ("g") or in sets ("s").

B. Related Work

Some prior research has characterized power in other parts of the processor. This paper extends our earlier work [23] with more accurate power modeling, a more accurate and thorough analysis of the PPD, data for integer and floating-point programs instead of just integer programs, a study of BTB configuration, and more analysis of the effects of pipeline gating. Pipeline gating was presented by Manne *et al.* [20] as an efficient technique to prevent mis-speculated instructions from entering the pipeline and wasting energy while imposing only a negligible performance loss. Albonesi [1] explored disabling a subset of the ways in a set associative cache during periods of modest cache activity to reduce cache energy dissipation. By exploring the performance and energy implications he showed that a small performance degradation can produce significant reduction in cache energy. Ghose and Kamble [12] looked at sub-banking and other organizational techniques for reducing energy in the cache. Zhu and Zhang [37] describe a low-power associative cache mode that performs tag match and data access sequentially, and described a way to predict when to use the sequential and parallel modes. Our PPD performs a somewhat analogous predictive function for branch prediction, although the predictor is not associative and the PPD controls whether the predictor is used at all. Kin *et al.* [19] and Tang *et al.* [32] described filter caches and predictive caches, which utilized a small "L0" cache to reduce accesses and energy in subsequent levels. Our PPD performs a somewhat analogous filtering function, although it is not itself a branch predictor. Ghiasi *et al.* [11] reasoned that reducing power at the expense of performance is not always correct. They proposed that software, including a combination of the operating system and user applications, should use a performance mechanism to indicate a desired level of performance and allow the micro-architecture to choose then between the extant alternative methods that achieve the specified performance while reducing power. Finally, Bahar and Manne [2] proposed an architectural energy-efficiency technique that dynamically adpats to instruction-throughput requirements. The technique, called *pipeline balancing*, dynamically tunes the resources of a general purpose processor to the needs of the application by monitoring performance within each application, but their work did not directly treat branch prediction.

The rest of this paper is organized as follows. The next section describes our simulation technique and our extensions to the Wattch power model. Section III then explores tradeoffs between predictor accuracy and power/energy characteristics, and Section IV explores changes to the branch predictor that save energy without affecting performance. Finally, Section V summarizes the paper.

II. Simulation Technique and Metrics

Before delving into power/performance tradeoffs, we describe our simulation technique, our benchmarks, the different types of branch predictors we studied and the ways in which we improved Wattch's power model for branch prediction.

A. Simulator

For the baseline simulation we use a slightly modified version of the Wattch [3] version 1.02 power-performance simulator. Wattch augments the SimpleScalar [4] cycle-accurate simulator (*sim-outorder*) with cycle-by-cycle tracking of power dissipation by estimating unit capacitances and activity factors. Because most processors today have pipelines longer than five stages to account for renaming and en-queuing costs like those in the Alpha 21264 [18], Wattch simulations extend the pipeline by adding three additional stages between decode and issue. In addition to adding these extra stages to sim-outorder's timing model, we have made minor extensions to Wattch and sim-outorder by modeling speculative update and repair for branch history and for the return-address stack [26], [27], and by changing the fetch engine to recognize cache-line boundaries. A more important change to the fetch engine is that we now charge apredictor and BTB lookup for each *cycle* in which the fetch engine is active. This accounts for the fact that instructions are fetched in blocks, and that—in order to make a prediction by the end of the fetch stage—the branch predictor structures must be accessed before any information is available about the contents of the fetched instructions. This is true because the instruction cache, direction predictor, and BTB typically must be all accessed in parallel. Thus, even if the I-cache contains pre-decode bits, their contents are typically not available in time. This is the most straightforward fetch-engine arrangement; a variety of other more sophisticated arrangements are possible, some of which are explored in Section IV.

Unless stated otherwise, this paper uses the baseline configuration shown in Table I, which resembles as much as possible the configuration of an Alpha 21264 [18]. The most important difference for this paper is that in the 21264 there is no separate BTB, because the I-cache has an integrated next-line predictor [6]. As most processors

TABLE I

SIMULATED PROCESSOR CONFIGURATION, WHICH MATCHES AN ALPHA 21264 AS MUCH AS POSSIBLE.

Processor Core	
Instruction Window	RUU=80; LSQ=40
Issue width	6 instructions per cycle: 4 integer, 2 FP
Pipeline length	8 cycles
Fetch buffer	8 entries
Functional Units	4 Int ALU, 1 Int mult/div, 2 FP ALU, 1 FP mult/div, 2 mem ports
Memory Hierarchy	
L1 D-cache Size	64KB, 2-way, 32B blocks, write-back
L1 I-cache Size	64KB, 2-way, 32B blocks, write-back
L1 latency	1 cycle
L2	Unified, 2MB,4-way LRU 32B blocks, 11-cycle latency, write-back
Memory latency	100 cycles
TLB Size	128-entry, fully assoc., 30-cycle miss
Branch Predictor	
Branch target buffer	2048-entry, 2-way
Return-address-stack	32-entry

currently do use a separate BTB, our work models a separate, 2-way associative, 2 K-entry BTB that is accessed in parallel with the I-cache and direction predictor.

To keep in line with contemporary processors, we use the process parameters for a 0.18μm process at V_{dd} 2.0V and 1.2 GHz. All the results use Wattch's non-ideal aggressive clock-gating style ("cc3"). In this clock-gating model, power is scaled linearly with port or unit usage, and inactive units still dissipate 10% of the maximum power.

B. Benchmarks

As benchmarks we evaluate the programs from the SPECcpu2000 [31] suite. Basic branch characteristics are presented in Table II. Branch mispredictions also induce other negative consequences, like cache misses due to mis-speculated instructions, but we do not attempt to quantify or solve those second-order effects here. All benchmarks are compiled using the Compaq Alpha compiler with the SPEC *peak* settings, and the statically-linked binaries include all library code. Unless stated otherwise, we always use the provided reference inputs. We use Alpha EIO traces and the EIO trace facility provided by SimpleScalar for all our experiments. This ensures reproducible results for each benchmark across multiple simulations. *252.eon* and *181.mcf*, from SPECint2000, and *178.galgel* and *200.sixtrack*, from SPECfp2000, were not simulated due to problems with our EIO traces. All benchmarks were fast-forwarded past the first 2 billion instructions and then full-detail simulation was performed for 200 million instructions.

C. Metrics

The following metrics are used to evaluate and understand the results.

- *Average Instantaneous Power*: Power dissipation averaged on a per-cycle basis.
- *Energy*: The product of average power and total execution time. This metric translates directly to battery life.
- *Energy-Delay Product*: The product of energy and execution time. This metric [13] captures the tradeoff between energy efficiency and performance.
- *Performance*: We use the common metric of instructions per cycle (IPC).

D. Branch Predictors Studied

The bimodal predictor [29] consists of a simple *pattern history table* (PHT) of saturating two-bit counters, indexed by branch PC. This means that all dynamic executions of a particular branch site (a "static" branch) will map to the same PHT entry. This paper models 128-entry through 16 K-entry bimodal predictors. The 128-entry predictor is the same size as that in the Motorola ColdFire v4 [33]; 4 K-entry is the same size as that in the Alpha 21064 [9] and is at the point of diminishing returns for bimodal predictors, although the 21164 used an 8 K-entry

TABLE II

BENCHMARK SUMMARY.

	Dynamic Unconditional Branch Frequency (% of instructions)	Dynamic Conditional Branch Frequency (% of instructions)	Prediction Rate w/ Bimod 16K	Prediction Rate w/ Gshare 16K
gzip	3.05%	6.73%	85.87%	91.06%
vpr	2.66%	8.41%	84.96%	86.27%
gcc	0.77%	4.29%	92.03%	93.51%
crafty	2.79%	8.34%	85.88%	92.01%
parser	4.78%	10.64%	85.37%	91.92%
perlbmk	4.36%	9.64%	88.10%	91.25%
gap	1.41%	5.41%	86.59%	94.18%
vortex	5.73%	10.22%	96.58%	96.66%
bzip2	1.69%	11.41%	91.81%	92.22%
twolf	1.95%	10.23%	83.20%	86.99%
wupwise	2.02%	7.87%	90.38%	96.62%
swim	0.00%	1.29%	99.31%	99.68%
mgrid	0.00%	0.28%	94.62%	97.00%
applu	0.01%	0.42%	88.71%	98.95%
mesa	2.91%	5.83%	90.68%	93.31%
art	0.39%	10.91%	92.95%	96.39%
equake	6.51%	10.66%	96.98%	98.16%
facerec	1.03%	2.45%	97.58%	98.70%
ammp	2.69%	19.51%	97.67%	98.31%
lucas	0.00%	0.74%	99.98%	99.98%
fma3d	4.25%	13.09%	92.00%	92.91%
apsi	0.51%	2.12%	95.24%	98.78%

predictor [10]. The gshare predictor [21], shown in Figure 1a, is a variation on the two-level GAg/GAs global-history predictor [22], [36]. The advantage of global history is that it can detect and predict sequences of correlated branches. In a conventional global-history predictor (GAs), a history (the global branch history register or GBHR) of the outcomes of the h most recent branches is concatenated with some bits of the branch PC to index the PHT. Combining history and address bits provides some degree of anti-aliasing to prevent destructive conflicts in the PHT. In gshare, the history and the branch address are XOR'd. This permits the use of a longer history string, since the two strings do not need to be concatenated to fit into the desired index width. This paper models a 4 K-entry GAs predictor with 5 bits of history [28]; a 16 K-entry gshare predictor in which 12 bits of history are XOR'd with 14 bits of branch address (this is the configuration that appears in the Sun UltraSPARC-III [30], the shorter global history string giving good anti-aliasing); a 32 K-entry gshare predictor, also with 12 bits of history; and a 32 K-entry GAs predictor with 8 bits of history [28].

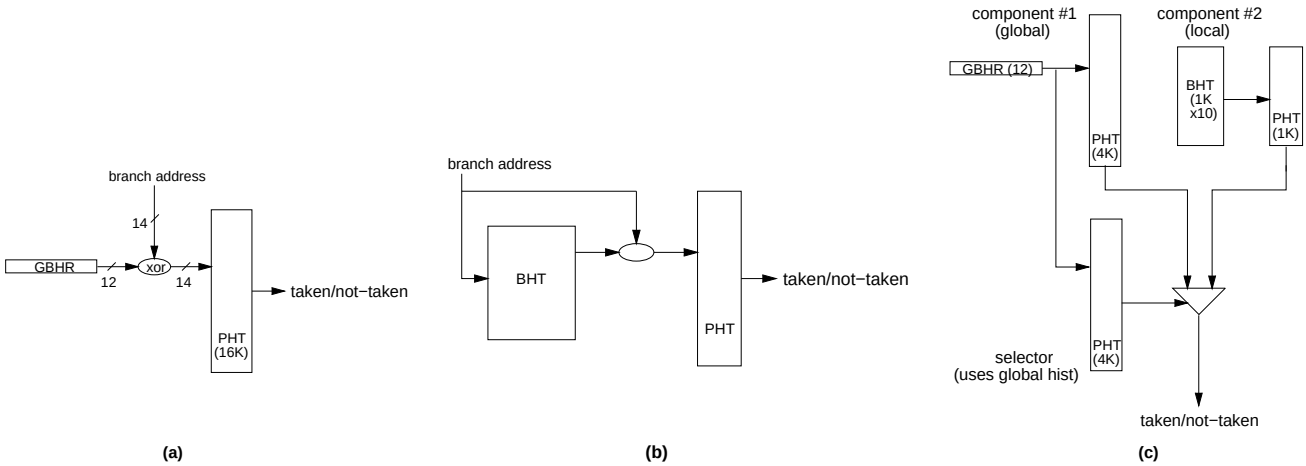


Fig. 1. (a) Gshare global-history branch predictor like that in the Sun UltraSPARC-III. (b) PAs local-history predictor. (c) Hybrid predictor like that in the Alpha 21264.

Instead of using global history, a two-level predictor can track history on a per-branch basis; Figure 1b shows a PAs predictor. In this case, the first-level structure is a table of per-branch history registers—the *branch history table* or BHT—rather than a single GBHR shared by all branches. The history pattern is then combined with some number of bits from the branch PC to form the index into the PHT. Local-history prediction cannot detect correlation, because—except for unintentional aliasing—each branch maps to a different entry in the BHT. Local history, however, is effective at exposing patterns in the behavior of individual branches. The Intel P6 architecture is widely believed to use a local-history predictor, although its exact configuration is unknown. This paper examines two PAs configurations: the first one has a 1 K-entry, 4-bit wide BHT and a 2 K-entry PHT; the second one has a 4 K-entry, 8-bit wide BHT and a 16 K-entry PHT. Both are based on the configurations suggested by Skadron *et al.* [28].

Because most programs have some branches that perform better with global history and others that perform better with local history, a hybrid predictor [8], [21], combines the two as shown in Figure 1c. It operates two independent branch predictor components in parallel and uses a third predictor—the *selector* or *chooser*—to learn for each branch which of the components is more accurate and chooses its prediction. Using a local-history predictor and a global-history predictor as the components is particularly effective, because this accommodates branches regardless of whether they prefer local or global history. This paper considers four hybrid predictors, the first one being similar to the one in the Alpha 21264 [18], the other three being based on configurations found to perform well by Skadron *et al.* [28]:

1. *Hybrid_1*: a hybrid predictor with a 4K-entry selector that only uses 12 bits of global history to index its PHT; a global-history component predictor of the same configuration; and a local history predictor with a 1 K-entry, 10-bit wide BHT and a 1 K-entry PHT. This configuration appears in the Alpha 21264 [18] and is depicted in Figure 1c. It contains 28 Kbits of information.
2. *Hybrid_2*: a hybrid predictor with a 1 K-entry selector that uses 3 bits of global history to index its PHT; a global-history component predictor of 2K entries that uses 4 bits of global history; and a local history predictor with a 512 entry, 2-bit wide BHT and a 512 entry PHT. It contains 8 Kbits.
3. *Hybrid_3*: a hybrid predictor with an 8 K-entry selector that uses 10 bits of global history to index its PHT; a global-history component predictor of 16K entries that uses 7 bits of global history; and a local history predictor with a 1 K-entry, 8-bit wide BHT and a 4 K-entry PHT. It contains 64 Kbits.
4. *Hybrid_4*: a hybrid predictor with an 8 K-entry selector that uses 6 bits of global history to index its PHT; a global-history component predictor of 16K entries that uses 7 bits of global history; and a local history predictor with a 1 K-entry, 8-bit wide BHT and a 4 K-entry PHT. It also contains 64 Kbits.

A brief summary of all the branch predictors studied is given in Table III.

E. Modeling Power in the Branch Predictor

All the tables used to store information—whether caches, branch predictors, or BTBs—typically consist of essentially the same structure: a memory core of SRAM cells accessed via row and column decoders. Each SRAM cell stores a bit and is placed in a matrix-like structure where a row represents the wordline and a column the bitline, as shown in Figure 2. The address contains the information that is necessary to access the desired bits in this structure. The address feeds the row and column decoders that are responsible for driving the correct wordline and selecting the correct bitlines, respectively. As a result, the selected bit values are pulled from the bitlines to the sense amps and into the bus (through the pass-gate transistors of the column multiplexor).

PHT and/or BHT tables are used in different ways according to the actual branch predictor that is implemented. When considering different branch predictors for low power or energy-delay optimization, it is necessary to be able to model their power consumption.

Although one may be accustomed to thinking of array structures according to their logical dimensions, in reality their physical implementation may be quite different, as it is optimized for delay and energy considerations. The physical dimensions are typically chosen such that the layout is as square as possible and the bitline and wordline lengths are minimized. The data that is logically referred to is physically accessed by feeding the column and row decoders with the appropriate index bits. The word line is activated using the row decoder, and the bit lines are then selected by the column decoder that controls a multiplexor. The column decoder is not in the critical path, because it can work in parallel with the driving of the wordline.

In Wattch, the row decoder is implemented with a predecoder made of 3-input NAND gates followed by NOR gates. This implementation is rather basic, yet is commonly used in practice and is the one adopted for our model.

TABLE III

SUMMARY OF BRANCH PREDICTORS STUDIED.

Predictor	No. of branch bits	No. of global history bits	No. of local history bits	Methodology of combining	Size of BHT (bits)	Size of PHT (bits)	Total Size (bits)
Bim_128	7	X	X	X	X	256	256
Bim_4K	12	X	X	X	X	8K	8K
Bim_8K	13	X	X	X	X	16K	16K
Bim_16K	14	X	X	X	X	32K	32K
GAs_1_4K_5	7	5	X	Concat	X	8K	8K
GAs_1_32K_8	7	8	X	Concat	X	64K	64K
Gsh_1_16K_12	14	12	X	XOR	X	32K	32K
Gsh_1_32K_12	15	12	X	XOR	X	64K	64K
Hybrid_1							28K
Global	X	12	X	X	X	8K	8K
Local	X	X	10	X	10K	2K	12K
Selector	X	12	X	X	X	8K	8K
Hybrid_2							8K
Global	7	4	X	Concat	X	4K	4K
Local	7	X	2	Concat	1K	1K	2K
Selector	7	3	X	Concat	X	2K	2K
Hybrid_3							64K
Global	7	7	X	Concat	X	32K	32K
Local	4	X	8	Concat	8K	8K	16K
Selector	3	10	X	Concat	X	16K	16K
Hybrid_4							64K
Global	7	7	X	Concat	X	32K	32K
Local	4	X	8	Concat	8K	8K	16K
Selector	7	6	X	Concat	X	16K	16K
PAs_1K_2K_4	7	X	4	Concat	4K	4K	8K
PAs_4K_16K_8	6	X	8	Concat	32K	32K	64K

Wattch 1.02 does not model the column decoder, so we added it for all the array structures. Modeling the column decoder adds an offset of about 10% to the predictor power values calculated by the original Wattch model, and this offset increases slightly as predictor sizes get larger (see [24] for details).

A fairly accurate and complete model is important because one approach might yield a wider structure than other approaches. For example, banking causes some overhead in multiplexing and disabling the banks, and without a correct implementation of the column decoders, comparators and drivers, one would miss that part of the power dissipation, biasing the analysis. Indeed, we did find that the behavior and order of several benchmarks change when we added the column decoders to the model, and that the Hybrid_2 predictor gets much less benefit from banking than would be found if the column decoders were not included (see [24]).

In our model, each column of the PHT is actually made of two bits and the column decoder driver drives two sets of pass-gate transistors. The same implementation is used for the BHT, with the difference that bits are taken in groups of h (h being the history length) as specified by the specific configuration of the branch predictor. The power model of the branch target buffer (BTB) includes components such as the comparator, the tag bit drivers and the multiplexor drivers. The size and associativity of the BTB are parameters of the model and all capacity contributions are included.

Finally, it is important to note that the logical dimensions of the predictor structures must fit into a physical organization that is as square as possible, so Wattch must find the best "squarified" organization. In Wattch this "squarification" is done automatically so that the physical number of rows in the branch prediction structure becomes as equal as possible to the number of columns. When these two dimensions cannot be equal, however, it may be that one organization (e.g. more rows than columns) has slightly better power characteristics while another (e.g. more columns than rows) has better delay characteristics. This happens, for example, with the 8 K-entry and 32 K-entry predictors (see [24] for more details).

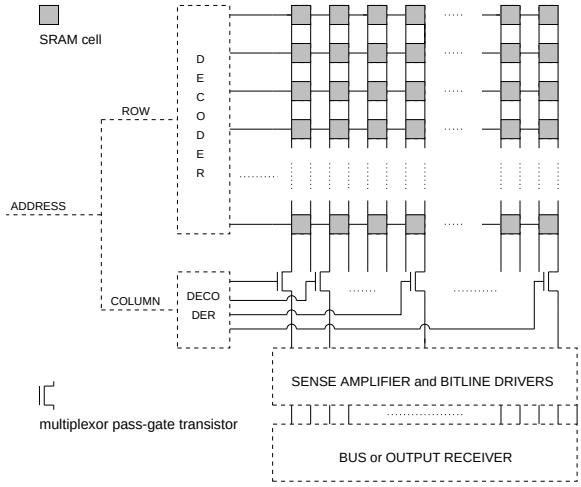


Fig. 2. SRAM Array

III. Performance-Power Tradeoffs Related to Branch Prediction

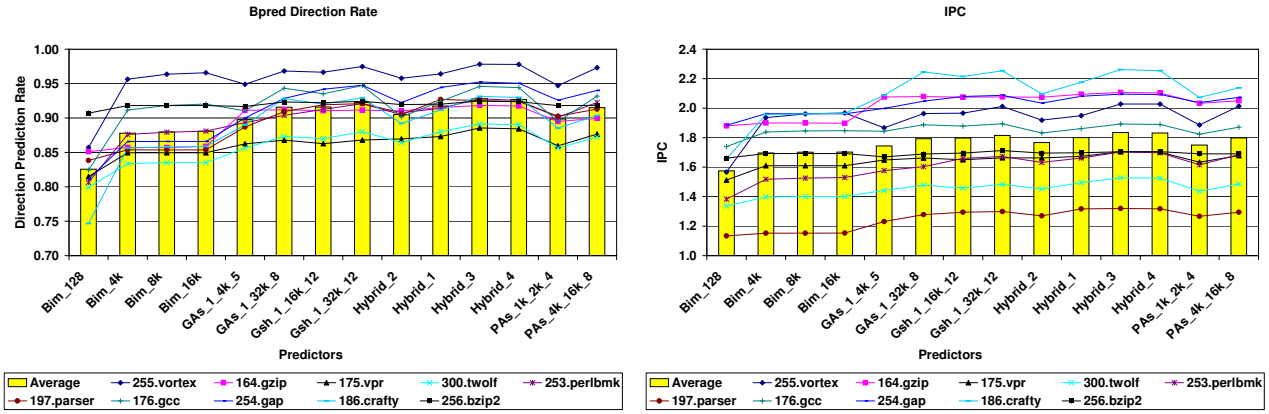
We now examine the interaction between predictor configuration, performance, and power/energy characteristics for integer and floating-point benchmarks. Most of this section is dedicated to the impact of direction-predictor design, but we also briefly explore the role of BTB design. In our discussion below, the term "average", wherever it occurs, means the arithmetic mean for that metric across all the benchmarks simulated.

A. Base Simulations for Integer Benchmarks

Figure 3a presents the average branch predictor direction accuracy for integer benchmarks, and Figure 3b presents the corresponding IPC. For each predictor type (bimodal, GAs, gshare, hybrid, and PAs), the predictors are arranged in order of increasing size, and the arithmetic mean is superimposed on each graph as a set of bars. The trends are exactly as we would expect: larger predictors get better accuracy and higher IPC, but eventually diminishing returns set in. This is most clear for the bimodal predictor, for which there is little benefit to sizes above 4K entries. For the global-history predictors, diminishing returns set in at around 16K–32K entries. Among different organizations, gshare slightly outperforms GAs, and hybrid predictors are the most effective at a given size. For example, compare the 32 K-entry global predictors, hybrid_3 and 4, and the second PAs configuration: they all have 64 Kbits total area, but the hybrid configurations are slightly better on average and also for almost every benchmark.

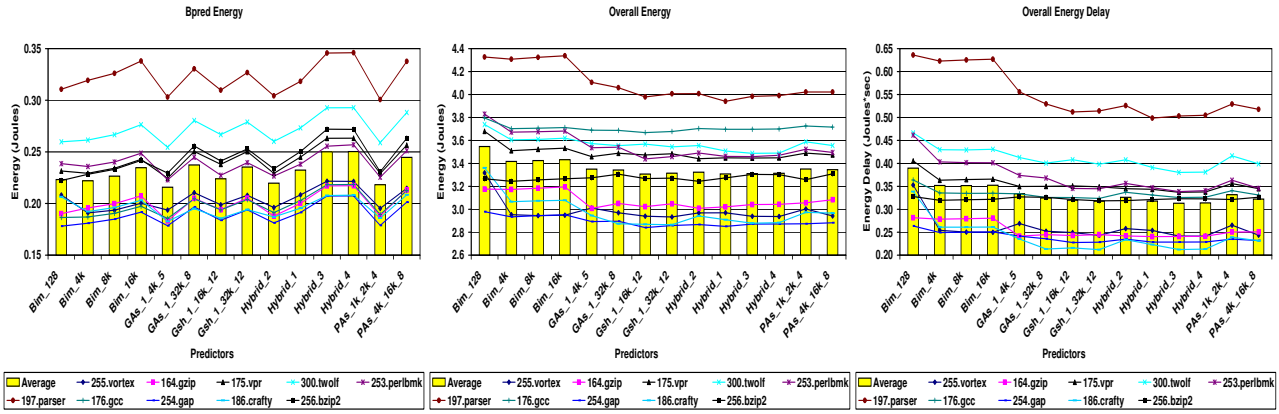
Figure 4 gives the energy and energy-delay characteristics. Together, Figure 4a and Figure 4b show that processor-wide energy is primarily a function of predictor *accuracy* and not of the energy expended in the predictor. For example, although the energy spent locally in hybrid_3 and hybrid_4 is larger than for a gshare predictor of 16 K-entry, the chip-wide energy is almost the same. And the small or otherwise poor predictors, although consuming less energy locally in the predictor, actually cause substantially more energy consumption chip-wide. The hybrid_4 predictor, for example, consumes about 6% less chip-wide energy than bimodal-128 despite consuming 9% more energy locally in the predictor. This suggests that "low-power" processors (which despite their name are often more interested in long battery life) might be better off using *large* and aggressive predictors if the die budget and cooling budget can afford it. The best predictor from an energy standpoint is actually hybrid_1, the 21264's predictor, which attains a slightly lower IPC but makes up for the longer running time with a predictor of less than half the size. Although hybrid_1 is superior from an energy standpoint, it shows less advantage on energy-delay; the 64 Kbit hybrid predictors (hybrid_3 and hybrid_4) seem to offer the best balance of energy and performance characteristics.

The power data in Figure 5 shows that power dissipation in the predictor itself is mostly a function of predictor size, and that unlike energy, power in the processor as a whole tracks predictor *size*, not predictor *accuracy*. This is because power is an instantaneous measure and hence is largely unaffected by program running time. Yet average activity outside the branch predictor is not independent of predictor accuracy. This means that even though predictor size becomes the primary lever on overall power, the changes in chip-wide power dissipation (Figure 5b) are larger than the small changes of less than 1W that are observed locally in the predictor. Figure 5 also shows that if power dissipation or power density is more important than energy, GAs_1_4K, gshare_16K, or one of the smaller hybrid



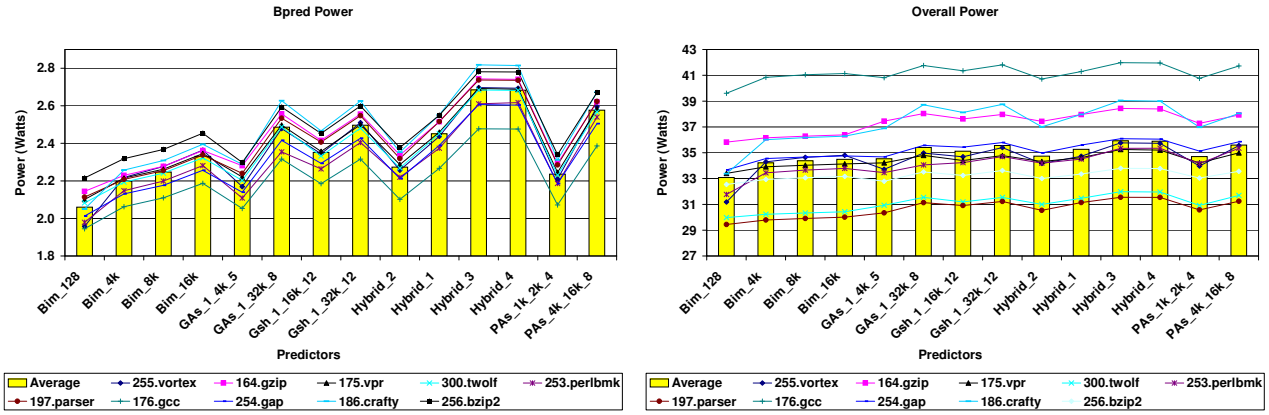
(a) (b)

Fig. 3. (a) Direction-prediction accuracy and (b) IPC for SPECint2000 for various predictor organizations. For each predictor type, the predictors are arranged in order of increasing size along the X-axis. The arithmetic mean is shown by the bars in each of the graphs.



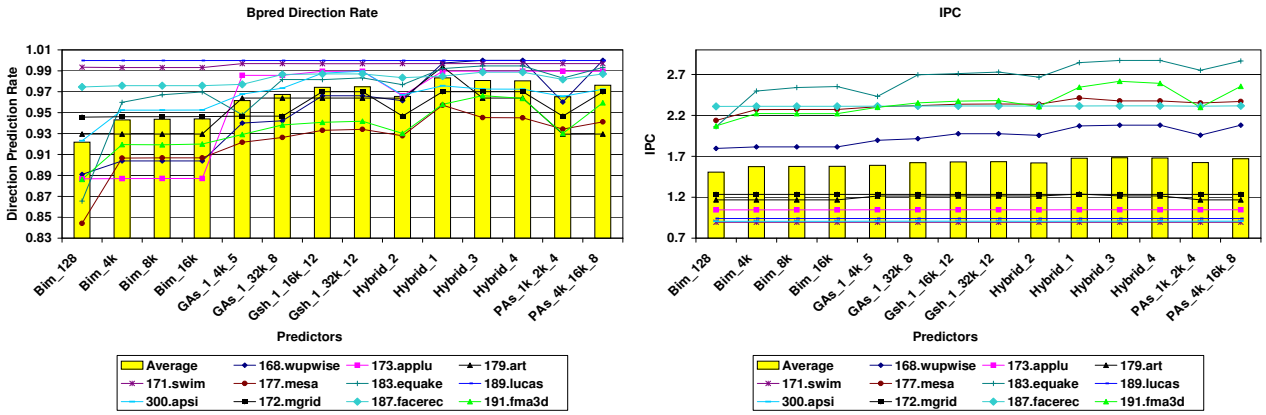
(a) (b) (c)

Fig. 4. Energy in (a) the branch predictor and (b) the entire processor, and (c) energy-delay for the entire processor for SPECint2000.



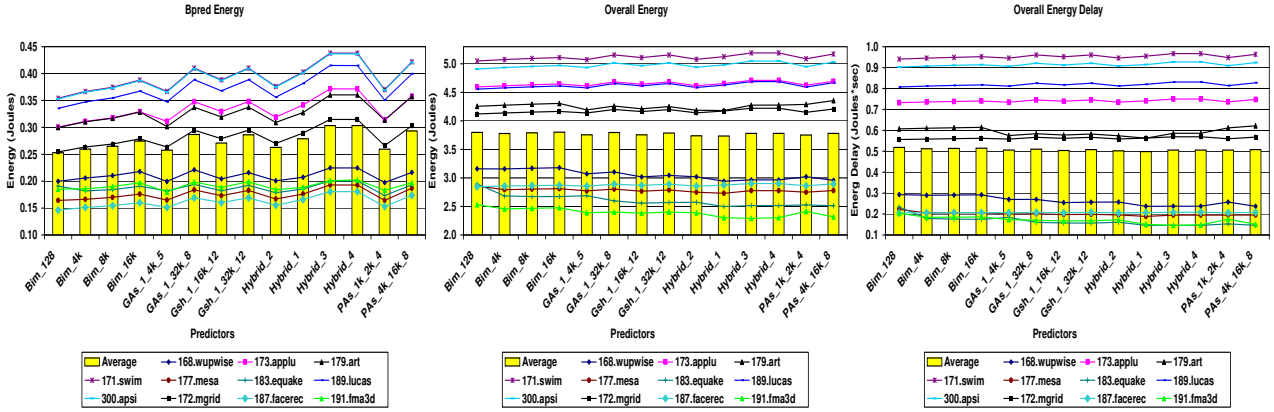
(a) (b)

Fig. 5. Power dissipation in (a) the branch predictor and (b) the entire processor for SPECint2000.



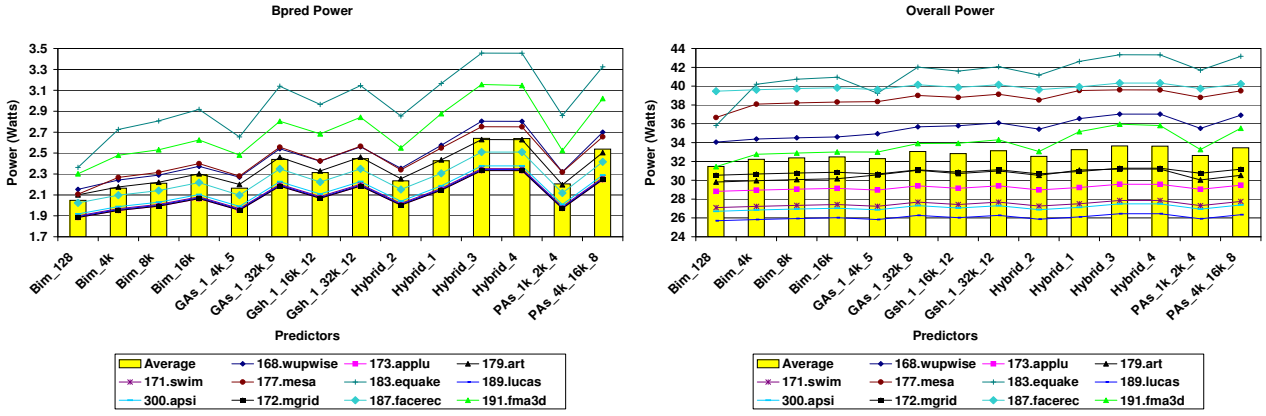
(a) (b)

Fig. 6. (a) Direction-prediction accuracy and (b) IPC for SPECfp2000.



(a) (b) (c)

Fig. 7. Energy in (a) the branch predictor and (b) the entire processor; (c) energy-delay for the entire processor for SPECfp2000.



(a) (b)

Fig. 8. Power dissipated in (a) the predictor and (b) the entire processor for SPECfp2000.

predictors is the best balance of power and performance.

An important point to note is that the branch-predictor power data reported here differ from that in our earlier work [23] because our previous work mistakenly double-counted the branch predictor power. Qualitative trends, however, remain the same.

Finally, Figures 3, 4 and 5 also show data for individual benchmarks. It is clear that *gap* and *vortex*, with their high prediction rates, have high IPCs and correspondingly low overall energy and energy-delay despite higher predictor and total instantaneous power. *Crafty* and *gzip* do not have the highest prediction rates but still have high IPCs, and hence exhibit similarly low energy and energy-delay characteristics. *Parser* and *twolf*, at the other extreme, have the exact opposite properties. This merely reinforces the point that almost always there is no rise (and in fact usually a decrease) in total energy if one uses larger branch predictors to obtain faster performance!

B. Base Simulations for Floating Point Benchmarks

Figures 6–8 repeat these experiments for SPECfp2000. The trends are almost the same, with two important differences. First, because floating-point programs tend to be dominated by loops and because branch frequencies are lower, these programs are less sensitive to branch predictor organization. Second, because they are less sensitive to predictor organization, the energy curves for the processor as a whole are almost flat. Indeed, the mean across the benchmarks is almost entirely flat. This is because the performance and hence energy gains from larger predictors are much smaller and are approximately offset by the higher energy spent in the predictors.

C. Tradeoffs for Branch-Target Buffers

So far, our results have focused on the impact of organization for the branch direction predictor. In fact, the bulk of the energy during branch prediction comes from the BTB (about $\frac{7}{8}$). We have explored the impact of BTB size and associativity on power-performance tradeoffs, and as expected, found that its size has a major impact on power and energy. The smallest configuration we explored—a 128-entry, 1-way BTB—dissipates 75% less power than the largest configuration—a 2048-entry, 2-way BTB. Average power and energy dissipation for various configurations are presented in Figure 9.

The correct choice of BTB configuration that balances power and performance considerations is difficult to make with the benchmarks available to academia: most such benchmarks have small branch footprints that barely stress even a 128-entry BTB, while many commercial applications have footprints that stress even a large, 2048-entry BTB. For these reasons, we found that IPC changed by only 2% on average when moving from a 128-entry to a 2048-entry BTB, and the largest IPC change among our integer benchmarks was only 5.7% for *crafty*. These small performance effects mean that the power, energy, and energy-delay trends we observed are similar and simply follow the configuration.

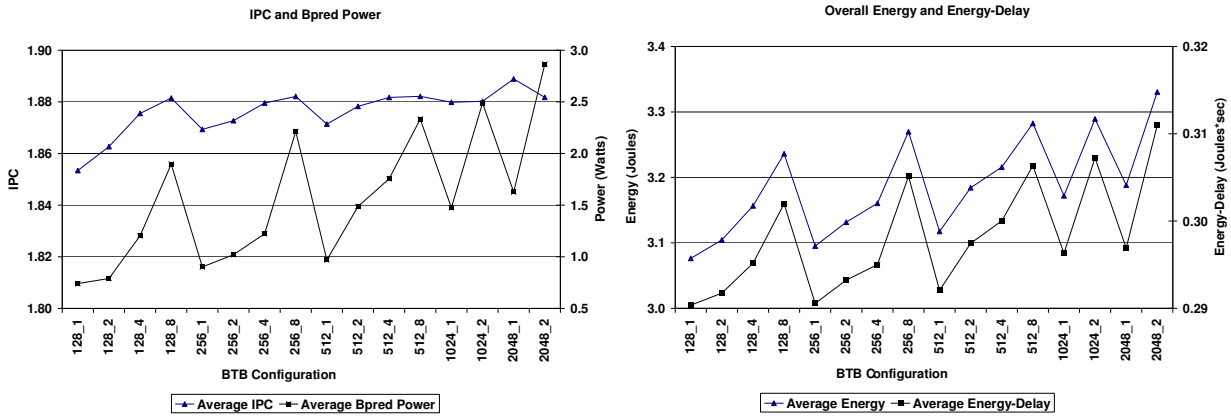
Based on these results, the most energy-efficient configuration for SPEC benchmarks is a direct-mapped, 128-entry BTB. This reduces the total power dissipated by branch prediction from almost 3 W to about 0.75 W; and the possible energy savings by a corresponding amount. However, because so many commercial applications of performance require much larger BTBs and many existing processors employ large BTBs, we follow the advice of our industry contacts and use a 2048-entry BTB for all our other experiments.

D. Potential Gains from Improved Accuracy

Just to illustrate how much leverage branch-prediction accuracy has on both performance and energy, Figure 10 shows the effect of using an idealized, omniscient direction predictor and BTB that never mispredict. In the interests of space, we chose a random sample of seven integer benchmarks. Because a comparison against a control of no prediction would be meaningless, the comparison is done against a known good predictor, a GAs predictor with 32 K-entries and 8 bits of global history. Perfect prediction improves performance for the seven benchmarks by 20% and energy by 16% on average. Issue width was held fixed here; since better prediction exposes more instruction-level parallelism, even greater gains could be realized. These results further illustrate just how much leverage prediction accuracy and its impact on execution time translate into energy savings.

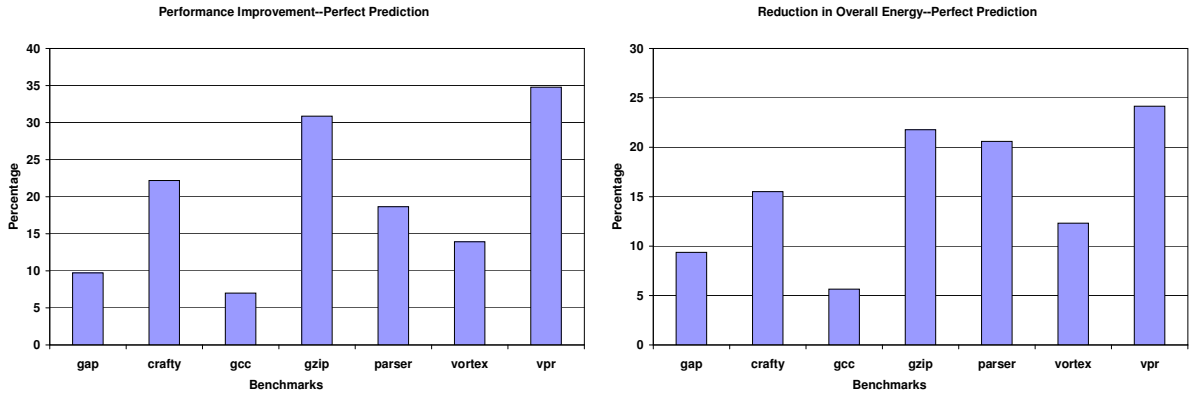
IV. Reducing Power That Stems from Branch Prediction

The previous section showed that in the absence of other techniques, smaller predictors that consume less power actually *raise* processor-wide energy because the resulting loss in accuracy increases running time. This section explores four techniques for reducing processor-wide energy expenditure without affecting predictor accuracy:



(a) (b)

Fig. 9. (a) IPC and power dissipation in the predictor and (b) energy and energy-delay as a function of BTB organization.



(a) (b)

Fig. 10. (a) Performance improvement from perfect prediction and (b) Percentage reduction in overall energy using perfect prediction compared to a 32k-entry global-history (GAs) predictor.

banking; a "prediction probe detector" to identify when direction and target prediction can be avoided; identification of highly biased branches to avoid even more predictions; and pipeline gating.

All remaining experiments use only integer programs because they represent a wider mix of program behaviors. We have chosen a subset of seven integer benchmarks: *gzip*, *vpr*, *gcc*, *crafty*, *parser*, *gap* and *vortex*. These were chosen from the ten original integer benchmarks to reduce overall simulation times while maintaining a representative mix of branch-prediction behavior.

A. Banking

As shown by Jiménez, Keckler, and Lin [17], slower wires and faster clock rates will require multi-cycle access times to large on chip structures, such as branch predictors. The most natural solution to that is banking. We use our modified Cacti [35] model (part of Wattch) to determine the access times for a banked branch predictor. We assume that for any given access only one bank is active at a time; therefore banking not only reduces access times but also saves power spent in the branch predictor, as shown in Figure 11. We plot cycle times normalized with respect to the maximum value, because achievable cycle times are extremely implementation-dependent and might vary significantly from the absolute numbers reported by Cacti. Banking might come at the cost of extra area, (for

example due to extra decoders) but exploring area considerations is beyond the scope of this paper. The number of banks range from 1 in case of smaller predictors of size 2 Kbits or smaller, to 4 in case of larger predictors of size 32 Kbits or 64 Kbits. The number of banks for different branch predictor sizes is given in Table IV.

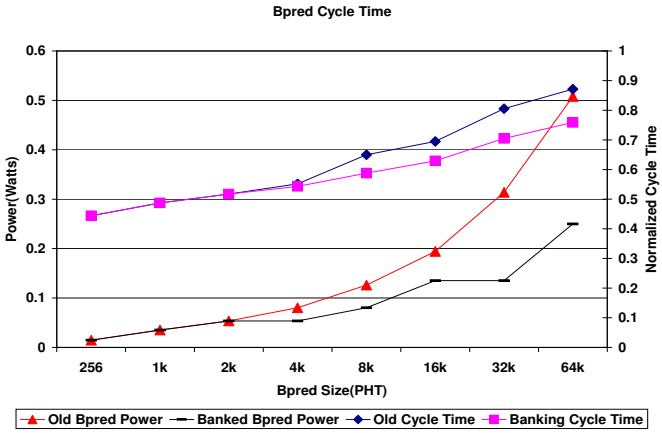


Fig. 11. Cycle time for a banked predictor.

TABLE IV
NUMBER OF BANKS FOR VARIOUS ARRAY SIZES.

Array size	128 bits	4 Kbits	8 Kbits	16 Kbits	32 Kbits	64 Kbits
No. of banks	1	2	2	4	4	4

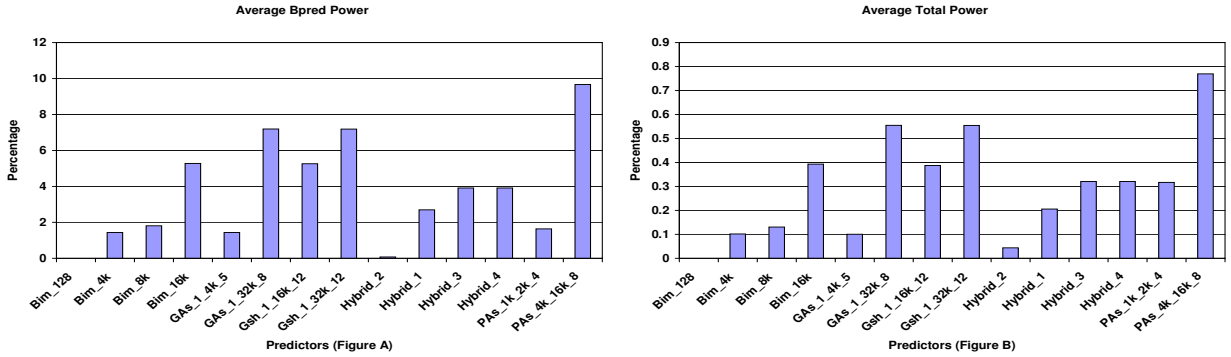
Figures 12 and 13 show the difference between the base simulations and the banked configurations. It can be observed that the largest decrease in predictor power comes for larger predictors. This is exactly as expected, since these predictors are broken into more banks. The large hybrid predictors do not show much difference, however, because they are already broken into three components of smaller sizes and banking cannot help much. Banking results in modest power savings in the branch predictor, and only reduces overall power and energy by about 0.3–0.5%. Of course, we did not look at banking's effect on the BTB, where larger savings would presumably be achieved.

B. Reducing Lookups Using a PPD

A substantial portion of power/energy in the predictor is consumed during lookups, because lookups are performed every cycle in parallel with the I-cache access. This is unfortunate, because we find that the average distance between control-flow instructions (conditional branches, jumps, etc.) is approximately 12 instructions. Figures 14 and 15 show that about 40% of conditional branches have distance greater than 10 instructions, and about 30% of control flow instructions have distance greater than 10 instructions. Jiménez *et al.* report similar data [17]. We also compared these results with gcc-compiled SimpleScalar PISA binaries. The results were similar, so these long inter-branch distances are not due to *nops* or predication. It could be that other programs outside the SPEC suite might have lower distance between branches. In this case the PPD technique proposed here might not perform as well.

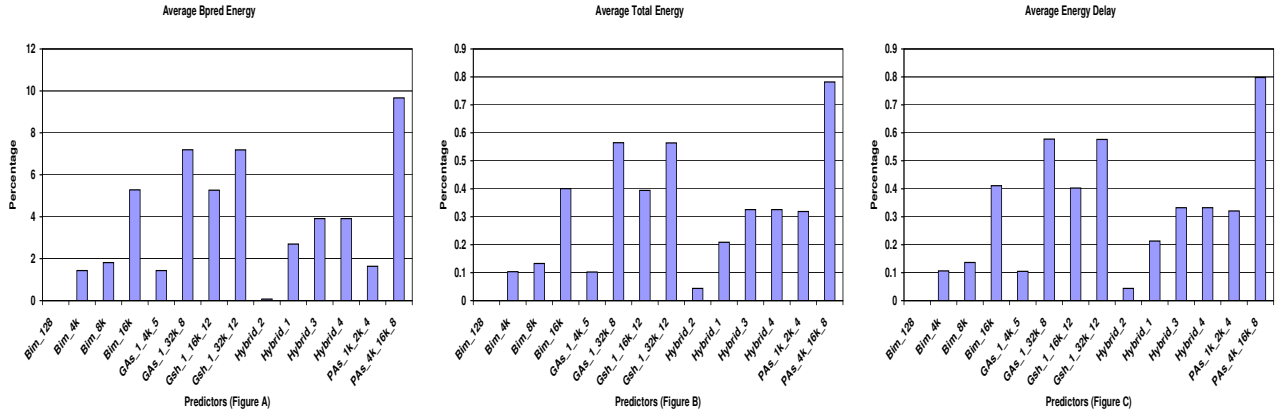
The fact that many cache lines have no control flow instructions suggests that we try to identify dynamically when a cache line has no conditional branches and thus avoid a lookup in the direction predictor, and that we identify when a cache line has no control-flow instructions at all, and thus eliminate the BTB lookup as well.

If the I-cache, BTB, and direction predictor accesses are overlapped, it is not sufficient to store pre-decode bits in the I-cache, because they only become available at the end of the I-cache access, after the predictor access has already begun. Instead, we propose to store pre-decode bits (and possibly other information) in a structure called the *prediction probe detector* (PPD). The PPD is a separate table with a number of entries exactly corresponding to I-cache entries. The PPD entries themselves are two-bit values; one bit controls the direction-predictor lookup, while the other controls the BTB lookup. This makes the PPD 4 Kbits for our processor organization. The PPD is



(a) (b)

Fig. 12. Banking results: (a) Percentage reduction in branch-predictor power and (b) overall power.



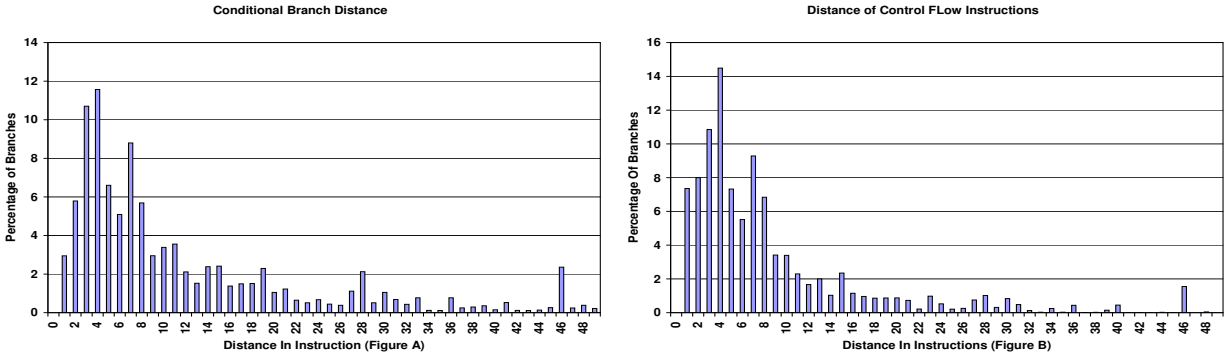
(a) (b) (c)

Fig. 13. Banking Results: (a) Percentage reduction in branch-predictor energy, (b) overall energy, and (c) energy-delay.

updated with new pre-decode bits while the I-cache is refilled after a miss. The PPD is similar to the Instruction Type Prediction Table (ITPT) proposed in [5]. The main difference is that the ITPT only relies on the program address and its prediction can be wrong. The notion that in a superscalar fetch prediction, knowing what type of instructions are in a block/I-cache line is the most critical piece of information was also pointed out in [34]. They use a Block Instruction Table (BIT) to store information about each instruction in the cache line in order to do multiple branch prediction. A schematic of the PPD's role in the fetch stage is shown in Figure 16a. There is a design issue with the PPD for set-associative instruction caches. In traditional implementation of caches, one does not know which way of the set is going to be selected until the I-cache access is complete. The only safe solution is to make the PPD "conservative". So the PPD bits of all the ways of the set are *OR'ed* to guarantee that if a branch is present in any way, then branch prediction is initiated. This scheme is conservative because many times the way that matches and is fetched might not have a branch. Note that our prior work [23] overlooked this issue and hence slightly overstated the benefits of the PPD.

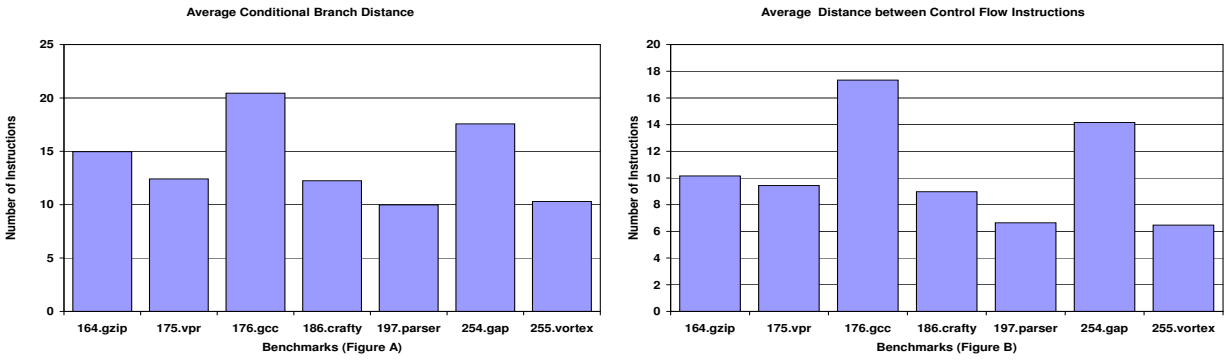
Because the PPD is an array structure and takes some time to access, it only helps if the control bits are available early enough to prevent lookups. A variety of timing assumptions are possible. Exploring fetch-timing scenarios is beyond the scope of this paper, so here we explore two extremes, shown in Figure 16b.

- Scenario 1: The PPD is fast enough so that we can access the PPD and then the BTB sequentially in one cycle. The BTB access must complete within one cycle; more flexibility exists for the direction predictor. The direction predictor is also accessed sequentially after the PPD; but either this access fits entirely within the same cycle, or, as with the 21264, overlaps into the second cycle. The former case is reasonable for smaller predictors; the latter case



(a) (b)

Fig. 14. (a) Average distance (in terms of instructions) between conditional branches. (b) Average distance between control-flow instructions (conditional branches plus unconditional jumps).



(a) (b)

Fig. 15. (a) Average distance (in terms of instructions) between conditional branches for different benchmarks. (b) Average distance between control-flow instructions (conditional branches plus unconditional jumps) for different benchmarks.

applies to large predictors, as shown in both the 21264 and by Jiménez *et al.* [17]. Due to the small size of PPD (4 Kbits) it can be seen from Figure 11 that the time to access the PPD is less than a quarter of the time to access a predictor of size 64 K bits.

- Scenario 2: We also consider the other extreme. Here the assumption is that the BTB and the direction predictor need to be accessed every cycle and these accesses take too long to be placed after the PPD access. Instead, we assume that the PPD access completes in time to stop the BTB/direction-predictor accesses after the bitlines (before the column multiplexor). The savings in this case are clearly less, but the PPD is still able to save the power in the multiplexor and the sense-amps.

Now, instead of accessing the BTB and direction predictor every cycle, we must access only the PPD every cycle. This means we must model the overhead in terms of extra power required for the PPD access and update (about 2% of the total predictor power). Note that we omit the power associated with calculating the appropriate pre-decode bits to update the PPD, assuming that this is a negligible cost if pre-decode hardware already exists.

If the PPD does not prevent enough BTB/predictor lookups, then introducing a PPD may actually increase power dissipation. Fortunately, there are indeed a sufficient number of cache lines that need no BTB/predictor lookups that the PPD is indeed effective. As explained earlier, due to the "conservative" nature of the PPD, our scheme works best in the case of direct mapped caches. A further consideration that must be taken into account is whether the predictor is banked. If the predictor is banked, the PPD saves less power and energy (because some banks are already not being accessed), but the combination of techniques still provides significant savings.

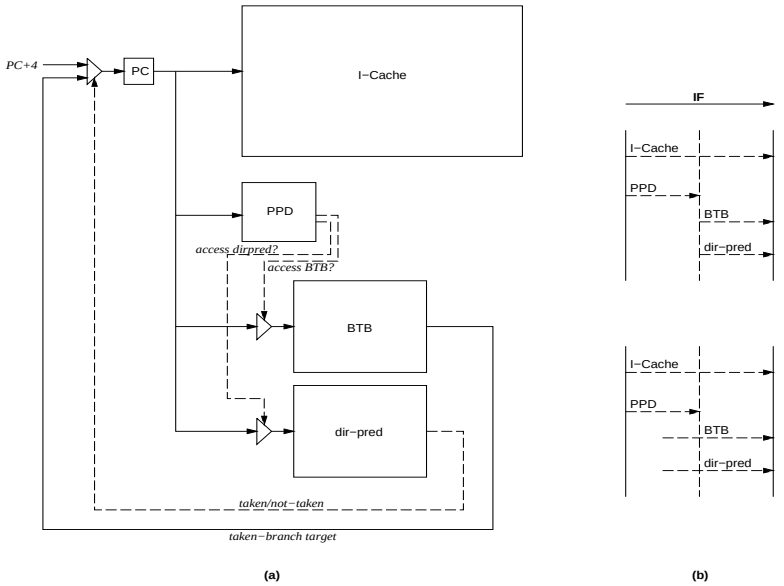


Fig. 16. (a) A schematic of the PPD in the fetch stage. (b) The two timing scenarios we evaluate.

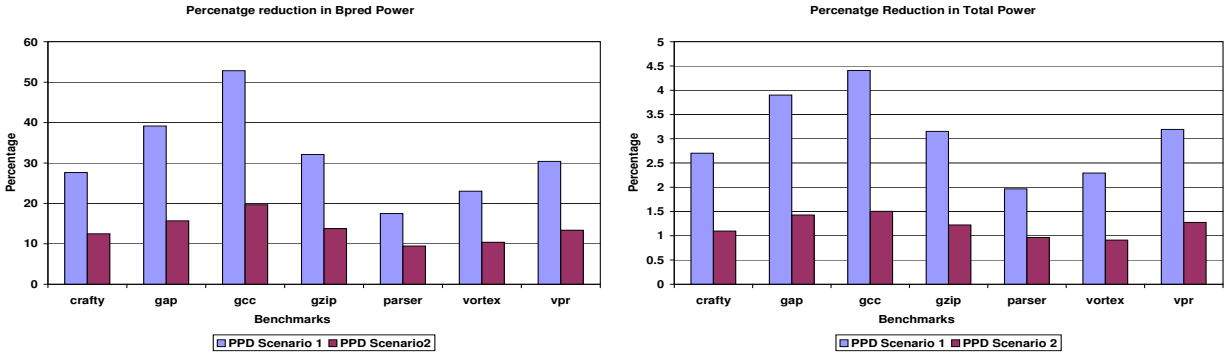
Figures 17–18 show the effect of a PPD on a banked 32 K-entry GAs predictor for a direct mapped cache of the same size. Figure 17 shows the average reduction in power for the branch predictor—31% on average for Scenario 1 and 13% for Scenario 2—and in the overall processor power—3.1% and 1.2%, respectively. We observe a similar trend in Figure 18 for the energy metrics (energy and energy-delay metrics are almost identical): 3% chip-wide improvement for Scenario 1 and 1.2% for Scenario 2. The PPD is small enough and effective enough that spending this extra power on the small PPD brings us larger benefits overall. Since the PPD simply permits or prevents lookups (reducing lookups by 32% on average for a direct-mapped cache), the percentage savings will be similar for other predictor organizations besides GAs. It can also be observed that the greater the average distance between branches for a benchmark, the more the savings one gets from the PPD. For Scenario 2, the power savings are closely tied to the timing assumptions, and further work is required to understand the potential savings in other precharge and timing scenarios.

As mentioned, the increasing likelihood that a branch exists in any way of an I-cache set means that the conservative behavior of the PPD (*OR'ing* bits for each way) reduces its effectiveness for higher associativities. Figure 19a shows the average percentage reduction in power and energy savings as a function of I-cache associativity (same cache size, 64 KB). For an eight-way I-cache there is practically no benefit at all—in fact, the energy and energy-delay results are slightly negative. In a similar way, the increasing likelihood that a branch exists somewhere within a fetch block means that storing PPD information on a per-block basis reduces its effectiveness for larger block sizes. Figure 19b shows the average percentage reduction in power and energy savings as a function of I-cache block size. Note that the effectiveness of the PPD is independent of fetch width, as long as the fetch width is less than or equal to an I-cache line.

If the PPD could somehow determine in advance which way to access, the pre-decode bits for each way would not need to be *OR'ed*, and higher-associativity caches could realize the same savings as the direct-mapped cache. Way prediction [6], [7] could be one way to accomplish this, but we were unable to find a way-prediction implementation that outperformed our conservative PPD. The cost of way prediction is in itself significant, but the main reason we did not find an effective way predictor is that way mispredictions are very costly, like a branch misprediction, which requires flushing the pipeline and a concomitant increase in execution time and energy. Our conservative PPD, on the other hand, never makes mistakes.

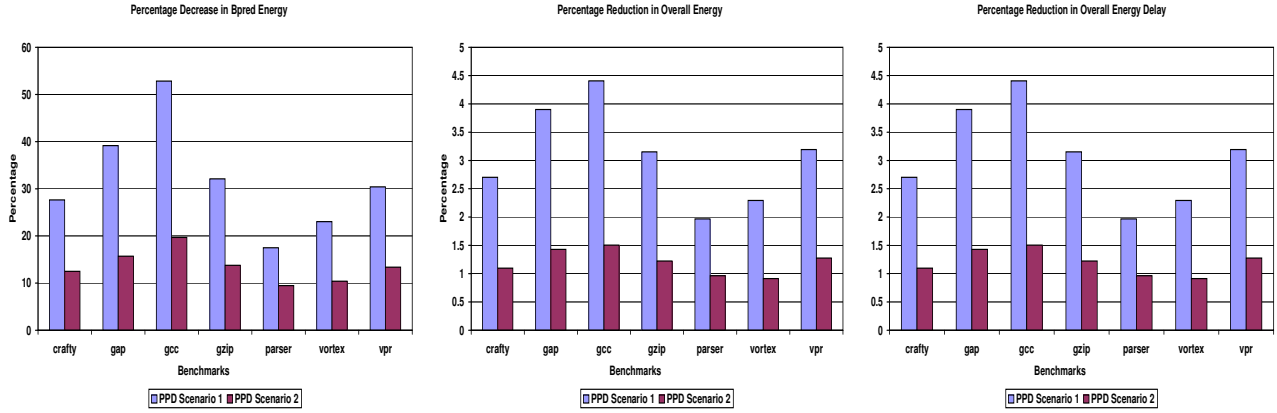
C. Highly Biased Branches

The PPD was also extended to recognize "unchanging" branches (which are always taken or always not taken) [25] by reflecting this property in the bits it stores: a never-taken branch is treated the same as the absence of a branch, and an always-taken branch requires an extra bit in the PPD. Mistakes are simply a new source of branch mis-



(a) (b)

Fig. 17. Net savings with a PPD for a banked, 32 K-entry GAs predictor and direct mapped cache in terms of (a) power in the branch predictor and (b) overall processor power with a PPD. Scenarios 1 and 2 refer to two timing scenarios we model.

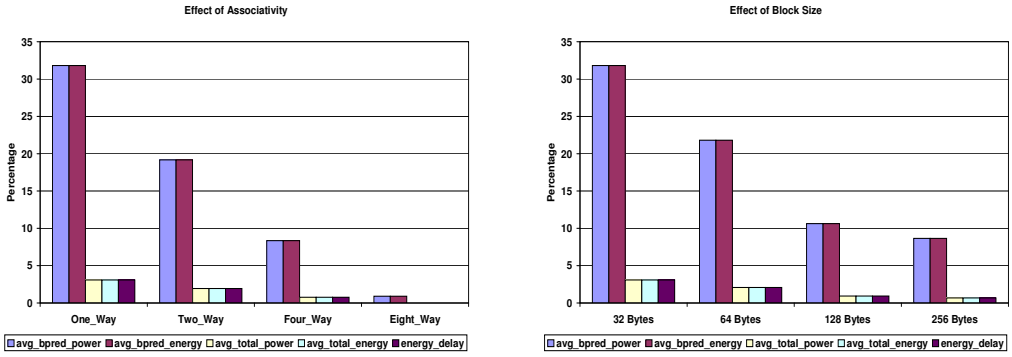


(a) (b) (c)

Fig. 18. Net savings with a PPD for a 32 K-entry GAs predictor and direct mapped cache in terms of (a) energy expended in the branch predictor, (b) total energy, and (c) energy-delay.

predictions. An actual implementation would entail compiler analysis or feedback-directed optimization in which the binary is annotated to convey this information as a bit in each branch instruction. For this study, we modified SimpleScalar's *sim-bpred* to identify different types of branches. Our analyzer goes through a program and stores a 2-tuple {branch address, branch type} value in a file. Wattch then reads this profile and calculates power for each type of branch.

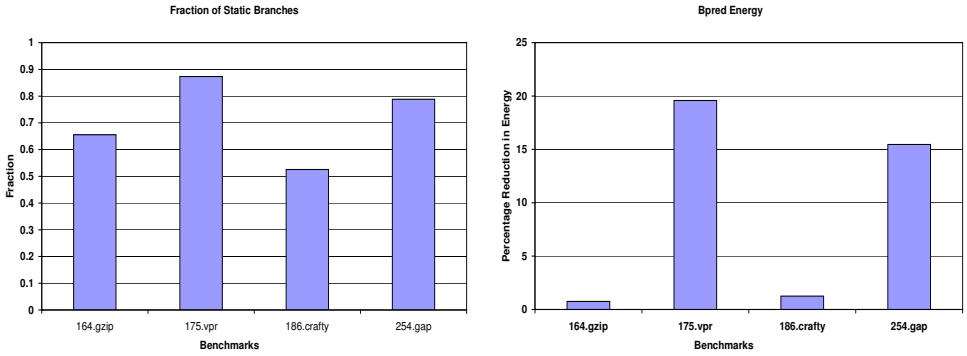
For correlation of power versus branch type, four representative integer benchmarks were characterized. For this experiment, we used training inputs for the profiling and reference inputs for the actual power calculations. This implies that some of the highly biased branches identified in the training input could change in the reference input. This is not a problem, as any incorrect hints will be just a misprediction, and detected in the later stages of pipeline (*i.e.*, for these biased-branch hints, the PPD can now introduce some mispredictions). The plots presented take all these effects into account. From Figure 20a, it can be observed that the "unchanging" branches category of branches make up more than half of all the static branches in the binary. The energy consumed by these branches, however, is not in the above proportion, as seen in Figure 20b. The reason is that "changing" branches are executed more often than "unchanging" branches. The percentage of energy consumed by unchanging branches can range from trivial (*gzip* and *crafty*) to substantial (*vpr* and *gap*). From the figures, it can be seen that unchanging branches consume from less than 1% to close to 20% of branch-prediction energy. Please note that this is the energy that



(a) (b)

Fig. 19. Net savings with a scenario-1 PPD for different (a) I-cache associativities and (b) different I-cache block sizes.

can be saved on top of the savings provided by the PPD described previously, in other words up to an extra 2% or so in total energy savings. This observation makes us conclude that if branch prediction hints were provided for unchanging branches, by profiling or static analysis, then it would lead to a substantial reduction in the power spent in the branch-prediction hardware. Further savings can likely be realized if the profiling is less strict, and instead of being confined to branches that never change direction, it also recognizes branches that change direction rarely enough to minimize the cost of the mispredictions incurred upon a change in direction.



(a) (b)

Fig. 20. (a) Static profile (not dynamic execution) of branches and (b) percentage of branch predictor energy consumed by "unchanging" branches.

D. Pipeline Gating and Branch Prediction

Finally, we briefly explore the power savings that can be obtained using speculation control or "pipeline gating," originally proposed by Manne *et al.* [20]. The stated goal of pipeline gating is to prevent wasting energy on mis-speculated contribution. Pipeline gating is explored here because it is natural to expect that the more accurate the branch predictor, the less gating will help save energy, since there is less mis-speculation to prevent. In fact, even with a very poor predictor, we find that the energy savings are quite small—smaller than previous work using the metric of "extra work" (*i.e.*, extra instructions) [20] would suggest. Furthermore, under certain conditions, pipeline gating can even harm performance and *increase* energy.

Figure 21 shows the operation of pipeline gating. It uses a confidence estimator [14] to assess the quality of each branch prediction. A high-confidence estimate means the prediction of this branch is likely to be correct. A low-confidence estimate means the prediction of this branch is likely to be a misprediction and subsequent computation

will be mis-speculated. These confidence estimates are used to decide when the processor is likely to be executing instructions that may not commit. The number of low-confidence predictions permitted before gating is engaged, N, is a design parameter. Once the number of in-flight low confidence branches, M, reaches the threshold N, the pipeline is gated, stalling the fetch stage.

We modified Wattch to model pipeline gating and did an analysis of power vs. performance. We used the "both strong" estimation method [20], which marks a branch as high confidence only when both of predictors of the hybrid predictor have the same direction (taken or not taken). The "both strong" method uses the existing counters of the branch predictor and thus has no additional hardware requirements. The drawback is that it only works for the hybrid predictor.

We simulated five different hybrid predictor configurations for the same integer benchmarks used elsewhere in this section; and added a new, very small and very poor hybrid predictor; hybrid_0, which has a 256-entry selector, a 256-entry gshare component, and a 256-entry bimodal component. Hybrid_0 of course yields an artificially bad prediction accuracy, so we include it to see the effect on pipeline gating in the extreme case of poor prediction. The results of hybrid_1, hybrid_2, hybrid_3 and hybrid_4 are quite close. We therefore just show results of the smallest one, hybrid_0, and the largest and best one, hybrid_3, in Figure 22. For each metric, results are normalized to the baseline case with no gating.

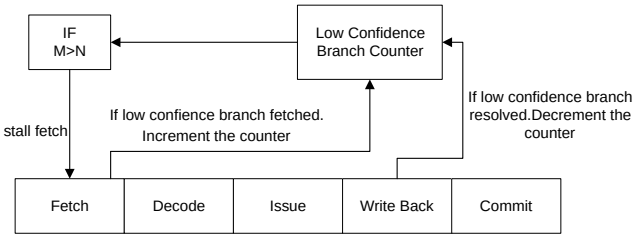
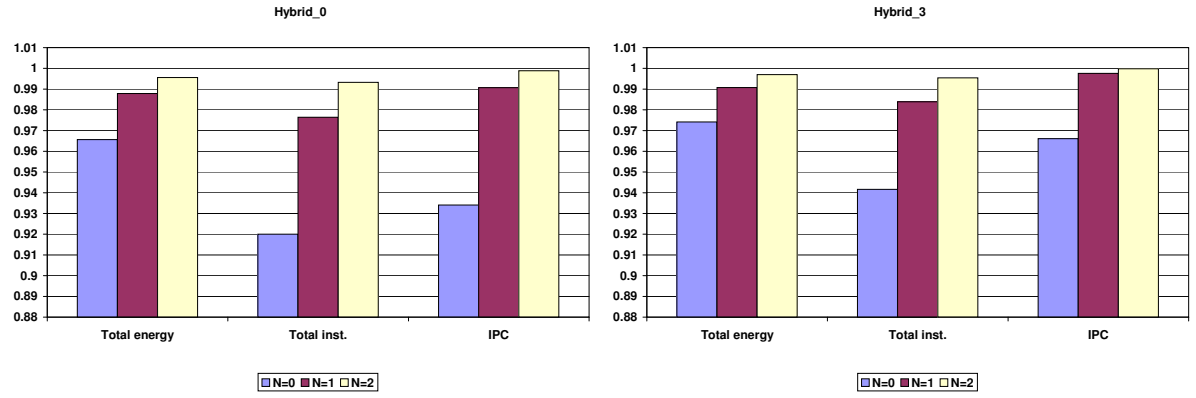


Fig. 21. Schematic showing the operation of pipeline gating.



(a) (b)

Fig. 22. Pipeline gating: overall results of (a) hybrid_0 and (b) hybrid_3.

The results show that only the most aggressive pipeline gating, $N = 0$, has a substantial effect on power. For more relaxed thresholds, the reduction in IPC is small but so are the energy savings.

At threshold $N = 0$, for hybrid_0, the average number of the executed instructions is reduced by 8%; the total energy is reduced by 3.5%, and the IPC is reduced by 6.6%. There are two reasons why the reduction in energy is less than the reduction in instructions would suggest. One reason is that these reduced "wrong path" instructions are squashed immediately when the processor detects the misprediction. Some mis-speculated instructions therefore spend little energy traversing the pipeline, hence preventing these instruction's fetching saves little energy. A second

reason is that errors in confidence prediction sometimes cause pipeline gating to stall the pipeline when the branch was in fact correctly predicted. This slows the program's execution and increases the overall energy consumption.

For hybrid_3 and $N = 0$, the average number of total executed instructions is reduced by 6%; the total energy is reduced by 2.6%, and the IPC is reduced by 3.4%. This suggests that better branch prediction does indeed reduce the benefits of pipeline gating: fewer branches are marked as low confidence and pipeline gating occurs less frequently.

It may be that the impact of predictor accuracy on pipeline gating would be stronger for other confidence estimators. While easy and inexpensive to implement, the accuracy of "both strong" confidence estimation is a function of the predictor organization. This is less true for other confidence estimators [14] that are separate from the predictor. This warrants further study.

The behavior of the benchmark *vortex*—see Figure 23—is especially interesting, because for $N = 0$, the total energy with pipeline gating is larger than without pipeline gating. Prediction accuracy is quite high for *vortex* (97%), so pipeline gating is likely to provide little benefit. Instead, confidence estimation is especially poor for vortex, causing many unnecessary pipeline-gating events. IPC drops by 14%, slowing execution time and increasing energy consumption.

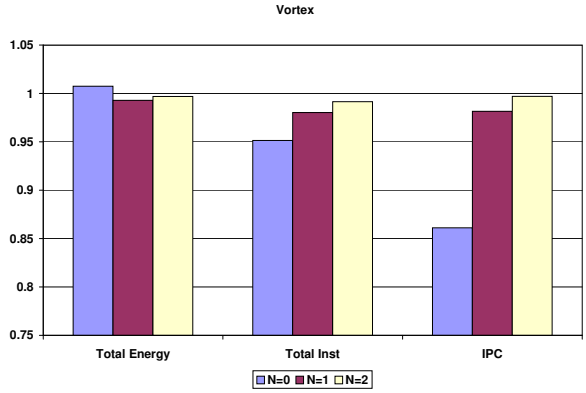


Fig. 23. Pipeline Gating: Results for *vortex*

The preceding results are for an 8-stage pipeline. As pipelines become deeper, pipeline gating might have more effect, so we explored the sensitivity of our results to pipeline depth, with results shown in Figure 24. All the extra stages were added at the front end of the pipeline as extra stages of decoding/renaming/enqueuing. Although this is not exactly representative of how current pipeline stages are allocated as pipelines grow deeper, it avoids artifacts from specific choices of how to allocate stages in other parts of the pipeline.

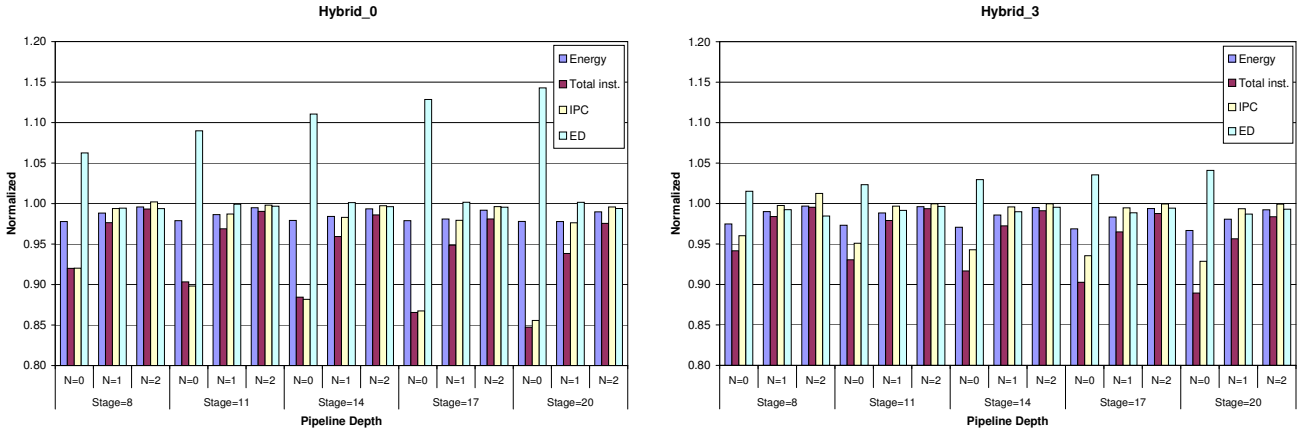
As the pipeline grows deeper, the number of possible in-flight branches grows, and the reduction in total instructions executed grows as well. Unfortunately, any erroneous pipeline gating incurs larger misprediction penalties, so performance suffers too. This means that as the pipeline gets deeper, energy and energy-delay savings remain disappointing. In fact, energy delay grows considerably for the $N = 0$ case.

Overall, our results show that pipeline gating can be modestly helpful in reducing energy but that (1) energy savings are substantially less than the previous metric of "extra work" suggests, and that (2) for benchmarks with already high prediction accuracies, pipeline gating may substantially reduce performance and increase energy.

V. Conclusions

The branch predictor structures, which are the size of a small cache, dissipate a non-trivial amount of power—about 7% of the total processor-wide power—and their accuracy controls how long the program runs and therefore has a substantial impact on energy. This paper explores the effects of branch predictor organization on power and energy expended both locally within the branch predictor and globally in the chip as a whole.

Section III showed that for all the predictor organizations we studied, total energy consumed by the chip is affected more strongly by predictor accuracy than by the local energy consumed by the predictor, because more accurate predictors reduce the overall running time. We found that for integer programs, large but accurate predictors actually reduce total energy. For example, a large hybrid predictor uses 9% more energy than a bimodal predictor



(a) (b)

Fig. 24. Pipeline Gating: Performance and energy results as a function of pipeline depth. All results are normalized to the values obtained with no gating. (a) hybrid_0, (b) hybrid_3.

but actually yields a 6% savings in total, chip-wide energy. For floating-point programs, the energy curves are flat across the range of predictor organizations, which means that choosing a large predictor to help integer programs should not cause harm when executing floating-point programs. This suggests that if the die and cooling budgets can afford it, processors for embedded systems that must conserve battery life might actually be better off with large, aggressive branch predictors rather than lower-power but less-accurate predictors.

Section IV showed that there are some branch-prediction-related techniques that do save energy without affecting performance. Banking reduces both access time and power consumption by accessing only a portion of the total predictor structure. A *prediction probe detector* (PPD) uses pre-decode bits to prevent BTB and predictor lookups, saving as much as 30% in energy expended in the predictor and 3% of total energy. These savings can be extended by annotating binaries to convey which branches are likely to be highly biased, providing further opportunities to prevent predictor lookups. On the other hand, larger I-cache blocks, wide fetch widths, and higher associativities reduce the benefits of the PPD. Finally, we revisited pipeline gating and showed that, for a variety of pipeline depths, it offers little energy savings and can actually increase energy consumption due to slower execution.

Overall, we hope that the data presented here will serve as a useful guide to help chip designers and other researchers better understand the interactions between branch behavior and power and energy characteristics, and help identify the important issues in balancing performance and energy when choosing a branch predictor design.

ACKNOWLEDGMENTS

This material is based upon work supported in part by the National Science Foundation under grants nos. CCR-0082671 and CCR-0105626, NSF CAREER MIP-9703440, a grant from Intel MRL, and by an exploratory grant from the University of Virginia Fund for Excellence in Science and Technology. We would also like to thank John Kalamatianos for helpful discussions regarding branch-predictor design; Margaret Martonosi and David Brooks for their assistance with Wattch and for helpful discussions on various power issues; Zhigang Hu for help with the EIO traces; and the anonymous reviewers for a number of insightful and helpful suggestions.

REFERENCES

[1] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–59, Nov. 1999.

[2] R. I. Bahar and Srilatha Manne. Power and energy reduction via pipeline balancing. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, June 2001.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

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

[5] B. Calder and D. Grunwald. Fast & accurate instruction fetch and branch prediction. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 2–11, May 1994.

[6] B. Calder and D. Grunwald. Next cache line and set prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 287–96, June 1995.

[7] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, pages 244–253, Feb. 1996.

[8] P.-Y. Chang, E. Hao, and Y. N. Patt. Alternative implementations of hybrid branch predictors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 252–57, Dec. 1995.

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

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

[11] S. Ghiasi, J. Casmira, and D. Grunwald. Using IPC variation in workload with externally specified rates to reduce power consumption. In *Proceedings of the Workshop on Complexity-Effective Design*, June 2000.

[12] K. Ghose and M. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, pages 70–75, Aug. 1999.

[13] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9), Sep. 1996.

[14] D. Grunwald, A. Klauser, S Manne, and A. Pleszkun. Confidence estimation for speculation control. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 122–31, June 1998.

[15] Z. Hu, P. Juang, P. Diodato, S. Kaxiras, K. Skadron, M. Martonosi, and D. W. Clark. Managing leakage for transient data: Decay and quasi-static memory cells. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, pages 52–55, Aug. 2002.

[16] Z. Hu, P. Juang, K. Skadron, D. Clark, and M. Martonosi. Applying decay strategies to branch predictors for leakage energy savings. In *Proceedings of the 2002 International Conference on Computer Design*, pages 442–45, Sept. 2002.

[17] D. A. Jiménez, S. W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 67–77, Dec. 2000.

[18] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 microprocessor architecture. In *Proceedings of the 1998 International Conference on Computer Design*, pages 90–95, Oct. 1998.

[19] J. Kin, M. Gupta, and W. Mangione-Smith. The filter cache: An energy-efficient memory structure. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 184–93, Dec. 1997.

[20] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: speculation control for energy reduction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 132–41, June 1998.

[21] S. McFarling. Combining branch predictors. Tech. Note TN-36, DEC WRL, June 1993.

[22] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Oct. 1992.

[23] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan. Power issues related to branch prediction. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, pages 233–44, Feb. 2002.

[24] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan. Power issues related to branch prediction. Technical Report CS-2001-25, University of Virginia Department of Computer Science, Nov. 2001.

[25] H. Patil and J. Emer. Combining static and dynamic branch prediction to reduce destructive aliasing. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, Jan. 2000.

[26] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 259–71, Dec. 1998.

[27] K. Skadron, D. W. Clark, and M. Martonosi. Speculative updates of local and global branch history: A quantitative analysis. *Journal of Instruction-Level Parallelism*, Jan. 2000. (http://www.jilp.org/vol2).

[28] K. Skadron, M. Martonosi, and D. W. Clark. A taxonomy of branch mispredictions, and alloyed prediction as a robust solution to wrong-history mispredictions. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, pages 199–206, Oct. 2000.

[29] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 135–48, May 1981.

[30] P. Song. UltraSparc-3 aims at MP servers. *Microprocessor Report*, pages 29–34, Oct. 27 1997.

[31] Standard Performance Evaluation Corporation. SPEC CPU2000 Benchmarks. http://www.specbench.org/osg/cpu2000.

[32] W. Tang, R. Gupta, and A. Nicolau. Design of a predictive filter cache for energy savings in high performance processor architectures. In *Proceedings of the 2001 International Conference on Computer Design*, pages 68–73, Sept. 2001.

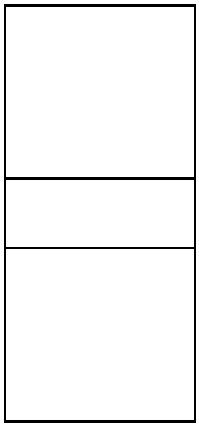
[33] J. Turley. ColdFire doubles performance with v4. *Microprocessor Report*, Oct. 26 1998.

[34] S. Wallace and N. Bagherzadeh. Multiple branch and block prediction. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 94–103, Feb. 1997.

[35] S. J. E. Wilton and N. P. Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–88, May. 1996.

[36] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61, November 1991.

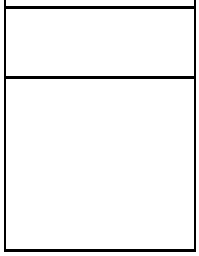
[37] Z. Zhu and X. Zhang. Access-mode predictions for low-power cache design. *IEEE Micro*, 22(2):58–71, Mar.-Apr. 2002.



Dharmesh Parikh received a BTech in Computer Science and Engineering from the Institute of Technology, BHU, India in 1996, and an MCS in computer science from the University of Virginia in 2003. His research interests lie in computer architecture, operating systems, and networks.

Kevin Skadron has been an assistant professor in the University of Virginia Department of Computer Science since 1999. His research interests include temperature- and power-aware computing, simulation methodology, and computer architecture. He earned his Ph.D. at Princeton University studying the areas of branch prediction and write-buffer design, after doing his undergraduate work at Rice University. He was the general chair for the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT), has served on numerous program committees, and is Associate Editor-in-Chief of Computer Architecture Letters. In 2002 he received the NSF CAREER award to investiage various aspects of temperature-aware computing, leakage, and feedback control for adaptive architectures. Skadron is a member of IEEE and ACM, and can be reached at skadron@cs.virginia.edu.

Yan Zhang received the B.S. and M.S degree in Electrical Engineering from Tsinghua University, Beijing, China, in 1997 and 2000, respectively. He is currently pursuing a Ph.D. in Electrical and Computer Engineering at the University of Virginia. His research interests include low- power VLSI design and architecture-level power modeling.



Mircea Stan received the Ph.D. and M.S. degrees in Electrical and Computer Engineering from the University of Massachusetts at Amherst, and the Diploma in Electronics and Communications from "Politehnica" University in Bucharest, Romania. Since 1996 he has been with the Department of Electrical and Computer Engineering at the University of Virginia, first as an assistant professor, and as an associate professor since 2002. Prof. Stan is teaching and doing research in the areas of high-performance and low-power VLSI, mixed-mode analog and digital circuits, computer arithmetic, embedded systems, and nanoelectronics. He has more than eight years of industrial experience as an R&D Engineer in Bucharest, Tokyo and Atlanta, and has been a visiting faculty at IBM in 2000, and at Intel in 2002 and 1999. He was the general chair for the Great Lakes Symposium on VLSI (GLSVLSI) 2003 and has been an Associate Editor for the IEEE Transactions on VLSI Systems since 2001. In 1997 Prof. Stan has received the NSF CAREER Award for investigating low-power design techniques. He is a senior member of the IEEE, a member of ACM and Usenix, and also of Eta Kappa Nu, Phi Kappa Phi and Sigma Xi.