Power Issues Related to Branch Prediction

Dharmesh Parikh†, Kevin Skadron†, Yan Zhang‡, Marco Barcella‡, Mircea R. Stan‡
Depts. of †Computer Science and ‡Electrical and Computer Engineering
University of Virginia
Charlottesville, VA 22904
{dharmesh,skadron}@cs.virginia.edu, {yz3w,mb6nj,mrs8n}@ee.virginia.edu

Abstract

This paper explores the role of branch predictor organization in power/energy/performance tradeoffs for processor design. We find 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. Two techniques, however, provide substantial reductions in power dissipation without harming accuracy. Banking reduces the portion of the branch predictor that is active at any one time. And a new on-chip structure, the prediction probe detector (PPD), can use pre-decode bits to entirely eliminate unnecessary predictor and branch target buffer (BTB) accesses. Despite the extra power that must be spent accessing the PPD, it reduces local predictor power and energy dissipation by about 45% and overall processor power and energy dissipation by 5–6%.

1. 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 predictor's power dissipation without harming performance. Branch prediction has long been an important area of study for micro-architects, because prediction accuracy is such a powerful lever over performance. Power-aware computing has also long 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 use high-performance processors but battery life remains a concern, and heat dissipation has become a design obstacle as it becomes more difficult to developing cost-effective packages that can safely dissipate the 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. Yet the branch predictor, including the BTB, is the size of a small cache and dissipates a non-trivial amount of power—10% or more of the total processor's power dissipation—and its accuracy controls how much mis-speculated execution is performed 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.

Reducing power dissipated in the branch predictor can actually have harmful effects. If reducing the power spent in the predictor comes at the expense of predictor accuracy and hence program performance, this localized reduction may actually increase total power (*i.e.*, energy) dissipated by the processor by making programs run longer. Fortunately, not all the techniques that reduce localized power dissipation 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 reducing precharge costs, and banking need not have any effect on prediction accuracy. Eliminating some branch-predictor accesses altogether is an even more powerful way to reduce power.

Overall, there are four main levers for controlling the branch predictor's power characteristics:

1. *Accuracy*: For a given predictor size, better prediction accuracy will not change power dissipation within the predictor, but will make the program run faster and hence reduce total energy.
2. *Configuration*: Changing the table size(s) and 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.
4. *Number of Updates*: Reducing the number of predictor updates is another obvious source, but is a less powerful lever because mis-speculated computation means that there are more lookups than updates, and we do not further consider it here.

Branch predictors predict conditional branches taken or not taken according to the outcome of the previous ones. 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 very important for accurate estimations of performance and power consumption.

1.1. Contributions

This work extends the Wattch 1.02 [3] power/performance simulator to more accurately model branch-predictor behavior, and then uses the extended system to:

- Characterize the power/performance characteristics of different predictor organizations. As a general rule, to reduce overall energy consumption it is worthwhile to spend *more* power in the branch predictor if it permits a more accurate organization that improves running time.
- Explore the best banked predictor organizations. Banking improves access time and cuts power dissipation at no cost in predictor accuracy.
- 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 over 50%.
- Revisit and develop new techniques for speculation control via pipeline gating [16]. Even despite adaptive speculation control based on recent predictor accuracy, 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 on power issues for a representative sample of the most widely used predictor types: bimodal [23], GAs/gshare [17, 29], PAs [29], and hybrid [17]. We focus mostly on the branch predictor that predicts directions of conditional branches, and except for eliminating unnecessary accesses using the PPD, do not explore power issues in BTB. The BTB has a number of design choices orthogonal to choices for the direction predictor. Exploring these is simply beyond the scope of this paper. Please note that data for the "predictor power" includes power for both the direction predictor and the BTB.

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/performance design goals.

1.2. Related Work

Some prior research has characterized power in other parts of the processor. Pipeline gating was presented by Manne *et al.* [16] as an efficient technique to prevent misspeculated 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. He explores the performance and energy implications and shows that a small performance degradation can produce significant reduction in cache energy dissipation. Ghose and Kamble [10] look at sub-banking and other organizational techniques for reducing energy dissipation in the cache. Kin *et al.* [15] and Tang *et al.* [26] describe filter caches and predictive caches, which utilize a small "L0" cache to reduce accesses and energy expenditures in subsequent levels. Our PPD performs a somewhat analogous filtering function, although it is not itself a branch predictor. Ghiasi *et al.* [9] reasoned that reducing power at the expense of performance is not always correct. They propose 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 then choose between the extant methods that achieve the specified performance while reducing power. Finally, Bahar and Manne [2] propose an architectural solution to the power problem that retains performance while reducing power. 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.

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 3 then explores tradeoffs between predictor accuracy and power/energy characteristics, and Section 4 explores changes to the branch predictor that save energy without affecting performance. Finally, Section 5 summarizes the paper.

2. Simulation Technique and Metrics

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

2.1. 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 [14], 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 [20, 21], and by changing the fetch engine to recognize cache-line boundaries. A more important change to the fetch engine is that we now charge a predictor 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 must typically all be 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 we explore in Section 4.

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 memory 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 cycles
L2	Unified,2MB,4-way LRU 32B blocks,11-cycle latency,WB
Memory latency	100 cycles
TLB Size	128-entry, fully assoc., 30-cycle miss penalty
Branch Predictor	
Branch target buffer	2048-entry, 2-way
Return-address-stack	32-entry

Table 1. Simulated processor configuration, which matches an Alpha 21264 as much as possible.

Unless stated otherwise, this paper uses the baseline configuration as shown in Table 1, which resembles as much as possible the configuration of an Alpha 21264 [14]. 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 [5]. As most processors 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, for Wattch technology parameters we use the process parameters for a 0.18μm process at V_{dd} 2.0V and 1200 MHz. 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.

We have also enhanced Wattch to account for power expenditures in column decoders and to better identify the best "square-ified" structure for the branch predictor. The column decoders increase power dissipation in the branch predictor by 10-15%, and this can be important when modeling changes that increase the size or numbers of decoders, such as with banking. Without a proper accounting for power expended in the column decoders, comparators and drivers, resulting conclusions might be incorrect. Accurate choice of the form factor of data arrays is important for similar reasons. Although we are accustomed to thinking of array structures (branch predictors, caches etc.) with their logical dimensions, their implementations is different and is based on delay and energy considerations. The physical dimensions are typically as square as possible so that the bitline and wordline lengths are minimized. Details about these modifications can be found in an extended version of this paper [19].

2.2. Benchmarks

We evaluate the programs from the SPECcpu2000 [25] benchmark suite. Basic branch characteristics are presented in Table 2. Branch mispredictions also induce other negative consequences, like cache misses due to mis-speculated instructions, but we do not treat those second-order effects here. All benchmarks were 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 mainly focus on the programs from the integer benchmark suite because the floating point benchmarks have very good prediction accuracy and very few dynamic branches. 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.

2.3. Metrics

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

- Average Instantaneous Power: The total power consumed on a per-cycle basis. This metric is important as it directly translates into heat and also gives some indication of current-delivery requirements.
- Energy: Energy is equal to the product of the average power dissipated by the processor and the total execution time. This metric is important as it translates directly to battery life.
- Energy-Delay Product: This metric [11] is equal to the product of energy and delay (*i.e.*, execution time). Its advantage is that it takes into account both the performance and power dissipation of a microprocessor.
- Performance: We use the common metric of instructions per cycle (IPC).

3. Performance-Power Tradeoffs Related to Branch Prediction

3.1. Branch Predictors Studied

The bimodal predictor [23] 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

	Dynamic Unconditional Branch Frequency	Dynamic Conditional Branch Frequency	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%	83.2%	86.99%
wupwise	2.02%	7.87%	90.38%	96.62%
swim	0.00%	1.29%	99.31%	99.68%
mgrid	0.00%	00.28%	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%

Table 2. Benchmark summary.

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 [27]; 4 K-entry is the same size as that in the Alpha 21064 [7] and is at the point of diminishing returns for bimodal predictors, although the 21164 used an 8 K-entry predictor [8]. The gshare predictor [17], shown in Figure 1a, is a variation on the two-level GAg/GAs global-history predictor [18, 29]. 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 and both fit into the desired index width. This paper models a 4 K-entry GAs predictor with 5 bits of history; 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 [24]); a 32 K-entry gshare predictor, also with 12 bits of history; and a 32 K-entry GAs predictor with 8 bits of history. Instead of using global history, a two-level predictor can track history on a per-branch basis. 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. Figure 1b shows a PAs predictor. 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 known 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.* in [22].

Because most programs have some branches that perform better with global history and others that perform better with local history, a hybrid predictor [6, 17], Figure 1c combines the two. 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 it accommodates branches regardless of whether they prefer local or global history. This paper models four hybrid configurations:

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 [14] and is depicted in Figure 1c. It contains 26 Kbits of information.
2. Hybrid_2: a hybrid predictor with a 1 K-entry selector that only 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 only uses 10 bits of global history to index its PHT; a global-history component predictor of 16K entries that uses 7 bits of

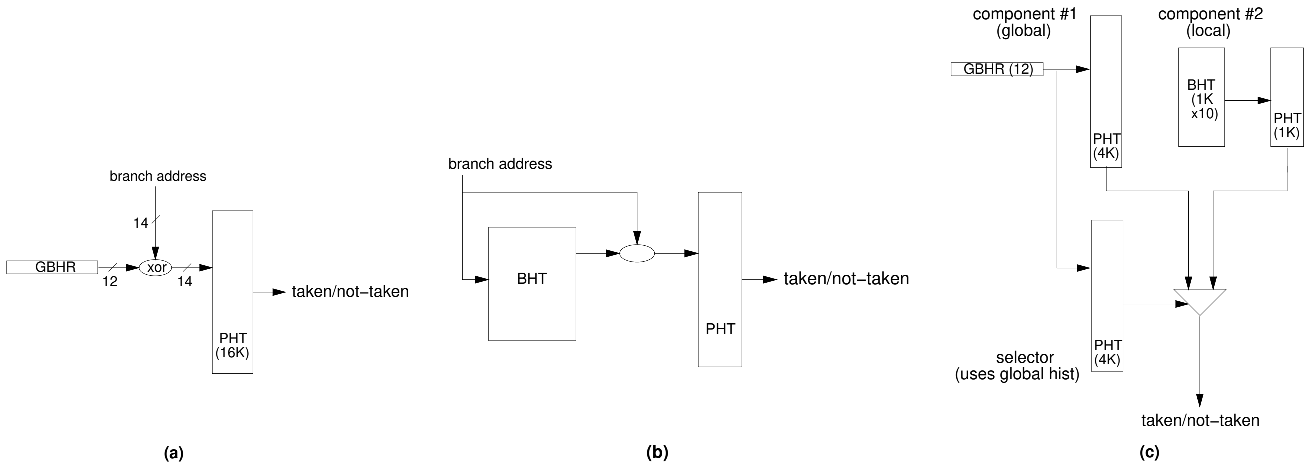

Figure 1. (a) A gshare global-history branch predictor like that in the Sun UltraSPARC-III. (b) A PAs local-history predictor. (c) A hybrid predictor like that in the Alpha 21264.

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 only 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.

Hybrid_2, 3, and 4 are based on configurations found to perform well by Skadron *et al.* in [22].

3.2. Base Simulations for Integer Benchmarks

We now examine the interaction between predictor configuration, performance, and power/energy characteristics. In our discussion below, the term "average", wherever it occurs, means the arithmetic mean for that metric across all the benchmarks simulated.

Figure 2 (left) presents the average branch predictor direction accuracy for integer benchmarks, and Figure 2 (right) 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 thicker and darker curve. 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 16K 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.

Together Figure 3a and Figure 3b show that processor-wide energy is primarily a function of predictor *accuracy* and not 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 chip-wide energy to be consumed. The hybrid_4 predictor, for example, consumes about 7% less chip-wide energy than bimodal-4K despite consuming 13% 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 to use *large* and aggressive predictors if the die 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 4 shows that power dissipation in the predictor 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 unaffected by program running time. Average activity outside the branch predictor is roughly the same regardless of predictor accuracy, so predictor size becomes the primary lever on overall power. Figure 4 also shows that if power dissipation is more important than energy, GAs_1_4K, gshare_16K, or one of the smaller hybrid predictors is the best balance of power and performance.

Finally, Figures 2, 3 and 4 also show data for individual benchmarks. It is clear that the group *crafty, gzip, vortex*, and *gap*, with high prediction rates, have high IPCs and correspondingly low overall energy and energy-delay despite higher predictor and total instantaneous power. The group

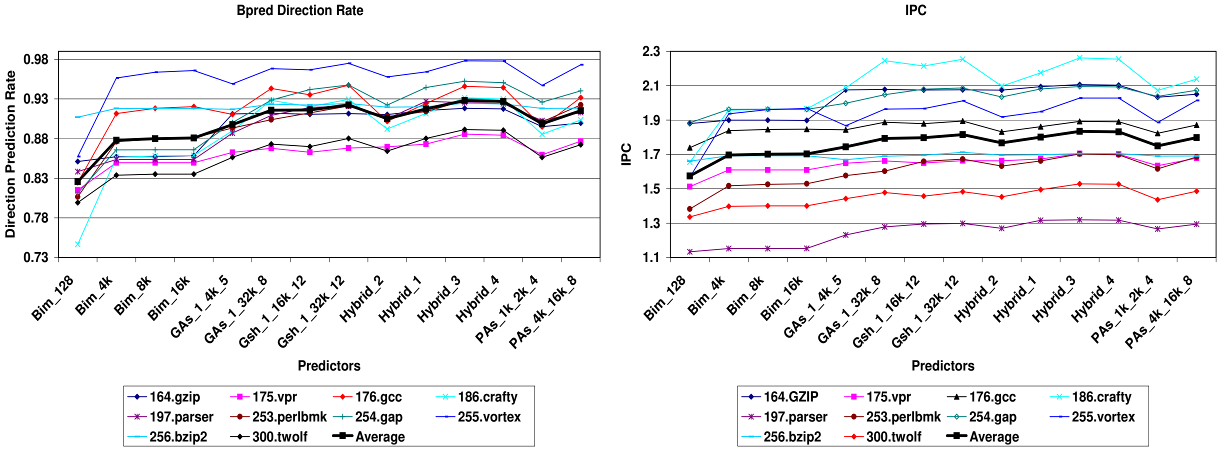

Figure 2. (a) Direction-prediction accuracy and (b) IPC for SPECint2000 for various predictor organizations. For each predictor type, the predictors are arranged in order of increasing size along the X-axis. The arithmetic mean is the dark curve in each of the graphs.

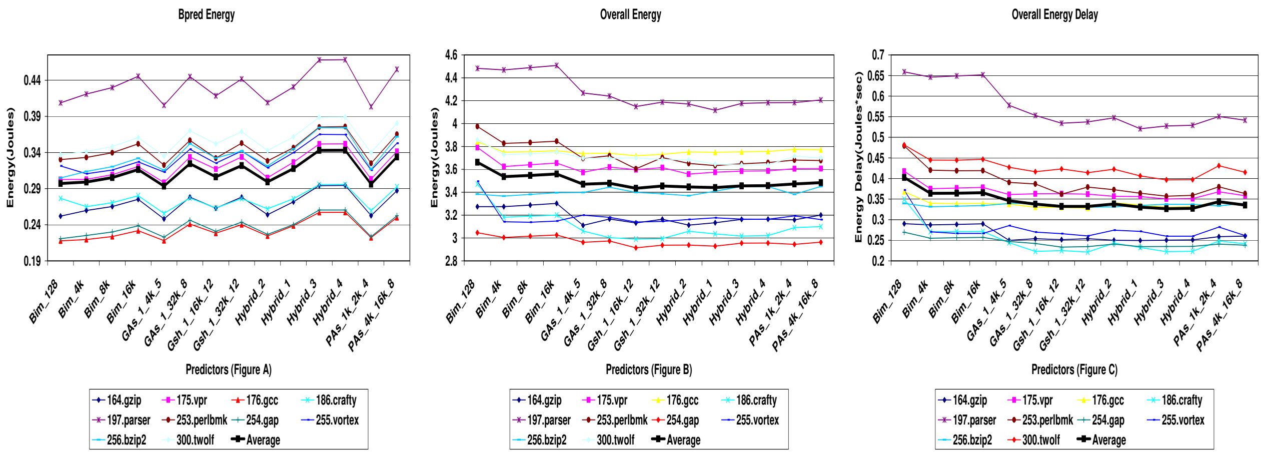

Figure 3. Energy expended in (a) the branch predictor and (b) the entire processor, and (c) energy-delay for the entire processor for SPECint2000.

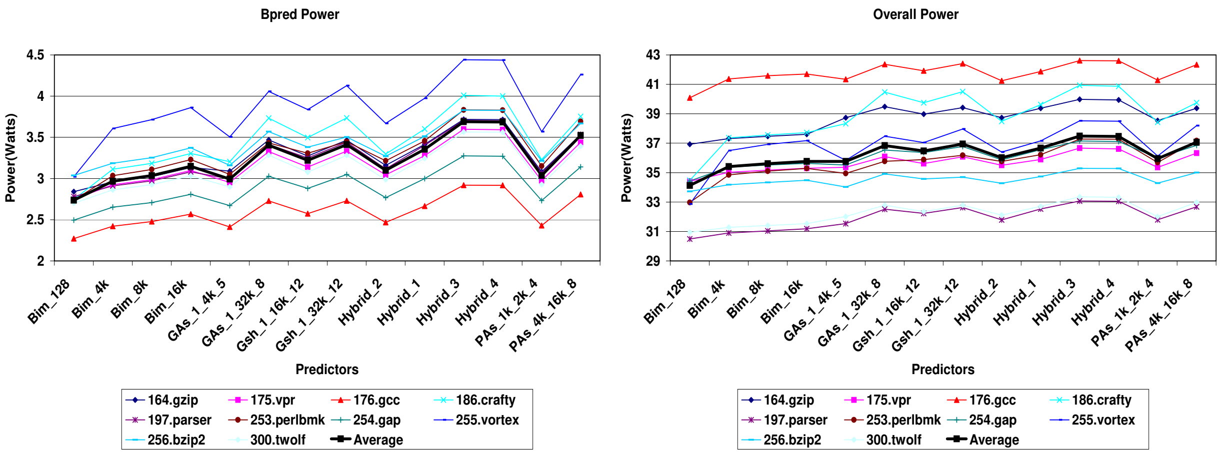

Figure 4. Power dissipation in (a) the branch predictor and (b) the entire processor for SPECint2000.

parser, twolf, and *vpr*, at the other extreme, have the exact opposite properties. This merely reinforces the point that almost always there would be no rise (and in fact usually a decrease) in total energy if we use larger branch predictors to obtain faster performance! We repeated 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 by larger predictors. Detailed plots can be found in an extended version of this paper [19].

4. 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 three techniques for *reducing* processor-wide energy expenditure without affecting predictor accuracy. All remaining experiments use only the 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 our ten original integer benchmarks to reduce overall simulation times but maintain a representative mix of branch-prediction behavior.

4.1. Banking

As shown by Jiménez, Keckler, and Lin [13], 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 used a slightly modified version of Cacti [28] 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 saves us power spent in the branch predictor but also reduces access time, as shown in Figure 5. 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 one in case of smaller predictors of size 2 Kbits or smaller to four 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 3.

Figures 6 and 7 show the average reduction in power and energy for the branch predictor and for the overall processor with respect to the base simulation results. 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, but only reduces overall power and energy by about 1%.

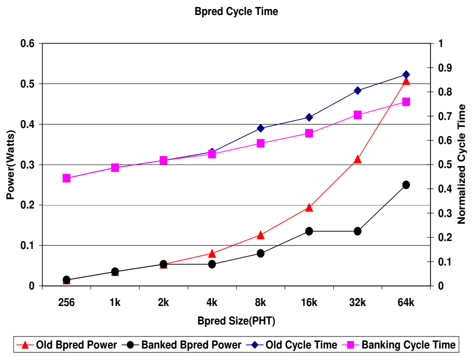

Figure 5. Cycle time for a banked predictor.

	No. of Banks
128bits	1
4Kbits	2
8Kbits	2
16Kbits	4
32Kbits	4
64Kbits	4

Table 3. Number of banks.

4.2. 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 12 instructions. Figure 8 shows that 40% of conditional branches have distance greater than 10 instructions, and 30% of control flow instructions have distance greater than 10 instructions. Jiménez *et al.* report similar data [13]. 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.

This suggests that we should identify when a cache line has no conditional branches so that we can avoid a lookup in the direction predictor, and that we identify when a cache line has no control-flow instructions at all, so that we can 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 accesses must begin.

Instead, we propose to store pre-decode bits (and possi-

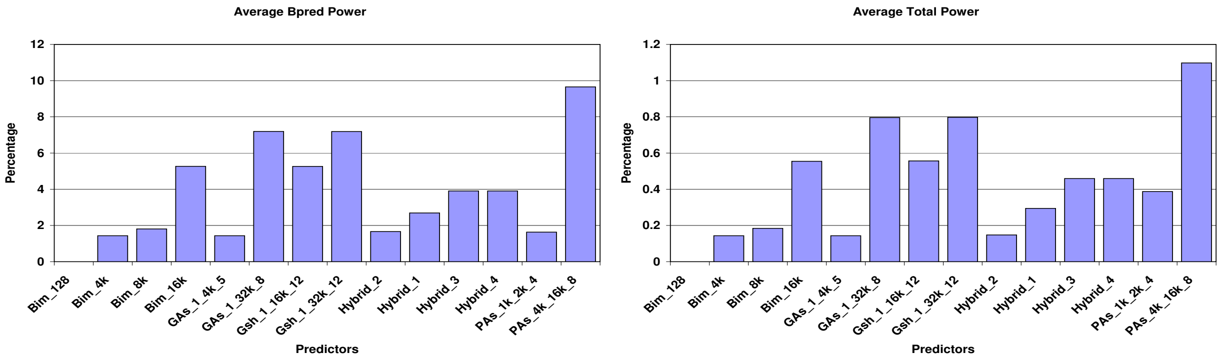

Figure 6. Banking results: percentage reduction in branch-predictor power (left) and overall power (right).

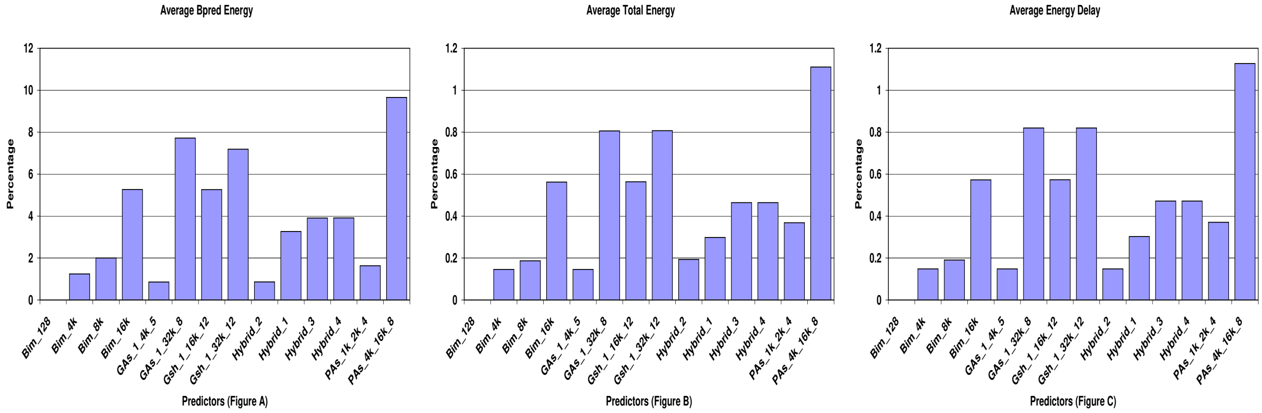

Figure 7. Banking Results: (a) Percentage reduction in branch-predictor energy, (b) overall energy, and (c) energy-delay.

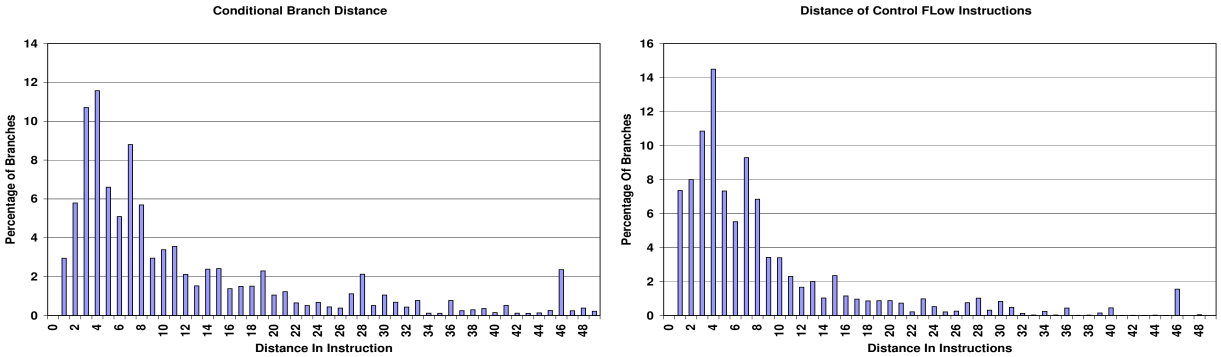

Figure 8. (a) Average distance (in terms of instructions) between conditional branches. (b) Average distance between control-flow instructions (conditional branches plus unconditional jumps).

bly 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 updated with new pre-decode bits while the I-cache is refilled after a miss. A schematic of the PPD's role in the fetch stage is shown in Figure 9a.

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 timings scenarios is a paper in its own right, so here we explore two extremes, shown in Figure 9b.

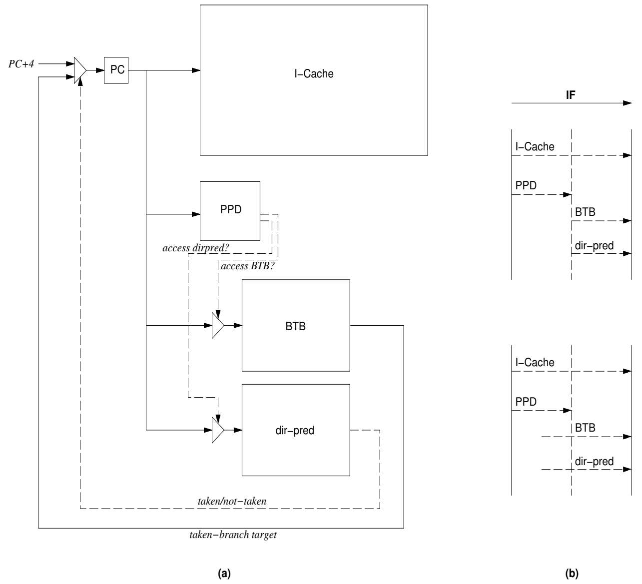

Figure 9. (a) A schematic of the PPD in the fetch stage. (b) The two timing scenarios we evaluate.

- 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 applies to large predictors, as shown in both the 21264 and by Jiménez *et al.*.
- Scenario 2: We consider the other extreme also. Here the assumption is that the BTB and the direction predictor need to be accessed every cycle and these accesses take too long to place 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 column multiplexor). The savings here 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 the PPD every cycle. This means we must model the overhead in terms of extra power required for the PPD. If the PPD does not prevent enough BTB/predictor lookups, then introducing a PPD actually increases power dissipation. Fortunately, there are indeed a sufficient number of cache lines that need no BTB/predictor lookups that the PPD is substantially effective.

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.

Figures 10–11 show the effect of a PPD on a 32 K-entry GAs predictor. We chose this configuration in order to be able to include the effects of banking. Figure 10 shows the average reduction in power for the branch predictor and in the overall processor power. We observe a similar trend in Figure 11 for the energy metrics. 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, savings will be proportional for other predictor organizations. It can also be observed that the greater the average distance between branches for a benchmark, the more the savings we get from the PPD. For Scenario 2, the power savings are closely tied to our timing assumptions, and further work is required to understand the potential savings in other precharge and timing scenarios.

4.3. 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.* [16]. The goal of pipeline gating is to prevent wasting energy on mis-speculated contribution. Pipeline gating is relevant because it is natural to expect that the more accurate the branch predictor, the less gating helps save energy: there is less mis-speculation to prevent. Indeed, even with a very poor predictor, we find that the the energy savings are quite small—smaller than previous work using the metric of "extra work" would suggest. Furthermore, under certain conditions, pipeline gating can even harm performance and *increase* energy.

Figure 12 shows the operation of pipeline gating. It uses a confidence estimator [12] 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 instruction that may not commit. The number of low-confidence predictions permitted, N, before gating is engaged is a design parameter. Once the number of in-flight low confidence branches, M, reaches the threshold N, we gate the pipeline, 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 [16] 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" 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, adding 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, but we only include it to see the effect on pipeline gating in the extreme case of poor prediction. The results

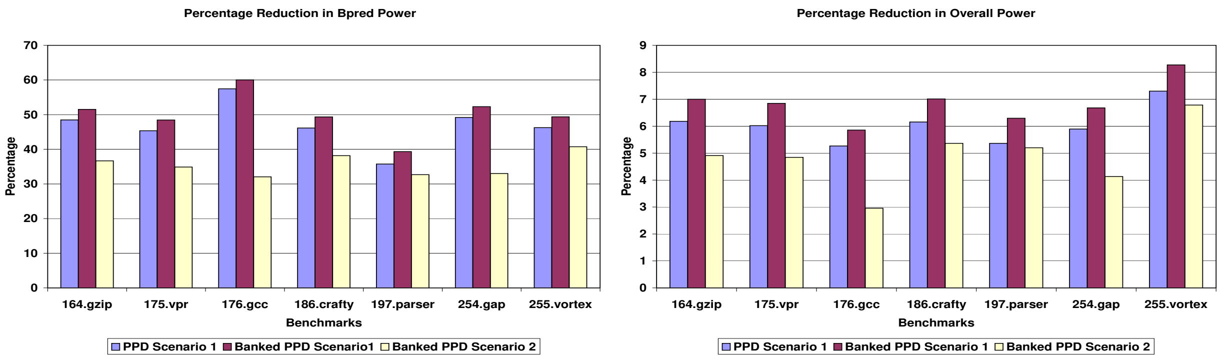

Figure 10. Net savings with a PPD for a 32 K-entry GAs predictor in terms of (a) power in the branch predictor and (b) overall processor power with a PPD. Scenarios 1 and 2 refer to two timing scenarios we model.

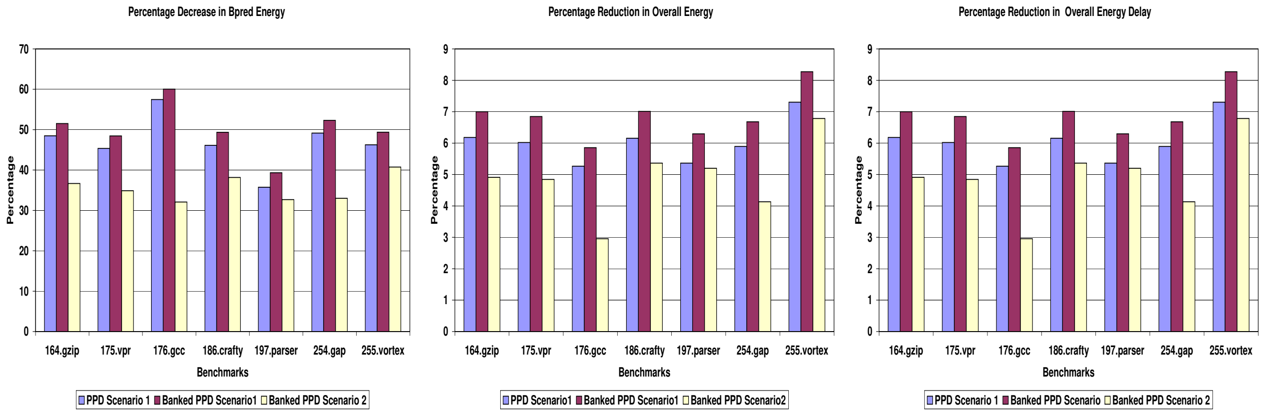

Figure 11. Net savings with a PPD for a 32 K-entry GAs predictor in terms of (a) energy expended in the branch predictor, (b) total energy, and (c) energy-delay.

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 one, hybrid_3 in Figure 13. For each metric, results are normalized to the baseline case with no gating.

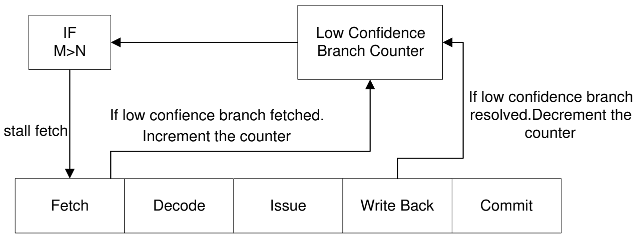

Figure 12. Schematic showing the operation of pipeline gating.

The results show that only the most aggressive pipeline gating, $N = 0$, has substantial effect. For the more relaxed thresholds, the reduction in IPC is small but so is 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 will be squashed immediately when the processor detects the misprediction. Some mis-speculated instructions therefore spend little energy traversing the pipeline, so preventing these instructions' fetch 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 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 esti-

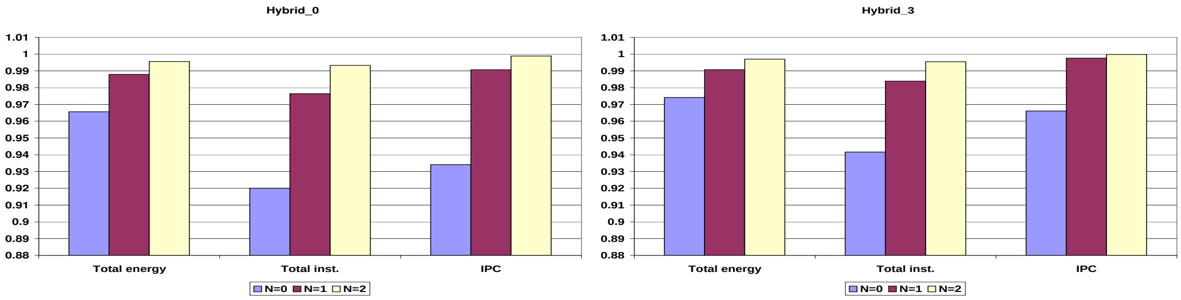

Figure 13. Pipeline gating: overall results of (a) hybrid_0 and (b) hybrid_3.

mators. 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 [12] that are separate from the predictor. This warrants further study.

Although not shown separately, the behavior of the benchmark *vortex* 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 14%, slowing execution time and increasing energy expenditure.

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.

5. Summary and Future Work

The branch predictor structures, which are the size of a small cache, dissipate a non-trivial amount of power—over 10% 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.

In Section 2, we pointed out that array structures (including caches) should model not only the row but also the column decoders. Although the column decoders are not on the critical timing path, they do dissipate a non-trivial amount of power. We also pointed that the choice of how to square-ify a predictor has little effect on its power dissipation .

Section 3 then showed that for all the predictor organizations we studied, total energy consumed by the chip is affected much more strongly by predictor accuracy rather than energy consumed locally by the predictor, because more accurate predictors reduce the running time. We found that for integer programs, large but accurate predictors actually reduce total energy. For example, a large hybrid predictor uses 13% more energy than a bimodal predictor but actually yields a 7% savings in total, chip-wide energy. For floating-point programs, the energy curves are flat across the range of predictor organizations, but this 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 budget 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 4 showed that there are some branch-prediction-related techniques that do save energy without affecting performance. Banking both reduces access time and saves power by accessing only a portion of the total predictor structure. A *prediction probe detector* (PPD) uses predecode bits to prevent BTB and predictor lookups, saving as much as 40–60% in energy expended in the predictor and 5–7% of total energy. Finally, we revisited pipeline gating and showed that it does offer modest energy savings on average, but at the risk of actually increasing energy consumption.

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.

There are a wide range of avenues for future work. This paper has not considered optimizations specific to the BTB’s organization, and this is worth further investigation. The tradeoff reported here between larger but more power-hungry structures and improved performance and energy dissipation are likely to apply to other on-chip structures like the caches and TLBs, and this tradeoff is important to understand. Finally, this paper only considered a few specific timing and precharge scenarios for the structures in the fetch stage. A more thorough characterization of this design space is needed to understand the range of benefits that a PPD might provide.

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 many helpful suggestions on how to improve the paper.

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. Next cache line and set prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 287–96, June 1995.

[6] 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.

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

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

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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).

[22] 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.

[23] 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.

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

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

[26] 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.

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

[28] 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.

[29] 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.