Power-Aware Branch Prediction: Characterization and Design

Masters Project Report, Dept. of Computer Science, UVA, Dec. 2002

Dharmesh Parikh

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 35% on average and overall processor power and energy dissipation by 3–5%. We also investigated the role of a branch predictor in a Simultaneous Multithreaded Processor (SMT). Power dissipation in an SMT processor was also explored. Finally we developed a tool for measuring leakage energy in cache-like structures in a microprocessor.*

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 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 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. Yet the branch predictor, including the BTB, is the size of a small cache and dissipates a non-trivial amount of power—10% of the total processor's power dissipation—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 [27] and [28].

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 spent in the predictor comes at the expense of predictor accuracy and hence program performance, this localized reduction may actually increase the 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 reduce precharge costs, and banking need not have any effect on prediction accuracy. Eliminating unnecessary 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) 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 updates here.

A. Contributions

This work extends the Wattch 1.02 [8] 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 40%.

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 [47], GAs/gshare [36], [56], PAs [56], and hybrid [36]. We focus mostly on the branch direction 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, as techniques like the PPD affect both.

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.

B. Related Work

Some prior research has characterized power in other parts of the processor. Pipeline gating was presented by Manne *et al.* [34] as an efficient technique to prevent mis-speculated instructions from entering the pipeline and wasting energy while imposing only a negligible performance loss. Albonesi [3] 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 [21] look at sub-banking and other organizational techniques for reducing energy dissipation in the cache. Zhu and Zhang [58] describe a low-power associative cache mode that performs tag match and data access sequentially, and describe 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.* [32] and Tang *et al.* [50] 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.* [20] 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 [5] 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, but this work does 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 [8] version 1.02 power-performance simulator. Wattch augments the SimpleScalar [9] 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 [31], 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 [44], [45], 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 IV.

Unless stated otherwise, this paper uses the baseline configuration as shown in Table I, which resembles as much as possible the configuration of an Alpha 21264 [31]. The most important difference for this paper is that in the 21264 there is no separate BTB, because

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 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

the I-cache has an integrated next-line predictor [11]. 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.

B. Benchmarks

We evaluate the programs from the SPECcpu2000 [49] benchmark 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 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.

C. 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 [22] 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).

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%

D. Branch Predictors Studied

The bimodal predictor [47] 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 [52]; 4 K-entry is the same size as that in the Alpha 21064 [16] and is at the point of diminishing returns for bimodal predictors, although the 21164 used an 8 K-entry predictor [17]. The gshare predictor [36], shown in Figure 1a, is a variation on the two-level GAg/GAs global-history predictor [38], [56].

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 [46]; 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 [48] and the shorter global history string gives 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 [46].

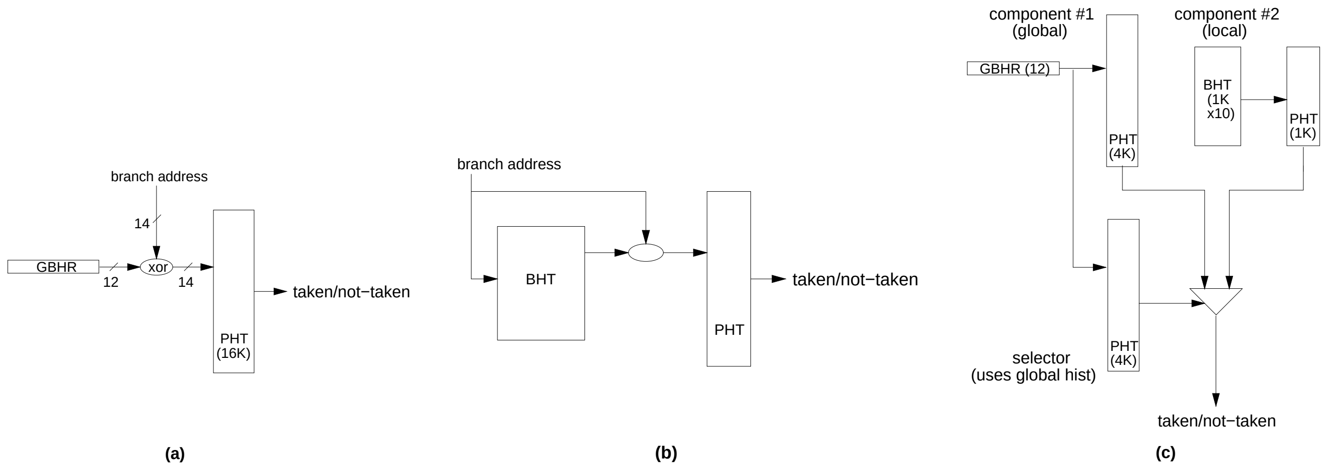


Fig. 1. (a) A gshare global-history branch predictor like that in the Sun UltraSPARC-III. (b) A PAs local-history predictor. (c) A hybrid predictor like that in the Alpha 21264.

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 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.* in [46].

Because most programs have some branches that perform better with global history and others that perform better with local history, a hybrid predictor [13], [36], 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 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 [31] 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.

Hybrid_2, 3, and 4 are based on configurations found to perform well by Skadron *et al.* in [46]. A brief summary of all the branch predictors studied is given in Table III

TABLE III

Summary of branch predictors studied .

	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

III. Performance-Power Tradeoffs Related to Branch Prediction

A. 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 2a presents the average branch predictor direction accuracy for integer benchmarks, and Figure 2b 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.

Figure 3 gives the energy and energy-delay characteristics. Together, Figure 3a and Figure 3b 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 7% less chip-wide energy than bimodal-4K 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 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

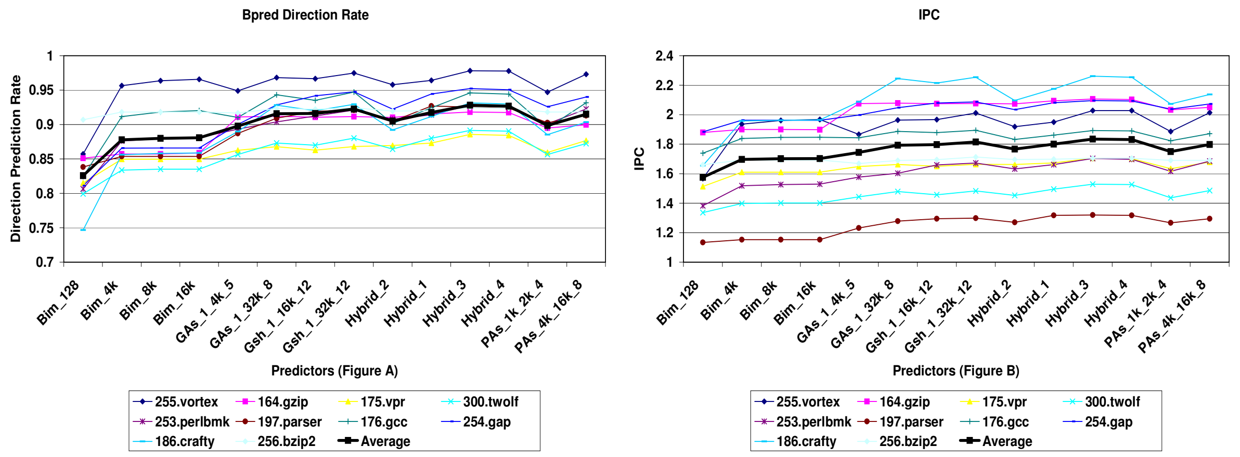


Fig. 2. (a) Direction-prediction accuracy and (b) IPC for SPECint2000 for various predictor organizations. For each predictor type, the predictors are arranged in order of increasing size along the X-axis. The arithmetic mean is the dark curve in each of the graphs.

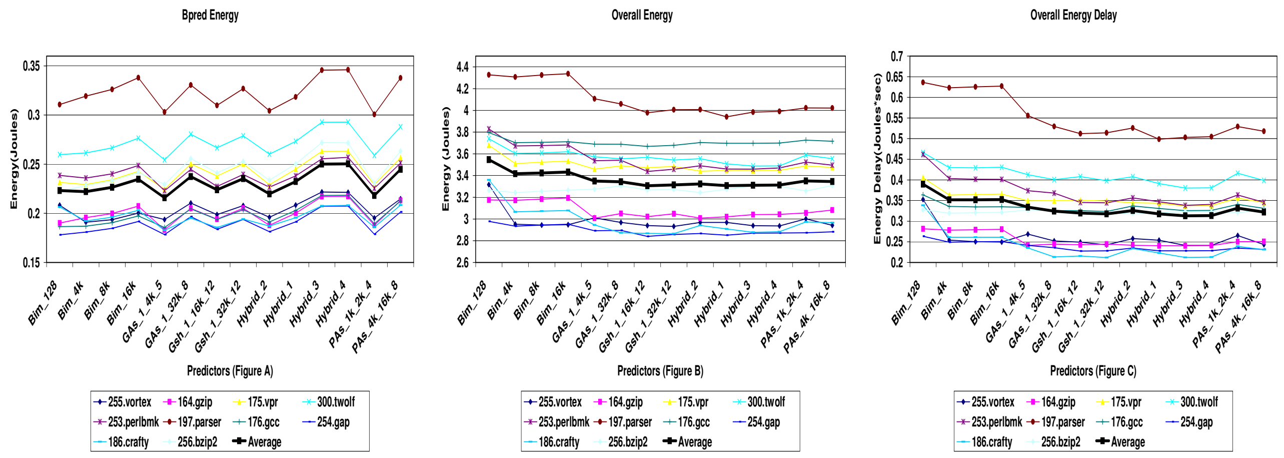


Fig. 3. Energy expended in (a) the branch predictor and (b) the entire processor, and (c) energy-delay for the entire processor for SPECint2000.

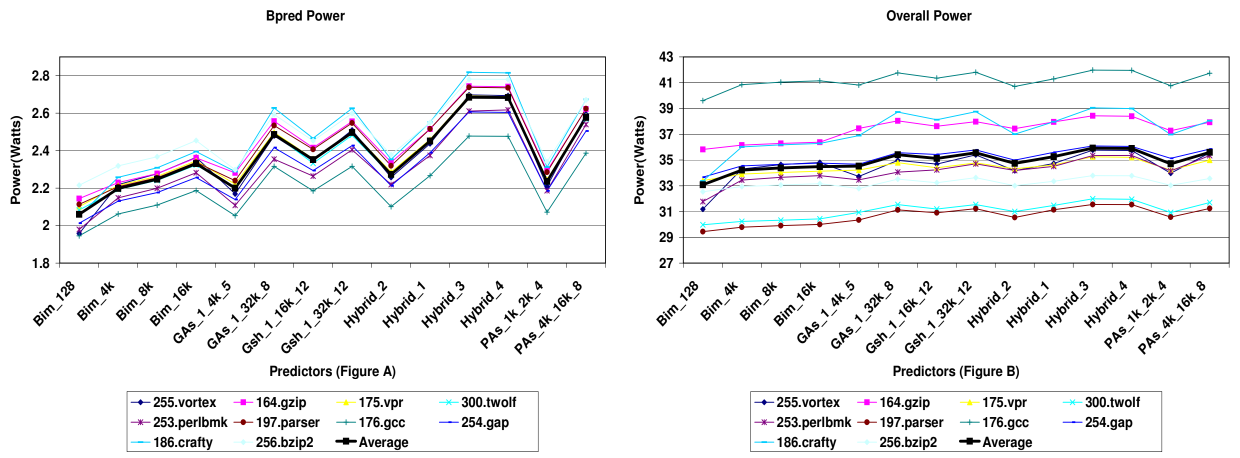
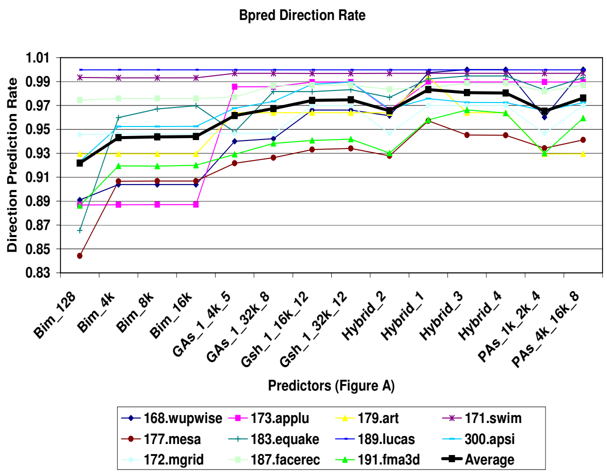


Fig. 4. Power dissipation in (a) the branch predictor and (b) the entire processor for SPECint2000.



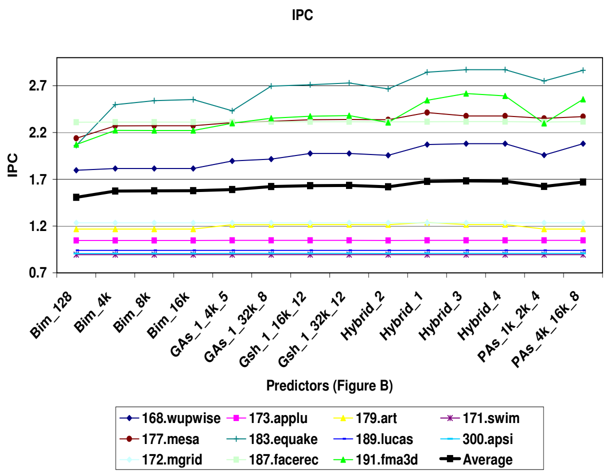
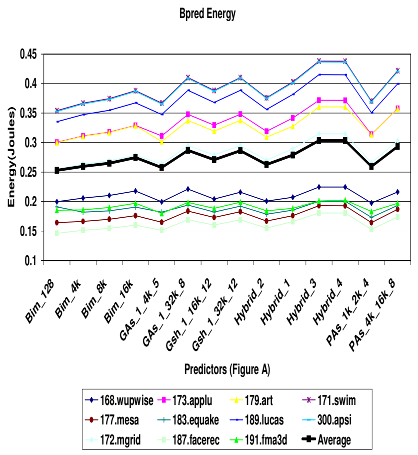
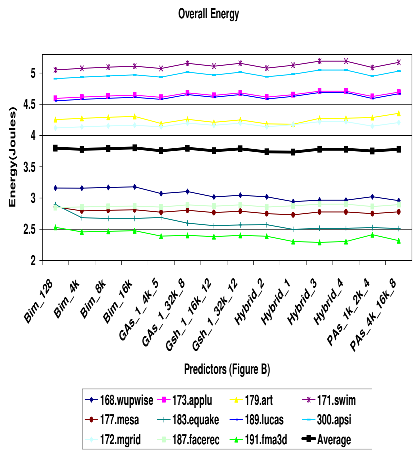


Fig. 5. (a) Direction-prediction accuracy and (b) IPC for SPECfp2000.





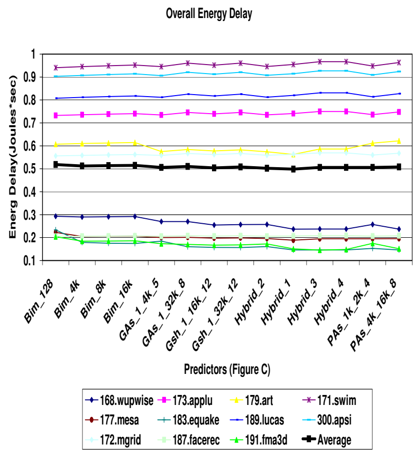
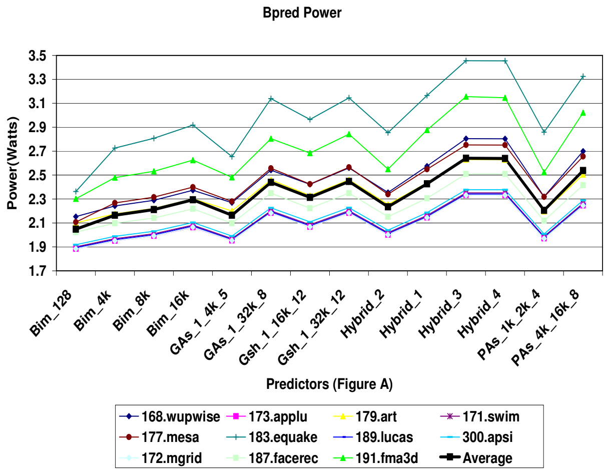


Fig. 6. Energy expended in (a) the branch predictor and (b) the entire processor, and (c) the energy-delay for the entire processor for SPECfp2000.



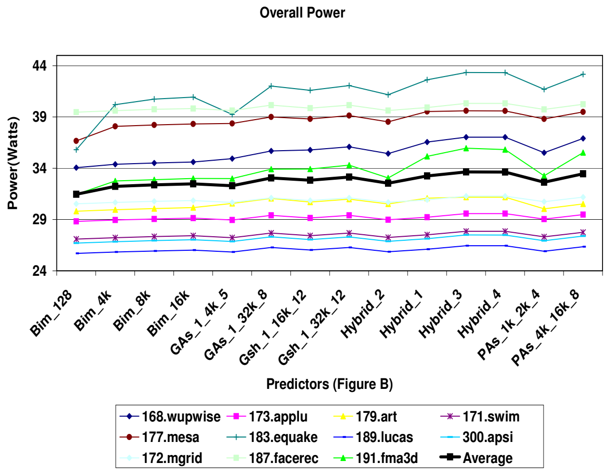


Fig. 7. Power dissipated in (a) the predictor and (b) the entire processor for SPECfp2000.

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 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 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 *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!

B. Base Simulations for Floating Point Benchmarks

Figures 5–7 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. Potential Gains from Improved Accuracy

Just to illustrate how much leverage branch-prediction accuracy has on both performance and energy, Figure 8 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.

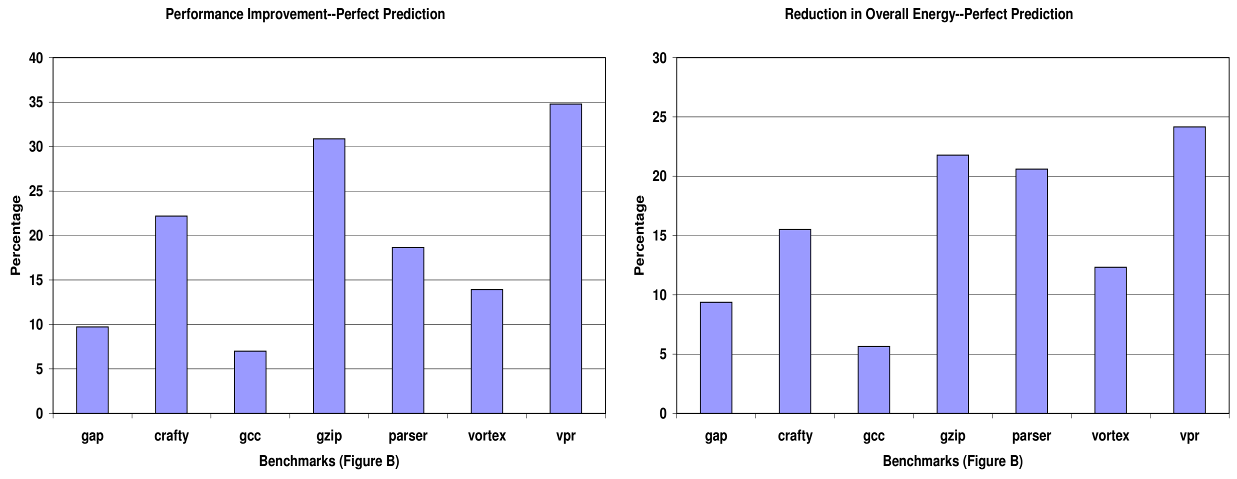


Fig. 8. (a) Performance improvement from perfect prediction and (b) Percentage reduction in overall energy using perfect prediction compared to a 32k-entry global-history (GAs) predictor.

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 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.

A. Banking

As shown by Jiménez, Keckler, and Lin [29], 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 again used help from the modified Cacti [55] 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 us power spent in the branch predictor, as shown in Figure 9. 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 resulting number of banks for different branch predictor sizes is given in Table IV.

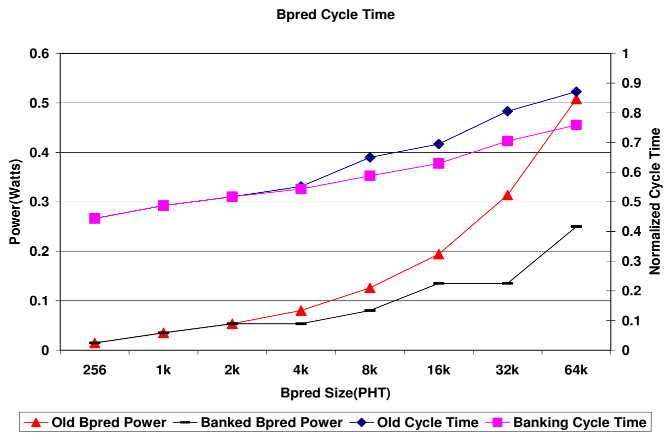


Fig. 9. Cycle time for a banked predictor.

Figures 10 and 11 show the difference between the base simulations and the banking figures. It can be observed that the largest decrease in predictor power comes for larger

TABLE IV

Number of banks.

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

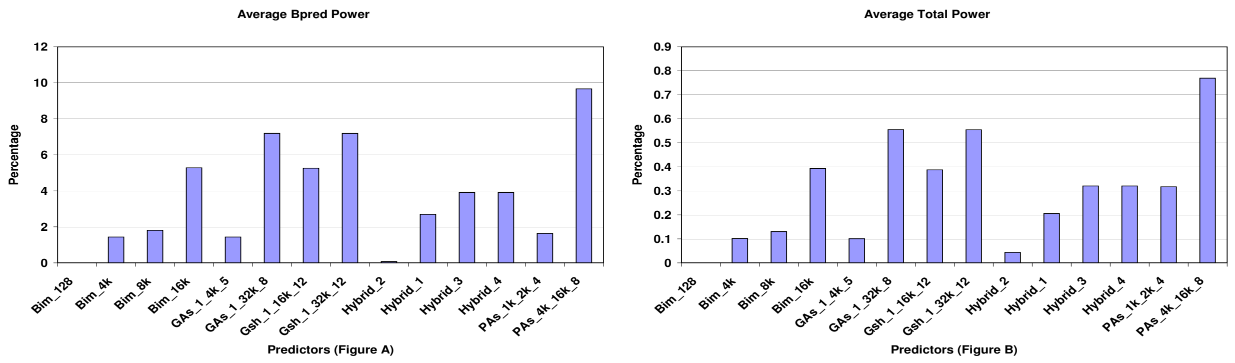


Fig. 10. Banking results: (a) Percentage reduction in branch-predictor energy (b) and overall power.

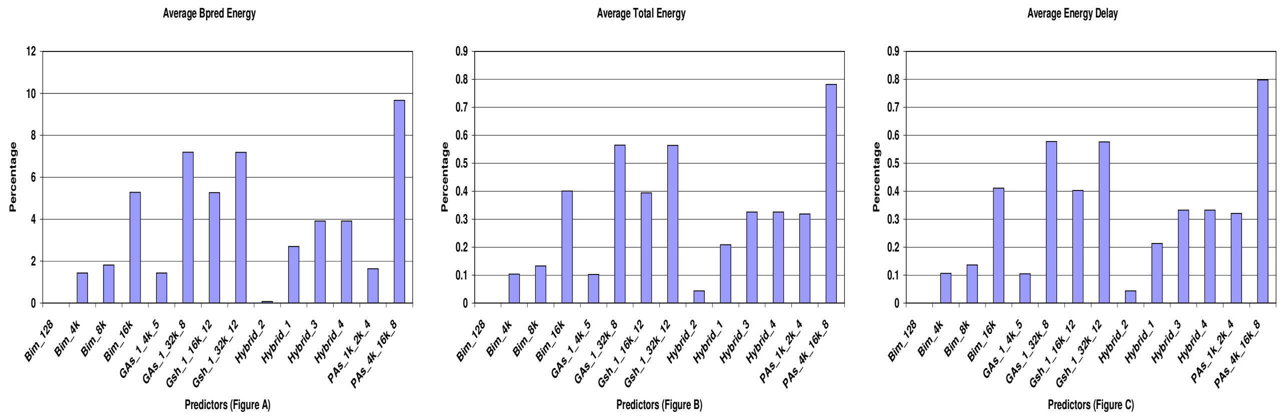


Fig. 11. Banking Results: (a) Percentage reduction in branch-predictor energy, (b) overall energy, and (c) energy-delay.

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

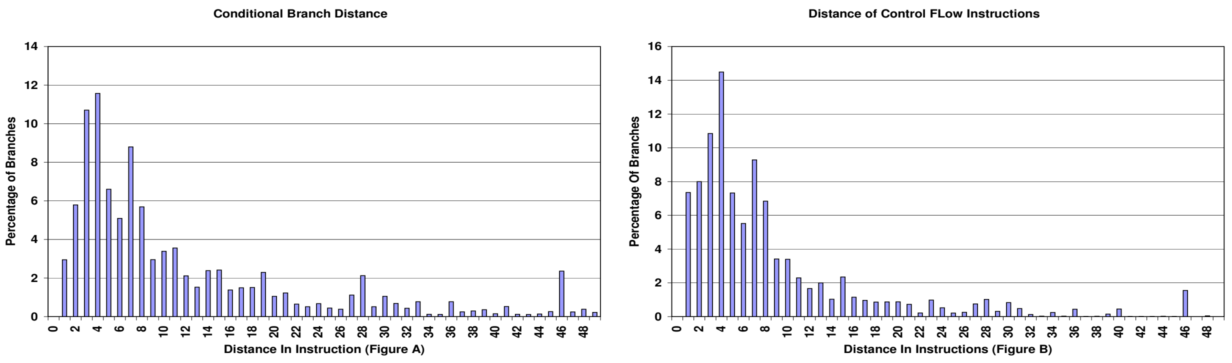


Fig. 12. (a) Average distance (in terms of instructions) between conditional branches. (b) Average distance between control-flow instructions (conditional branches plus unconditional jumps).

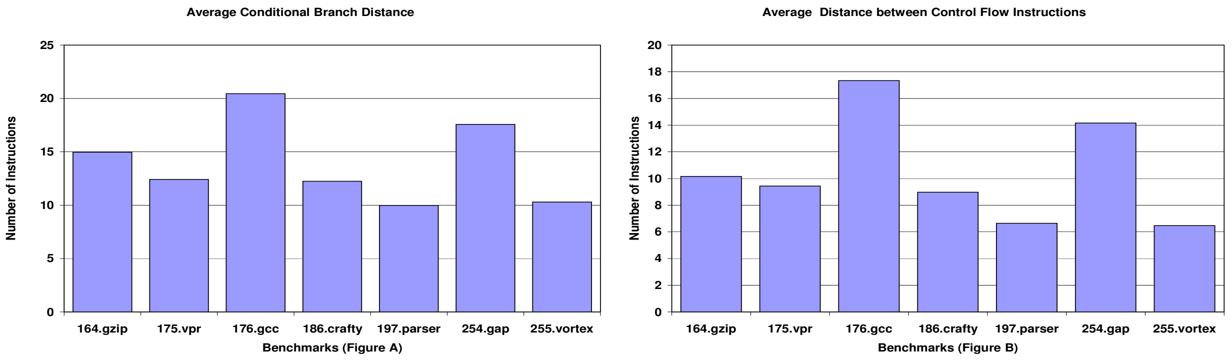


Fig. 13. (a) Average distance (in terms of instructions) between conditional branches for different benchmarks. (b) Average distance between control-flow instructions (conditional branches plus unconditional jumps) for different benchmarks.

and energy by about 1%.

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 12 instructions. Figures 12 and 13 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 [29]. 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 might not perform very well.

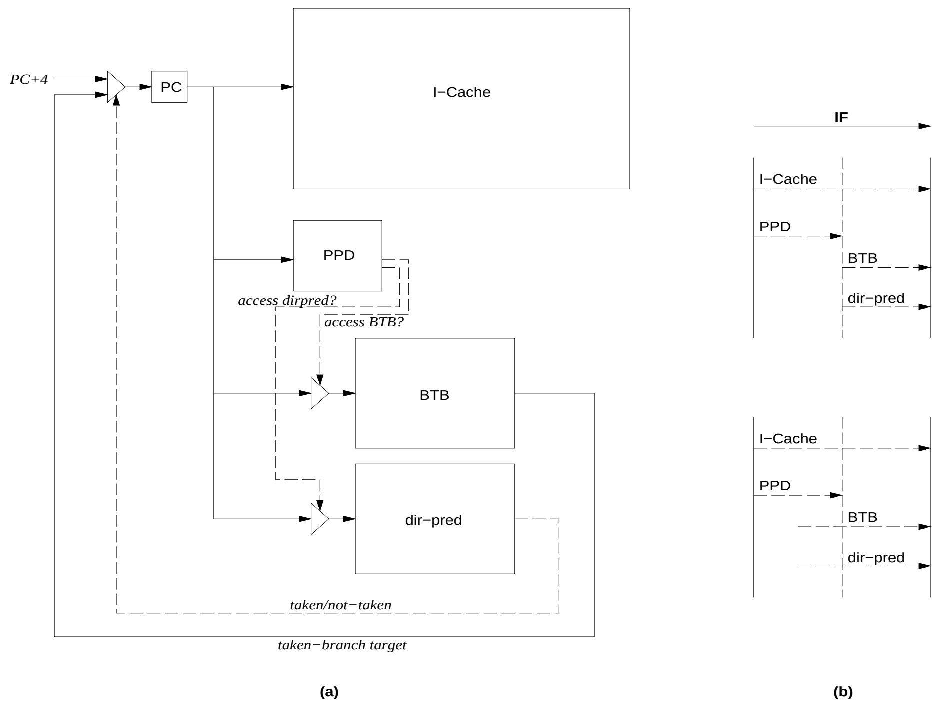


Fig. 14. (a) A schematic of the PPD in the fetch stage. (b) The two timing scenarios we evaluate.

The fact that many cache lines have no control flow instructions suggests that we should identify dynamically 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 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 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 [12]. The main difference is that 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/icache line is the most critical piece of information was also pointed out in [54]. 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 14a. 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* so as to guarantee that if a branch is present in any way, then branch prediction is done. This scheme is conservative because many times the way that matches and is fetched might not have a branch.

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 scenario is beyond the scope of this paper, so here we explore two extremes, shown in Figure 14b.

- 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.*. Due to the small size of PPD (4 Kbits) it can be seen from Figure 9 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 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 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 substantially effective. As explained earlier due to the "conservative" nature of the PPD, our scheme works best in 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.

Figures 15–16 show the effect of a PPD on a 32 K-entry GAs predictor for a direct mapped cache of the same size. We chose this configuration in order to be able to include the effects of banking. Figure 15 shows the average reduction in power for the branch predictor and in the overall processor power. We observe a similar trend in Figure 16 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. Figure 17 shows the average percentage reduction in power and energy for increasing associativity of I-cache (same size 64 K). It is clearly observed that the benefits of PPD decreases as the associativity increase. For eight-way I-cache there is practically no benefit at all.

C. Highly Biased Branches

The PPD was also extended to recognize "unchanging" branches (which are always taken or always not taken) [39]. We modified SimpleScalar's *sim-bpred* to identify different types of branches. Our analyzer goes through a program and stores a 2-tuple {branch

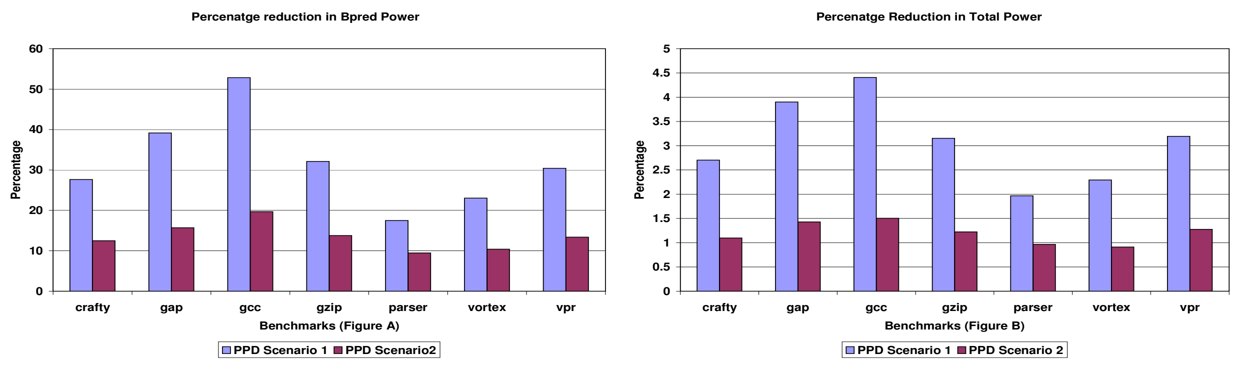


Fig. 15. Net savings with a PPD for a 32 K-entry GAs predictor and direct mapped cache in terms of (a) power in the branch predictor and (b) overall processor power with a PPD. Scenarios 1 and 2 refer to two timing scenarios we model.

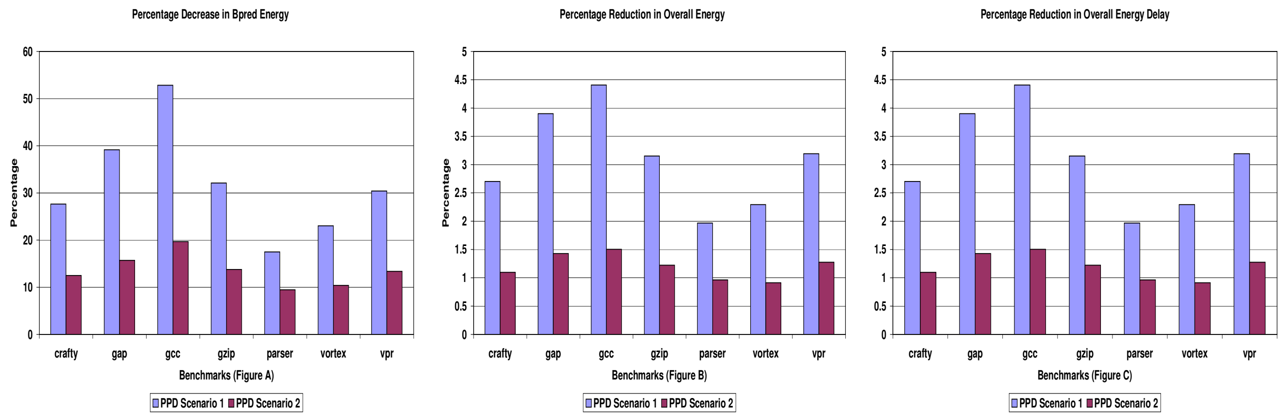


Fig. 16. Net savings with a PPD for a 32 K-entry GAs predictor and direct mapped cache in terms of (a) energy expended in the branch predictor, (b) total energy, and (c) energy-delay.

address, branch type} value in a file. Then Wattch reads this profile and calculates power for each type of branch. For "unchanging branches", direction look-up is not done. For correlation of power versus branch type, four 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 fine, as any incorrect hints will be detected in the later stages of pipeline. The plots presented take all these effects into account. From Figure 18a 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 is not in the above proportion, as seen

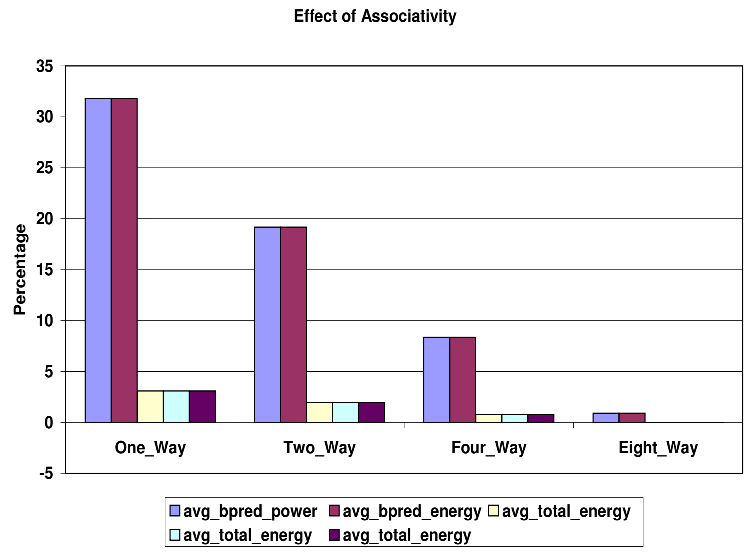


Fig. 17. Net savings with a PPD for different associativity for I-cache

in Figure 18b. The reason is that "changing" branches are executed more times than "unchanging" branches. The percentage of energy consumed by unchanging branches can range from trivial (164.gzip) to substantial (175.vpr). From the figures provided it can be seen that unchanging branches consume from less than 1% up to 20% of energy. Please note that this is the energy that can be saved on top of the savings provided by PPD. 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. The assumption we make is that the instructions can be annotated in the binary to specify that a particular instruction is a highly biased branch.

V. Summary and Future Work

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

5its power dissipation but does affect access time.

In Section III, we showed that for all the predictor organizations we studied, total energy

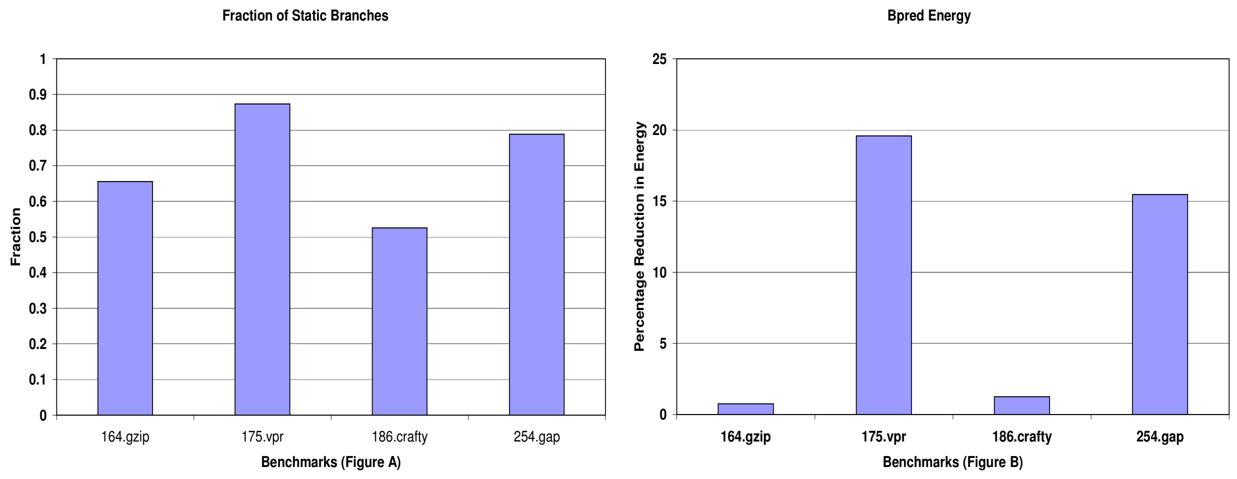


Fig. 18. (a) Static Profile (not dynamic execution) of branches and (b) Percentage of branch predictor energy consumed by "unchanging" branches.

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 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 IV 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 pre-decode bits to prevent BTB and predictor lookups, saving as much as 30–50% in energy expended in the predictor and 3–5% of total energy.

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; and Zhigang Hu for help with the EIO traces.

Appendix

I. Simultaneous Multithreaded Processor

A. SMT

Simultaneous Multi-threaded Processors (SMT) [51] is a processor that exploits both thread-level parallelism and instruction level parallelism. SMT combines hardware features of wide-issue superscalars and multi-threaded processors. From superscalars, it inherits the ability to issue multiple instructions each cycle; and like multi-threaded processors it contains hardware state for several programs (or threads). SMT adds minimal hardware complexity to conventional superscalars. The result is a processor that can issue multiple instructions from multiple threads each cycle, achieving better performance for a variety of workloads. Single-threaded program that must execute alone will have all machine resources available to it and will maintain roughly the same level of performance as when executing on a single-threaded, wide issue processor.

The disadvantages are that there is a slight increase in the chip area, the pipeline length increases (due to the increased access time to a very large register file), and resource contention among shared hardware structures such as caches, TLBs, issue queues, and the branch prediction tables might increase. Inter-thread contention on L1 caches can increase by 68% and branch mispredictions by 50%, as the number of threads vary from one to eight [18].

The fetch unit is one of the major bottlenecks in an SMT processor. The fetch unit can fetch from multiple threads at once, increasing the utilization of the fetch bandwidth, and it can be selective about which thread or threads to fetch from. Because not all paths provide equally useful instructions in a particular cycle, an SMT processor can benefit by fetching from thread(s) that will provide the best instructions.

Several Fetch Policies have been explored for an SMT processor. The most effective fetch policies investigated are:

• ICOUNT [51]: Here highest priority is given to those threads that have the fewest instructions in the decode stage, the rename stage, and the instruction queues.

• Fetch Prioritizing [33]: This policy sets up fetch priority for each thread based on the number of unresolved low-confidence branches from the thread.

• Fetch Prioritizing with gating [33]: This includes the previous fetch policy and prevents fetching from a thread once it has a stipulated number of outstanding low-confidence branches.

B. Design of a Branch Predictor for SMT.

Not much work has been done explicitly in this area. Hily and Seznec [26] studied branch prediction and multi-threading. Their study basically showed that the Return Address Stack (RAS) should be per-thread and that the size of the branch prediction tables should be in proportion to the number of threads. Gummaraju and Franklin [23] investigated branch prediction in a Single-Program Multi-Threaded Processor(SPMT). Their main points are:

• Private Predictors for each thread: In a Single Programmed Multiple Thread environment (SPMT) if the thread size is small, the performance is likely to be poor because of insufficient history or discontinuous history. Generally if the thread size is larger than per-thread predictor is better in terms of performance.But it will come at at an overhead of chip area.

• Shared Predictor: Recorded history of Pas is affected only when there are multiple instances of the same branches in multiple simultaneous active threads. Recorded history

of Gshare is affected whenever multiple branches are present in multiple simultaneously active threads. It is better to bank a large global predictor like Gshare and let each thread believe it has it own private predictor.
They then explore some way to increase the branch predictor accuracy in case of smaller threads (when neither a per-thread predictor or a shared predictor are superior).

The current state of the art for branch predictors in an SMT processor is to have the RAS duplicated. The global history register is also per-thread. The large global history array are shared amongst the threads with entries that are tagged with a thread id [35].

II. Simultaneous Multithreaded Processor: A Power Perspective

A. SMT and Power

Power is becoming a major constraint in the design of microprocessors, as shown earlier. SMT processors are attractive in the context of low-power for many of the same reasons that make them high-throughput. First, it supplies extra parallelism via multiple threads, allowing the processor to rely much less heavily on speculation; thus it wastes fewer resources on speculation. Second, it provides more and even parallelism over time running multiple threads, wastingless power on under-utilized execution resources. It was shown by [42] that an SMT processor utilizes up to 22% less energy per instruction than a single-threaded processor.

We improved upon an SMT simulator built upon Wattch. The SMT-Power simulator infrastructure is in its initial stages and does need some more work. The results that we got are more or less consistent with [42]. Figure 19 (A) shows how the energy per useful executed instruction (EUI) goes down for a multi-threaded workload, while the performance (IPC) goes up. The increase in power occurs because of the overall increase in utilization and throughput of the processor. Here all results are normalized to a baseline (the lowest single-thread value). Figure 19 (B) shows the results for a four-threaded workload compared to individual runs of the workload in a single-threaded mode. The length of the simulation for each benchmark is the same within the multi-threaded run and the single-threaded run. The results here are not normalized.

It is thus obvious that SMT is an attractive architecture when energy and power are

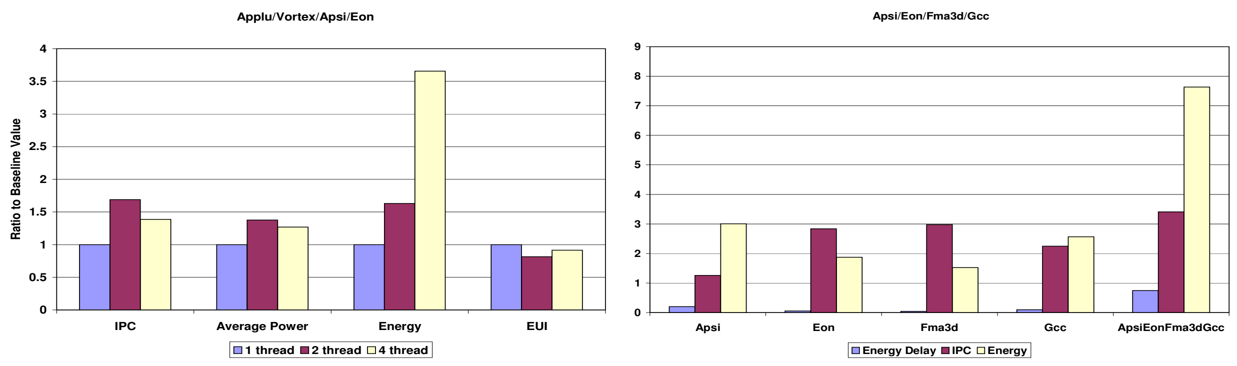


Fig. 19. Relative Performance, power and energy savings (a) As the number of threads vary. Here the normalization is done with the lowest values of a single thread. Here EUI is Energy per Useful Instruction. (b) For a multi-threaded run compared with the single-threaded run.

constraints. It can use significantly less energy per instruction while increasing the overall performance. SMT does speed up the overall performance of a mutithreaded workload, while some of the individual threads might be degraded in performance.

III. HotLeakage: An Accurate, Temperature-Aware, and Computationally-Efficient Leakage Model for Architects

A. HotLeakage

Power is rapidly become a design constraint not only in the domain of mobile devices but also in high performance processors. Dynamic power is caused by switching activity in CMOS circuits. It is the major source of total power dissipation in today's process generation. However static power, which is due to leakage current in the quiescent state of circuits, is gaining more importance. Technology scaling is increasing both the absolute and relative contribution of static power dissipation. Borkar [7] estimates that in the next several processor generations, leakage may constitute as much as 50% of total power dissipation (Figure 20 taken from [10]).

Recently, a great deal of research work in the architecture community has focused on reducing leakage power in the caches [19], [24], [25], [30], [40], [53], [57], branch predictor [27], register file [4], and issue queues [14], [15], [2], [1]. Leakage control at the architecture level is attractive, because architectural techniques can control large groups of circuits (*e.g.* cache lines, banks, or the entire cache) at once. Yet most of these studies use only

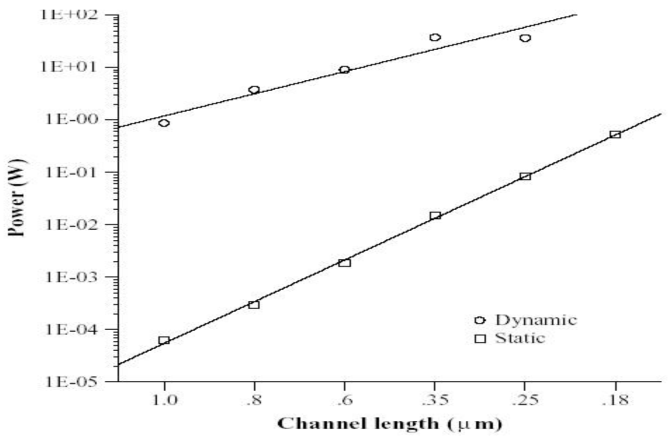


Fig. 20. Trends in dynamic and static power dissipation.

abstract models of leakage that do not fully account for all effects that may impact leakage, like supply voltage and temperature. Unlike for dynamic power, where widely-available simulators like Wattch [8] have enabled a widespread body of research, where is no widely available model for leakage power. Although Butts and Sohi [10] propose a simple model for use at the architecture-simulation level of abstraction, no corresponding software is available. Most importantly, their model cannot easily model leakage when temperature, supply voltage, or threshold voltage vary dynamically: a new "normalized leakage" and k_{design} must be calculated for every possible value. This is inconvenient although feasible for leakage-control schemes like drowsy [19] cache that uses two supply voltages, but intractable for any leakage studies that account for dynamically varying temperature or involve dynamic voltage scaling.

We developed a software model of leakage—based on BSIM3 [6] technology data—that is already publicly available, computationally very simple, can easily be integrated into popular power-performance simulators like Wattch, and can easily be extended to accommodate other technology models. It extends the Butts-Sohi model and corrects several important sources of inaccuracy. We call our model HotLeakage, because it includes the exponential effects of temperature on leakage. Temperature effects are important, because leakage current depends exponentially on temperature, and future operating temperatures may exceed 100$\circ$ C [43]. Currently the model is implemented on top of Wattch. The tool is completely modular and easy to understand. All the architectural parameters can be specified by the user. A user can easily add the model for a new structure into the tool. It will have a complete user manual when publicly released.

HotLeakage has circuit-level accuracy because the parameters are derived from transistor-level simulation (Cadence tools). Yet like the Butts and Sohi model, simplicity is maintained by deriving the necessary circuit-level model for individual cells, like memory cells or decoder circuits, and then taking advantage of the regularity of major structures to develop abstract models that can be expressed in simple formulas similar to the Butts-Sohi model. All necessary components of this formula are encapsulated in lookup tables.

B. Leakage Reduction Techniques in Caches

We also demonstrate the importance of the increased accuracy that HotLeakage provides, by comparing several popular leakage-control techniques (fine-grained gated-V$_{dd}$ [30], dual-V_t [41], drowsy cache [19], and reverse-body-bias [37]) and showing some non-obvious tradeoffs. Recently two techniques have been suggested to reduce leakage in caches .

• GatedVss: The gated-Vdd structure was introduced in [40]. This technique reduces the leakage power by using a high threshold transistor to turn off the power to the cell of a cache when the cell is set to low-power mode. This high threshold transistor drastically reduces the leakage of the circuit because of the exponential dependence of leakage on the threshold voltages. While this technique is very efficient in saving leakage there is the disadvantage that the cell looses its state (information). Thus this is a state loosing technique. This means that there will be some performance penalty.This technique was used in [30] to shut down lines in a cache to save leakage.

• Dynamic Vdd Scaling (DVS): It was proposed in [19] to use DVS to reduce the leakage power of caches. By scaling the voltage of the cell to approximately 1.5 times the threshold voltage the state of the cell can be maintained. Thus this is a state preserving technique. Since both leakage current and the voltage is reduced there is a significant reduction in leakage (though less than GatedVss).

Our model can be used to find the tradeoffs between such leakage saving techniques by varying all the parameters. Figure 21 (A) shows the tradeoff of using the above mentioned two techniques at a temperature of 110 Degree Fahrenheit.The cache that is controlled is a level one data cache. The cache is two-way associative with 512 sets and block size of 64 bytes. The latency of the level two data cache is 11 cycles. Figure 21 (B) shows the same tradeoffs at 85 Degree. These figures show that a designer can carry out a detailed study using our model by varying different parameters and can choose a technique based on his/her constraints.

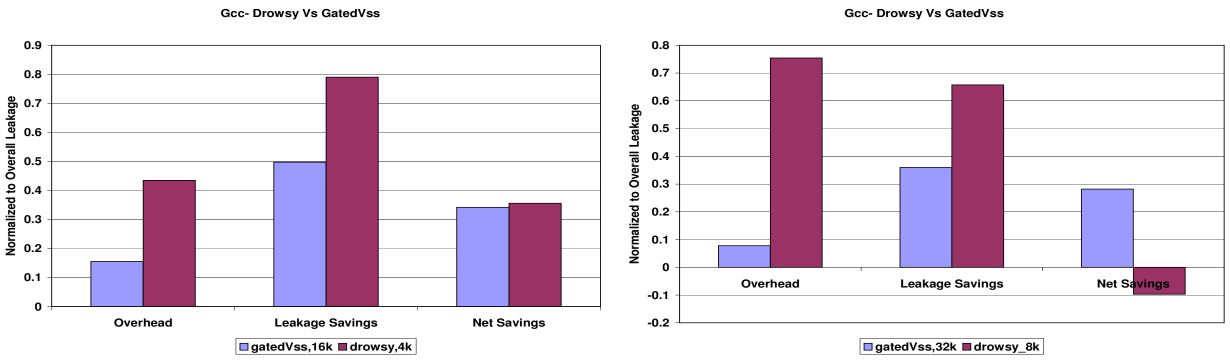


Fig. 21. Tradeoffs in using Drowsy Vs. GatedVss to save leakage in caches (a) For 110 Degree and (b) for 85 Degree Fahrenheit. Here all the values are normalized against the overall leakage energy dissipated in the cache for the whole length of the simulation. The best decay interval for both techniques are chosen here.

We hope that this new leakage model, its public availability, and the results we get will facilitate greater research on techniques for controlling leakage power at the architecture level.

References

[1] D. Brooks P. Bose P. W. Cook D. H. Albonesi A. Buyuktosunoglu, S. E. Schuster. An adaptive issue queue for reduced power at high performance. In *Workshop on Power-Aware Computer Systems*, Nov. 2000.

[2] P. Bose P. W. Cook A. Buyuktosunoglu, D. H. Albonesi and S. E. Schuster. Tradeoffs in power-efficient issue queue design. Aug. 2002.

[3] 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.

[4] A. Alvandpour, R. Krishnamurthy, K. Soumyanath, and S. Borkar. A low-leakage dynamic multi-ported register file in 0.13um CMOS. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 68–71, Aug. 2001.

[5] 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.

[6] U. C. Berkeley. BSIM3v3.1 SPICE MOS device models, 1997. `http://www-device.EECS.Berkeley.EDU/ bsim3/`.

[7] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, pages 23–29, Jul.–Aug. 1999.

[8] 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.

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

[10] J. A. Butts and G. S. Sohi. A static power model for architects. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 191–201, Dec. 2000.

[11] 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.

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

[13] 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.

[14] A. Gonzalez D. Folegnani. Energy-effective issue logic. June. 2001.

[15] K. Ghose D. Ponomarev, G. Kucuk. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the 34rth Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2001.

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

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

[18] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, pages 12–19, Sep. 1997.

[19] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 147–57, May 2002.

[20] 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.

[21] 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.

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

[23] J. Gummaraju and M. Franklin. Branch prediction in multi-threaded processors. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, pages 179–89, Oct. 2000.

[24] H. Hanson et al. Static energy reduction techniques for microprocessor caches. In *Proceedings of the 2001 International Conference on Computer Design*, pages 276–83, Sept. 2001.

[25] S. Heo, K. Barr, M. Hampton, and K. Asanović. Dynamic fine-grain leakage reduction using leakage-biased bitlines. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 137–47, May 2002.

[26] S. Hily and A. Seznec. Branch prediction and simultaneous multithreading. In *Proceedings of the 1996 International Conference on Parallel Architectures and Compilation Techniques*, pages 169–73, Oct. 1996.

[27] 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*, Aug. 2002. To appear.

[28] Z. Hu, K. Skadron, P. Juang, D. W. Clark, and M. Martonosi. Applying decay strategies to branch predictors for leakage energy savings. Technical Report CS-2001-24, University of Virginia Department of Computer Science, Oct. 2001.

[29] 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.

[30] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[31] 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.

[32] 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.

[33] K. Luo, M. Franklin, S. Mukherjee, and A. Seznec. Boosting smt performance by speculation control. In *Proceedings of the 5th Annual International Parallel and Distributed Processing Symposium*, March. 2001.

[34] 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.

[35] D. T. Marr, F. Binns, D. L. Hill, G. Hilton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, pages 4–16, Feb. 14, 2002.

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

[37] K. Nii et al. A low power SRAM using auto-backgate-controlled MT-CMOS. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 293–98, Aug. 1998.

[38] 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.

[39] 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.

[40] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, pages 90–95, July 2000.

[41] K. Roy. Leakage power reduction in low-voltage CMOS designs. In *Proceedings of the International Conference on Electronics, Circuits, and Systems*, pages 167–73, 1998.

[42] J. Seng, D. Tullsen, and G. Cai. Power-sensitive mutli-threaded architecture. In *Proceedings of the 2000 International Conference on Computer Design*, Sept. 2000.

[43] SIA. *International Technology Roadmap for Semiconductors*, 2001.

[44] 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.

[45] 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).

[46] 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.

[47] 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.

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

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

[50] 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.

[51] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May. 1996.

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

[53] S. Velusamy, K. Sankaranarayanan, D. Parikh, T. Abdelzaher, and K. Skadron. Adaptive cache decay using formal feedback control. In *Proceedings of the 2002 Workshop on Memory Performance Issues*, May 2002.

[54] Steven Wallace and Nager Bagherzadeh. Multiple branch and block prediction. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 94–103, February 1997.

[55] 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.

[56] 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.

[57] H. Zhou, M. Toburen, E. Rotenberg, and T. Conte. Adaptive mode control: A static-power-efficient cache design. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

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