Applying Decay Strategies to Branch Predictors for Leakage Energy Savings

– Univ. of Virginia Tech. Report CS-2001-24 –

Zhigang Hu, Philo Juang
Department of Electrical Engineering
Princeton University
hzg,pjuang@ee.princeton.edu

Kevin Skadron
Department of Computer Science
University of Virginia
skadron@cs.virginia.edu

Doug Clark
Department of Computer Science
Princeton University
doug@cs.princeton.edu

Margaret Martonosi
Department of Electrical Engineering
Princeton University
mrm@ee.princeton.edu

Abstract

This paper shows that substantial reductions in leakage energy can be obtained by deactivating groups of branch-predictor entries if they lie idle for a sufficiently long time. *Decay* techniques, first introduced by Kaxiras *et al*. for caches, work by tracking accesses to cache lines and turning off power to those that lie idle for a sufficiently long period of time (the *decay interval*). Once deactivated, these lines essentially draw no leakage current. The key trick is in identifying opportunities where an item can be turned off without incurring significant performance or power cost.

Branch predictors are, like caches, large array structures with significant leakage; as such, it is natural to consider applying decay techniques to them as well. Applying decay techniques to branch predictors is, however, not straightforward. The overhead for applying decay to individual counters in the predictor is prohibitive, so decay must be applied to groups of predictor entries. The most natural grouping is a row in the square data array used to implement the branch predictor, but then decay will only be successful if entire rows lie idle for sufficiently long periods of time. This paper shows that branch predictors do exhibit sufficient spatial and temporal locality to make decay effective for bimodal, gshare, and hybrid predictors, as well as the branch target buffer. In particular, decay is quite effective when applied intelligently to hybrid predictors, which use two predictors in parallel and are among the most accurate predictor organizations. Hybrid predictors are also especially amenable to decay, because inactive entries in one component can be left inactive if the other component is able to provide a prediction. Overall, this paper demonstrates that decay techniques apply more broadly than just to caches, but that careful policy and implementation make the difference between success and failure in building decay-based branch predictors.

1 Introduction

Power dissipation is an increasingly important problem in high-performance CPUs. As high-end processors increasingly stress both current-delivery and thermal limits in CMOS integrated circuits, methods for reducing both maximum and average power dissipation become crucial, not only for mobile computing but for all classes of processors. While circuit-level techniques remain an important mainstay in managing power problems, architecture-level approaches to power management have become increasingly popular and successful in complementing the circuit-level techniques.

Until quite recently, most architecture-level approaches for CPU power efficiency have focused on dynamic or switching power, which arises from the repeated charging and discharging of transistors and wires due to computing activity. Leakage or static power is another source of power dissipation and is due to the sub-threshold current that flows through transistors even when they are not switching. This static power has so far been a less significant component of power dissipation and hence has not been heavily studied at the architecture level. But as fabrication processes have worked to maintain clock speeds while scaling supply voltage, threshold voltages are being lowered to the point where leakage has become an important and growing fraction of total power dissipation in high-performance CMOS CPUs. If it is not addressed through fabrication or circuit-level changes, some forecasts predict as much as a five-fold increase in leakage energy per technology generation [2]. At such rates, the current leakage component, roughly 5% of total chip power now, would balloon to 50% or more in just a few generations.

In response to this trend, researchers have proposed circuit- and architecture-level mechanisms for managing leakage energy [10, 18]. In particular, prior work by Kaxiras *et al.* on cache *decay* techniques [10] showed that turning cache lines off if they have not been used in a long time (the *decay interval*) can be an effective way of reducing leakage energy with little performance impact. Evaluation must of course consider the power overhead that arises from any additional state required for applying decay, from any extra cache misses that are induced when decay is too aggressive, and from any increase in execution time. The appropriate metric is therefore the *net* reduction in cache leakage energy.

After caches, branch predictors are among the largest array structures in current CPUs and thus can be large dissipators of leakage power. Since applying decay techniques to caches has proven effective, applying decay techniques to branch predictors is an obvious next step.

Unfortunately, several factors complicate this task, for it is much less obvious when a branch predictor entry may be considered "dead" and can therefore be turned off with little performance impact. First, many branches may map to the same predictor entry and since this sharing is sometimes beneficial, notions of cache conflicts and eviction do not translate directly into the branch prediction world. Second, a branch predictor entry is not simply valid or invalid, as in a cache. A branch predictor entry may have reached the "strongly not taken" state due to the effects of several different branches and may be useful to the next branch that accesses it, even if this branch has never been executed before. Third, individual branch predictor entries are too small to deactivate individually, so one must consider some larger collection, such as a row of the square array in which the predictor is likely implemented. The challenge here is that unlike a grouping of data into a cache line, the grouping of branch predictor entries in a row is not something for which application programmers and systems builders have a sense of spatial locality. This paper evaluates design options related to these questions.

Further interesting questions arise when moving from simple *bimodal* branch predictors [15], which simply keep one two-bit counter per predictor entry, to multi-table predictors like *hybrid* predictors [13], which operate several prediction structures in parallel. For example, hybrid predictors may encounter instances when one of the predictor components has decayed but the other one has not. The chooser might be designed to pick the non-decayed component in such situations. Alternatively, for some branches, the chooser may exhibit a strong bias for one predictor component over the other. In this case, predictor entries that are not being selected might be deactivated.

This paper shows that although branch predictor decay methods cannot be adapted directly from their cache counterparts, one can nevertheless devise effective branch predictor decay arrangements despite their more nuanced behavior. The paper then goes on to explore the interaction of decay policies with some of the wealth of branch predictor design parameters. We show that:

- Decay can reduce net leakage energy in the conditional branch predictor by 40–60%.
- Decay can reduce net leakage energy in the branch target buffer (BTB) by 90%.
- Decay is slightly less effective for two-level predictors than for structures indexed strictly by branch PC.
- In hybrid predictors, decay policies can achieve 50% higher reductions in leakage energy if the decay policy takes advantage of the hybrid predictor organization to boost decay opportunities.
- Decay is most effective for intervals of 64K cycles or larger. If decay is applied too aggressively, extra mispredictions result with significant costs in both performance and dynamic power.

The next section describes the power-performance simulation infrastructure, benchmarks, and metrics used in this study. Section 3 lays out our basic decay ideas, which are then evaluated for three branch predictor types in Sections 4 (bimodal), 5 (gshare) and 6 (hybrid). Branch predictors commonly operate in cooperation with branch target buffers, so Section 7 shows that decay methods can work with these structures too. Section 8 concludes the paper and includes some suggestions for future work.

2 Experimental Setup

This section describes our simulation technique and benchmarks, our method for evaluating energy savings, and the branch predictor types used in our evaluation.

2.1 Simulation Setup

Simulations in this paper are based on the SimpleScalar 3.0 toolkit [1, 4]. Our model processor has microarchitectural parameters that resembles in most respects the Intel PIII processor [6]. The main processor and memory hierarchy parameters are shown in Table 1. For performance estimates and behavioral statistics, we use SimpleScalar's *sim-outorder* simulator. For energy estimates, we use the Wattch simulator [3]. Wattch uses SimpleScalar's *sim-outorder* cycle-accurate model and adds cycle-by-cycle tracking of power dissipation by estimating unit capacitances and activity factors. Because most processors today have pipelines longer than 5 states, our simulations extend the sim-outorder pipeline by adding three additional stages between decode and issue. For Wattch, it is necessary to specify technology parameters and clock speed. We chose a feature size of 0.18μ, a V_{dd} of 1.9V, and a clock speed of 1 GHz.

2.2 Benchmarks

We evaluate our results using benchmarks from the SPEC CPU2000 suite [17]. The benchmarks are compiled and statically linked for the Alpha instruction set using the Compaq Alpha compiler with SPEC *peak* settings and include all linked libraries. For each program, we skip the first 1 billion instructions to avoid unrepresentative behavior at the beginning of the program's execution. We then simulate 500M (committed) instructions using the reference input set. Simulation is conducted using SimpleScalar's EIO traces to ensure reproducible results for each benchmark

Processor Core	
Instruction Window	16-RUU, 8-LSQ
Issue width	4 instructions per cycle
Functional Units	4 IntALU,1 IntMult/Div, 4 FPALU,1 FPMult/Div, 2 MemPorts
Memory Hierarchy	
L1 D-cache Size	16KB, 4-way, 32B blocks
L1 I-cache Size	16KB, 1-way, 32B blocks
L2	Unified, 256KB, 4-way LRU, 64B blocks,6-cycle latency, WB
Memory	18 cycles
TLB Size	128-entry, 30-cycle miss penalty
Branch Predictor	
Branch predictor	4K-entry bimod or 16K-entry gshare or 21264-style hybrid
Branch target buffer	2048-entry, 4-way
Return-address-stack	32-entry

Table 1: Configuration of simulated processor

across multiple simulations. Table 2 provides a list of the benchmarks we study along with their basic branch and performance characteristics.

2.3 Energy Evaluation

An evaluation of the net energy savings from branch predictor decay must account for two opposite effects resulting from decay. On one hand, turning off V_{dd} in idle counters or rows can prevent them from leaking; on the other hand, decaying the counters causes them to lose history information stored with them, possibly degrading performance of the branch predictor and hence possibly resulting in more energy spent on mis-speculation and longer run time.

We use Wattch [3] to measure the total processor dynamic energy before and after applying decay, and subtract them to obtain the dynamic energy overhead. For leakage energy, from the low-V_t data given in Table 2 in [18], we know that each bit (SRAM cell) consumes about $1740 * 10^{-9}$ nJ per cycle. Here we are assuming a 1GHz processor frequency. Therefore, the 4K bimod predictor, 16K gshare predictor and 21264 style hybrid predictor consume about 0.014nJ, 0.056nJ and 0.030nJ leakage energy each cycle respectively. The overall leakage energy for the branch predictor can be calculated as

leakage energy per bit per cycle * number of bits * number of cycles

The leakage energy of the extra status bits can be calculated similarly. Since these status bits only switch infrequently (at most once every decay interval, *e.g.*, once every 64K cycles), we ignore their dynamic energy overhead in our calculation.

To evalulate the net effectiveness of decay for reducing leakage energy, we combine the new, reduced value for leakage energy with the extra overhead energy associated with the decay technique, and then compare to the original value for leakage energy. For each of the predictor types we study, we present plots of normalized leakage energy for different decay intervals, where the basis for normalization is the original value for leakage energy.

This approach for measuring the net reduction in leakage energy is similar to the techniques used by Kaxiras *et al.* in their work on cache decay [10].

	Dynamic Conditional Branch Frequency	Prediction Rate w/ Bimod 4K	Prediction Rate w/ Gshare 16K	Prediction Rate w/ Hybrid 21264-style
gzip	9.40%	87.58%	90.67%	92.40%
vpr	11.08%	90.00%	97.68%	97.03%
gcc	2.56%	99.61%	99.74%	99.75%
mcf	19.40%	98.42%	99.27%	98.94%
crafty	11.13%	91.76%	93.33%	94.38%
parser	15.60%	91.25%	94.31%	94.89%
eon	11.08%	81.03%	89.71%	91.76%
perlbmk	12.43%	95.15%	97.29%	97.60%
gap	6.62%	90.10%	96.50%	96.96%
vortex	16.00%	97.85%	97.61%	97.87%
bzip2	12.13%	94.06%	94.05%	94.12%
twolf	12.24%	86.44%	88.53%	88.48%
wupwise	10.50%	92.05%	97.54%	96.66%
swim	1.35%	99.36%	99.54%	99.55%
mgrid	0.33%	92.57%	97.77%	97.95%
applu	0.28%	93.07%	98.70%	98.21%
mesa	8.73%	94.09%	96.98%	96.31%
galgel	6.09%	99.13%	99.27%	99.29%
art	11.29%	92.99%	99.02%	96.40%
equake	17.13%	98.04%	99.39%	99.28%
facerec	3.49%	97.92%	99.04%	99.21%
ammp	21.67%	98.77%	99.27%	99.23%
lucas	8.67%%	90.84%	98.70%	98.84%
fma3d	18.09%	95.93%	98.39%	97.68%
sixtrack	8.08%	89.58%	99.72%	99.78%
apsi	3.51%	86.22%	96.52%	97.12%

Table 2: Benchmark summary.

2.4 Branch Predictors Studied

Although a wealth of dynamic branch predictors have been proposed, we focus on the effects of decay for a representative sample of predictor types: bimodal, gshare, and hybrid.

The bimodal predictor [15] 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) map to the same PHT entry, and means that there are never more PHT entries in use at any one time than there are active branch sites. This paper models a 4 K-entry (8 Kbit) bimodal predictor. This is the configuration that appears in the Alpha 21064 [7], although the 21064 uses one-bit rather than two-bit counters. The Alpha 21164 [8] used a larger PHT of 8 K entries, but we conservatively choose the smaller PHT to make it more difficult to show benefits from decay.

The gshare predictor [13], shown in the left-hand portion of Figure 1, is a variation on the two-level global-history predictor [14, 19]. The advantage of global history is that it can detect and predict sequences of correlated branches. In a conventional global-history predictor, a history (the global branch history register or GBHR) of the outcomes of the N 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 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 [16].

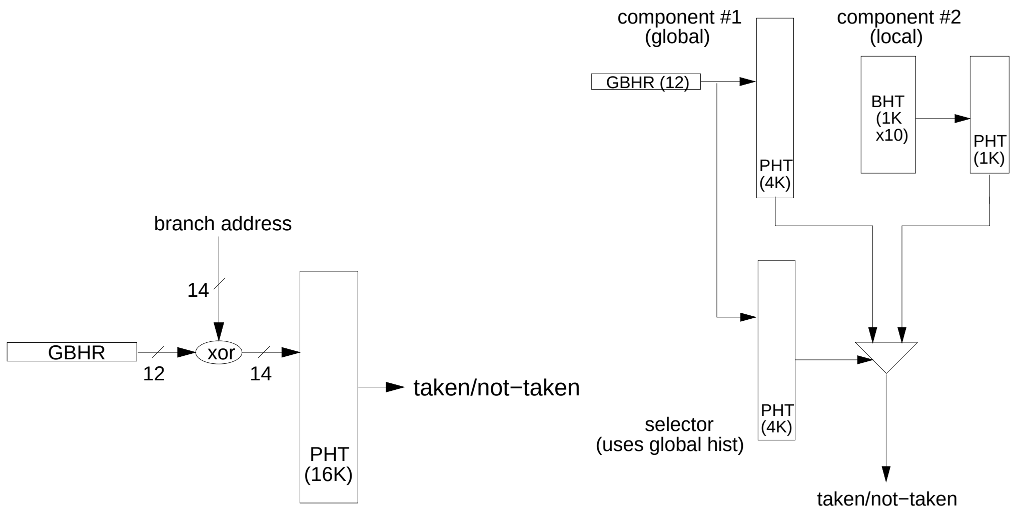


Figure 1: Gshare predictor in the Sun UltraSPARC-III (Left) and 21264-style hybrid predictor (Right).

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. 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.

Because most programs have some branches that perform better with global history and others that perform better with local history, a hybrid predictor [5, 13] 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. Choosing 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 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 [11] and is depicted in the right-hand portion of Figure 1.

3 Opportunities for Decay in Branch Predictors

3.1 Overview of Proposed Implementations

Our predictor decay techniques have the following general structure. At regular intervals, all groups of predictor entries that have not been used during the interval are assumed to have *decayed* and are therefore *deactivated*. The interval, called the *decay interval*, is measured in processor cycles, and is a critical parameter for these schemes. The shorter the interval, the more opportunities for rows to be deactivated but the more likely it is that rows are deactivated prematurely and induce extra mispredictions. Intervals long enough to mimimize extra mispredictions, on the other hand, result in the deactivation of fewer entries.

The groups of entries are the *rows* of a square memory array (see Figure 2). One bit per row,

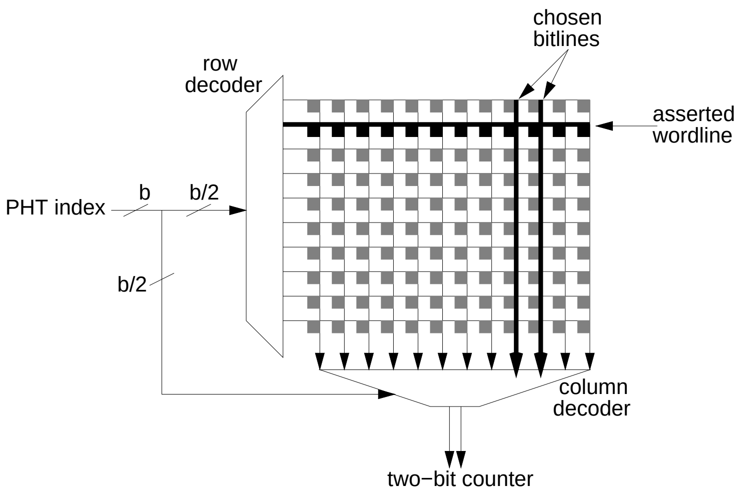


Figure 2: A schematic of a squarified branch predictor table of two-bit counters (the PHT).

the *reference bit*, indicates whether any predictor entry in that row has been accessed within the last decay interval. All reference bits are cleared at the end of each interval. A second bit for each row, the *active bit*, indicates whether that row is currently active (*i.e.*, not decayed). If a predictor lookup tries to access a decayed row, the predictor signals that a prediction cannot be made; the row is re-activated and possibly initialized to some desired starting state; the active bit is set; and in the meantime a default prediction is made. Upon activation, our experiments use a default prediction of not taken and initialize all the counters to 01. Thus, all subsequent branches using the re-activated line start in the weakly not taken state.

The *active ratio* in a particular experiment is the average percentage of predictor rows found to be active (not decayed) at the end of each interval; it is a proxy for the actual leakage energy consumed by the predictor. Of course shorter decay intervals yield smaller active ratios (and larger leakage energy savings), but performance may suffer, since useful predictor entries are sometimes deactivated. Exploring this power-performance tradeoff is a key objective of this paper.

As mentioned in the previous section, the *net* reduction in leakage energy is calculated by measuring the savings in leakage energy from deactivated rows, and subtracting both the leakage energy from the decay status bits and the extra dynamic energy resulting from extra mis-speculation and execution time caused by induced mispredictions. If decay is therefore causing too many mispredictions or the overhead of the decay status bits is too large, this is indicated by a lack of net energy savings or even higher energy costs.

Logically, branch predictor structures are tall and narrow. That is, they may be thought of as having many entries (which makes them tall) but each entry is typically just two bits wide (which makes them narrow). And logically, when first considering applying decay techniques to branch predictors, it might be tempting to consider deactivating individual predictor entries. This is untenable, however, because our methods require two bits of state per independently-activated unit; such overhead would be excessive if applied to individual two-bit predictor entries.

Physically, however, branch predictors are, like caches, typically implemented as square or nearly-square array structures. "Squarifying" the predictor array helps to minimize the complexity of the row and column decoders and balance wordline and bitline length and delay. The predictor array is similar to a cache array, except that it needs no tags. Since predictors are squarified, a predictor structure consists of approximately $\sqrt{entries}$ wordlines (rows), each connected to approximately $2 * \sqrt{entries}$ bits. On a branch predictor access, the row decoders activate one wordline, and the values of these entries are then sent on the bitlines to the column decoder, which selects the one entry specified by the predictor index. Figure 2 shows a diagram of a typical branch predictor.

The physical implementation of branch predictors as squarified arrays is relevant to our branch predictor decay studies, because it means that a natural choice for turning off entries might be at the granularity of rows in the array structure rather than individual entries. This requires only two bits of state per row (the reference bit and the active bit) and hence a total overhead of $2 * \sqrt{entries}$ bits.

3.2 Spatial and Temporal Locality in Branch Predictors

The first question in exploring decay for branch predictors is to determine how often an entire row of branch predictor entries is likely to lie idle long enough for decay techniques to be effective. In today's machines, branch predictor rows typically include 16-64 counter entries apiece. Very large programs with random branch distributions might be expected to scatter branches across the predictor and keep at least one of these counter entries active at all times. But if program branches are clustered rather than random, then we should see some rows with heavy activity while other rows are idle and can be deactivated. A basic test, then, is to show that branch-predictor accesses exhibit spatial or temporal locality at the granularity of array rows.

3.2.1 Spatial Locality

Clearly, programs exhibit spatial locality in the instruction cache. Over a short period of time, only one or a few small contiguous regions of the program are likely to be active, and branch accesses are therefore likely to be close in terms of their PC. Even taken branches are often likely to remain close in terms of PC, since many taken branches just perform short jumps associated with if-then-else statements.

Code locality can translate to spatial locality in branch predictor array rows as well. This is most true for the bimodal predictor, which is indexed only by PC. Indeed, the probability that two successive conditional branches fall into the same row in a 4 K-entry bimodal predictor is greater than 40% for all our benchmarks, and greater than 50% for all but five. If the branches were uniformly distributed, we would expect rates close to $1/rows$ (about 2% for this 4 K-entry predictor array). For gshare or hybrid predictors, spatial locality is not as pronounced as for bimodal, since these other predictors are indexed by the branch history. Nevertheless, in half of the benchmarks the probability of hitting the same row as the previous branch is above 10%. This is much higher than a random distribution (about 1% for the large, 16 K-entry gshare), so these predictors also exhibit some spatial locality.

3.2.2 Temporal Locality

Strong spatial locality means that at any point in the program, active rows are likely to have many counters active and idle rows are likely to be entirely idle. But without temporal locality, the rows that are active might change rapidly, reducing opportunities for decay. Fortunately, many benchmarks have small static branch footprints (the number of unique branch instruction sites that are executed), as seen in Table 3. Decay will therefore clearly help bimodal prediction, because each static branch touches only one predictor entry and we know from the data above that these are clustered.

Other predictor structures, however, may not do as well. With gshare in particular, the branch address is XOR'd with the global branch history, so that one branch can touch many PHT entries. It is therefore helpful to measure, for various interval lengths, how many rows stay inactive for

sample interval	1K cycles	10Kc	100Kc	1Mc	Overall
gzip	24	32	45	103	281
vpr	31	45	58	65	742
gcc	2	9	79	193	512
mcf	65	83	92	116	565
crafty	104	305	592	855	1701
parser	53	90	157	294	2265
eon	81	289	357	415	652
perlbmk	90	453	631	1112	1541
gap	62	281	325	576	745
vortex	124	502	1227	1642	1996
bzip2	22	33	45	56	460
twolf	48	210	300	334	351
wupwise	42	52	53	55	193
swim	3	6	11	15	687
mgrid	3	6	9	25	500
applu	1	2	4	7	579
mesa	83	114	139	267	697
galgel	2	6	8	10	508
art	2	2	5	18	109
equake	167	192	193	202	226
facerec	7	24	25	39	144
ammp	11	26	105	230	794
lucas	3	3	3	4	242
fma3d	80	450	452	465	499
sixtrack	39	49	55	99	734
apsi	14	85	117	125	342
geomean	20	46	70	109	529

Table 3: Average number of static branches touched every sample interval for SPEC2000. The rightmost column labeled ‘Overall’ gives the static branch footprint for the whole simulation period.

the duration of the interval. This can be measured by the active ratio. Smaller active ratios are better for decay. Figure 3 shows the geometric mean for active ratio across the benchmarks for both banked and unbanked 16 K-entry gshare predictors and the 4 K-entry bimodal predictor.

As expected, the active ratio is quite small for the bimodal predictor. The active ratio is larger (*i.e.*, worse from a decay point of view) for gshare. Yet significant numbers of rows remain untouched. This indicates that even for predictor structures designed to smear branch addresses over many entries, decay-based techniques still show significant promise for addressing leakage concerns.

We include data in Figure 3 for a banked version of gshare, because breaking the predictor into banks makes the active ratio smaller (better for decay) by reducing the granularity over which activity is measured. Indeed, the active ratio for the banked organization is 15–35% smaller if the large gshare predictor is broken into four banks of 4K entries each.

4 Decay with Bimodal Predictors

The active-ratio statistics in the previous section suggest substantial opportunities for decay to reduce leakage power with minimal performance penalty. This section and the two that follow therefore look in more detail at the success of decay techniques for bimodal, gshare, and hybrid predictors.

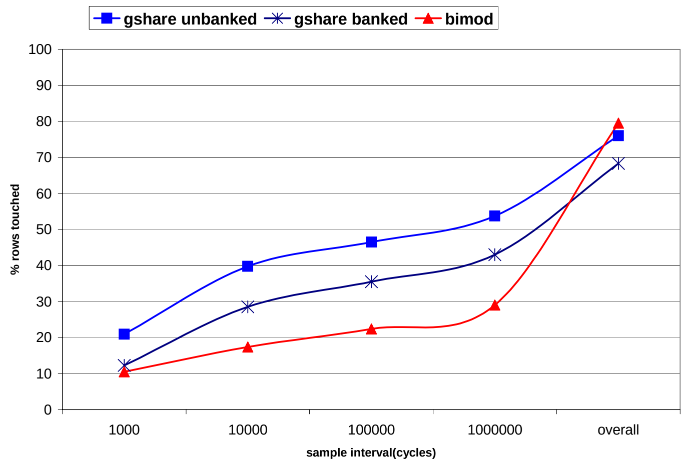


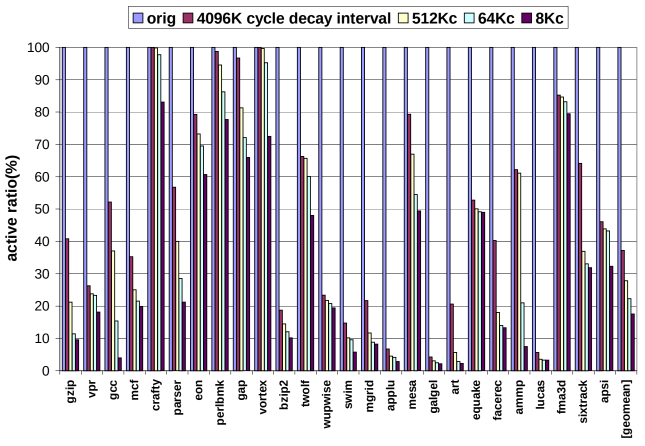
Figure 3: Mean active ratio for unbanked and banked gshare predictors and a bimodal predictor.

Figure 4 shows the active ratio and direction prediction accuracy for 4K-entry bimodal branch predictors with different decay intervals. We see from the active ratio graph that decay is very effective at shutting off idle counters in bimodal predictors. The geometric mean active ratio, which is the percentage of predictor entries powered on (so smaller values are better), is 37%, 28%, 22% and 18% for decay intervals of 4096K, 512K, 64K and 8K cycles respectively. This means that decay techniques have the potential to reduce branch predictor leakage by 2X or more. Since bimodal predictors are indexed by PC, benchmarks with large static branch footprints tend to have higher active ratios, as shown in *crafty, perlbmk, gap and vortex*. Other benchmarks typically leave more than half of their counters deactivated due to their small footprints.

Furthermore, the prediction rate graph shows that shutting off these idle counters comes with minimal loss in performance except for the apparently too-aggressive decay interval of 8K cycles. With a 64K cycle decay interval, the overall loss in prediction accuracy is about 0.14%, with only one benchmark (*mgrid*) over 1%.

Interestingly, in some benchmarks (*wupwise* and *facerec*) we actually observed tiny improvements in prediction accuracy. We attribute this to the effect of removing some destructive interefer-ence. Interference occurs when two different branches map to the same counter. The interference is destructive when the branches are biased in opposite directions, for example, when one of the conflicting branches is strongly taken and the other is strongly not taken. Deactivation resets the counter to the neutral, weakly not taken value, which effectively isolates the two conflicting branches.

Figure 5 compares the leakage energy before and after applying decay. When decay is enabled, we add to the leakage energy the extra leakage energy from the decay status bits as well as any dynamic-energy overhead due to extra mispredictions or longer run time. We show the geometric mean reduction in leakage energy for SPEC2000 benchmarks. The figure demonstrates the trade-off between savings in leakage energy and the overhead incured. With a small decay interval such as 8K cycles or less, branch prediction rate degrades enough so that the dynamic energy overhead dominates the savings in leakage energy. When the decay interval is longer, few mispredictions are induced, the extra dynamic energy incurred becomes minimal, and over 60% of the original leakage energy can be saved.



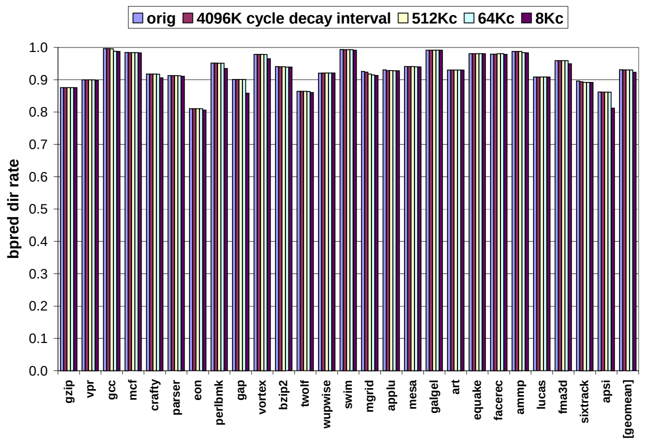


Figure 4: Active ratio (Left) and prediction success rate (Right) for a 4 K-entry bimodal predictor

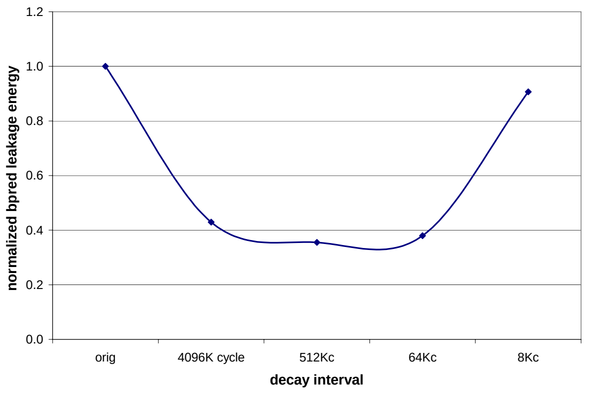


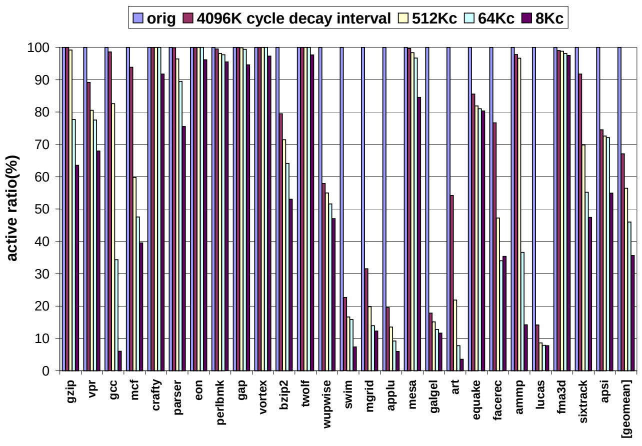
Figure 5: Normalized leakage energy for the 4K bimodal predictor

5 Decay with Gshare Predictors

Figure 6 shows active ratio and direction prediction accuracy for an un-banked global-history predictor. The geometric mean active ratio across the 26 benchmarks is 46% for a 64 K-cycle decay interval, and the average drop in predictor accuracy is negligible. 10 benchmarks stand out as having less energy benefit from decay, but even these benchmarks suffer negligible loss in prediction accuracy, so decay does not harm their performance. Because gshare is designed to spread branch state across the predictor to minimize aliasing, its decay benefits are not as pronounced as for the bimod predictor. Nevertheless, decay still produces substantial reductions in leakage power with minimal performance impact. For example, Figure 7 shows that the reduction in leakage energy is 41% for a 64 K-cycle decay interval.

Further gains can be realized by breaking the predictor into banks. In the interests of space, we do not present extra graphs for active ratio and misprediction rate. But for the same 64 K-cycle interval and the banked 16K gshare predictor, the active ratio falls to 36% and, as with unbanked, misprediction rate remains unaffected. Leakage energy falls to 51% of its original value.

Note that, with the banked predictor, the reduction in leakage energy will be somewhat less than the reduction in active ratio for two reasons. First, because the banked organization has smaller rows and hence the overhead of more decay status bits. Second, because the banked organization is more aggressive than unbanked gshare in deactivating rows for the same decay interval. This makes it more likely that the banked organization will deactivate a row that actually harms prediction accuracy. This extra dynamic-power overhead, however, decreases with increasing decay interval



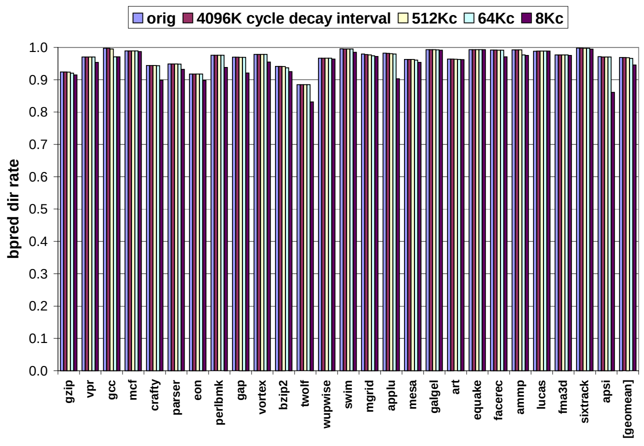


Figure 6: Active ratio (Left) and prediction success rate (Right) for unbanked gshare predictor

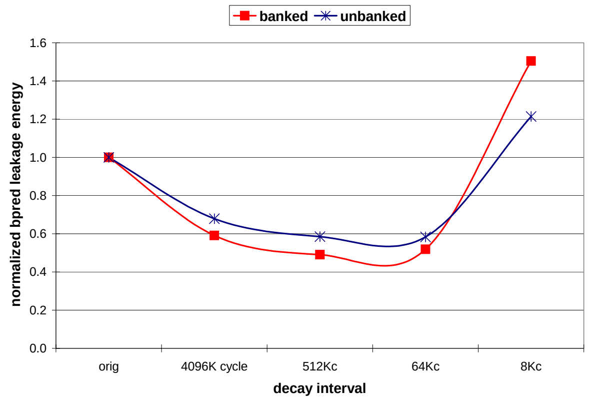


Figure 7: Normalized leakage energy for gshare branch predictor for both unbanked and banked predictors.

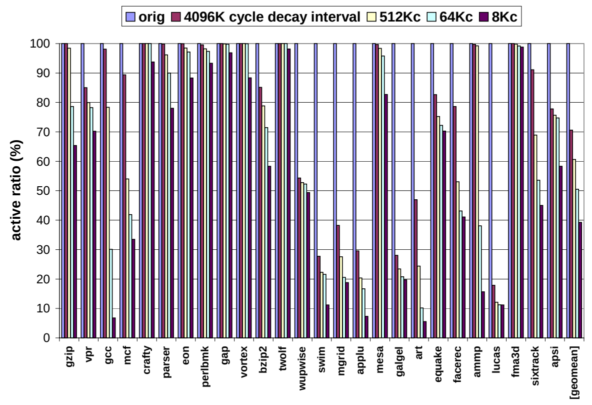
because for very long intervals, even a banked row that is idle is extremely likely to have genuinely decayed. This is why the difference in energy savings is larger for 512K cycles and 4096K cycles than for 64K cycles.

6 Decay with Hybrid Predictors

This section examines decay for hybrid predictors. First we examine the effectiveness of decay for a straightforward application of decay to a predictor like that in the Alpha 21264; then we explore more sophisticated decay policies. Note that when a row in the chooser is reactivated after decay, its counters are set to “weakly choose local.”

6.1 Naive Decay

In Figure 8, we see that even though decay has negligible impact on prediction rate for intervals of 64K cycles or larger, the active ratios are also higher than bimod or gshare. (In order to compute active ratio sensibly on a multi-table structure, it is computed over all prediction and chooser bits in the structure.) Nevertheless, as Figure 9 shows, decay still realizes strong reductions in energy savings—40% for a 64 K-cycle interval.



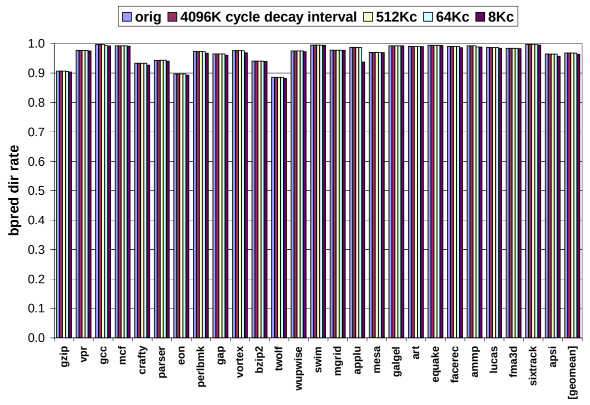


Figure 8: Active ratio (Left) and prediction rate (Right) for 21264's hybrid predictor

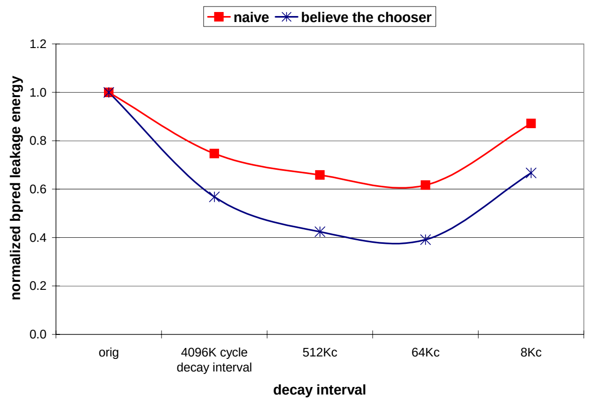


Figure 9: Normalized leakage energy of 21264-style Hybrid predictor with naive and "believe the chooser" policies

This energy savings is larger than the preceding gshare data would suggest: the two subcomponents in the hybrid predictor are only 4K-entry each and should have higher active ratios than the 16K-entry structures examined previously. The reason decay performs better here is that the components are indexed based solely on history bits, rather than being indexed by history bits XORed with the branch address. Omitting the XOR allows more spatial locality in the branch array, and improves the potential for decay. Space considerations prevent us from presenting active ratio information separately for all three components.

We can obtain even better energy savings by taking advantage of particular aspects of the hybrid structure. For example, in the naive approach, all three predictor structures are probed every cycle. Even if the row in one component has decayed while the other remains active and capable of making a prediction, all three components are forced active. This is true even if the decayed row will not be chosen by the selector. This observation suggests more intelligent wakeup policies, which are explored in the next section.

6.2 Decay Choices in Hybrid Predictors

By improving on the previous subsection's naive decay policy, this section's proposals give the hybrid predictor excellent decay performance, making it one of the most power-efficient predictor organizations while still preserving its excellent prediction accuracy. The interaction of decay policies and hybrid-predictor parameters also highlights some interesting design choices.

In a hybrid predictor, a lookup is performed on both component predictors for every branch. Only one result, however, is selected by the chooser, while the other is discarded. This redundency allows more flexible decay policies to be employed. Since the three components (global, local and chooser) each can be in either of two states (active or decayed), there are a total of eight possible combinations of decayed and active components. Our schemes take advantage of the fact that in the 21264's hybrid predictor, the chooser has the same configuration as the global-history component and hence they always share the same index. This in turn means that a particular row is always in the same state of decay in both the global-history predictor and the chooser. Only three combinations of decay may therefore occur in the 21264 predictor:

1. The rows in all three components are active. Here no decay or wakeup decisions need to be made. Each component assumes its normal function as in a conventional hybrid predictor.

2. The rows in all three components have decayed. In this situation all three rows are activated. The branch is predicted "weakly not taken," which is the default value when decayed components are reactivated.

3. The row in only one of the two components is decayed. Since one of the predictors is still active, it will most likely be more accurate than the other, which has lost all history information due to decay. So the prediction from the active component is used as the prediction of the whole hybrid predictor, regardless of whether the chooser is awake and regardless of what its choice would be.

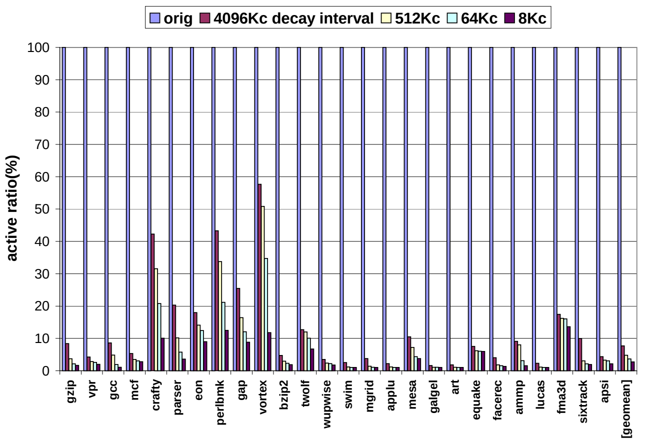
   In this situation, there remains the decision of whether to reactivate the decayed component. In the naive approach of Section 6.1, we always reactivate the decayed component, which results in high active ratios. In this section, however, the deactivated component will be reactivated only if the chooser wants to select it. Note that if the row in the global-history component is inactive, the same row in the chooser must also be inactive. In this case the chooser has no useful information, so we leave the chooser and the global-history component inactive and return the prediction from the local-history component. We call this policy "believe the chooser".

As shown in Figure 9, this more sophisticated policy leads to leakage power reductions that are about 50% better than for the naive policy.

7 Decay with the Branch Target Buffer

In addition to the structure for predicting the direction of conditional branches, a branch target buffer (BTB) [9, 12] is commonly used to store the targets of taken branches. The BTB is typically organized like a cache, either direct-mapped or set associative, but tagged in order to identify hits and misses. Since it is solely indexed by PC, it has similar locality characteristics as instruction caches and bimodal predictors. However, the granularity in the BTB is single branch instructions, instead of instruction blocks as in instruction caches. This allows even finer control for decay. To round out our exploration of decay for branch prediction, this section briefly evaluates the effectiveness of decay for a 2048-entry, 4-way associative BTB, which appears in the Intel PIII processor [6].

Figure 10 depicts active ratios and hit rates of the BTB with different decay intervals. Most benchmarks have a very low active ratio, except for *vortex*, *perlbmk*, *crafty*. This is no surprise, considering that the static footprints of most benchmarks in Table 3 are very small compared to our BTB capacity. With decay, these static footprints are automatically tracked and all other idle slots



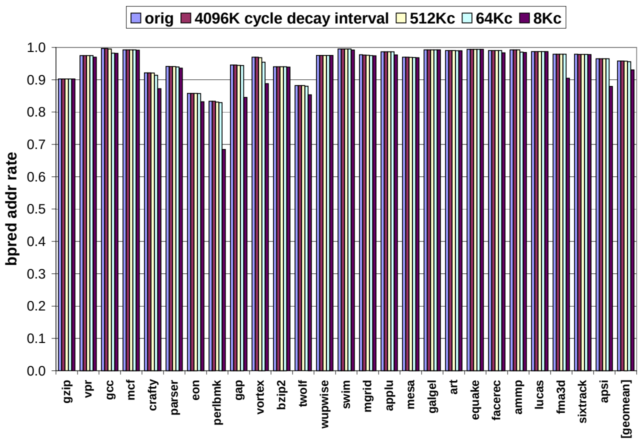


Figure 10: Active ratio (Left) and hit rate (Right) for BTB

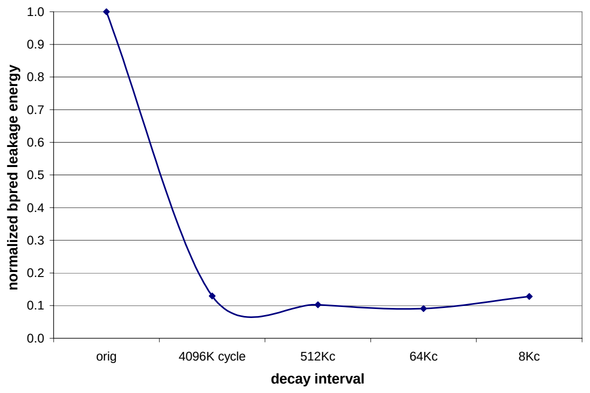


Figure 11: Normalized leakage energy for BTB with different decay intervals

are turned off to save leakage energy. On the other hand, BTB hit rates observe only negligible degradation until decay interval drops to 8K cycles. Figure 11 depicts the effect of decay on leakage energy. Overall, about 90% of the leakage energy can be saved.

8 Conclusion and Future Work

In this paper we have explored the application of decay-based techniques to reduce static or leakage power in branch predictor structures. Our results show that all the predictor organizations we studied have significant numbers of rows that are inactive for long periods of time. For sufficiently long decay intervals, decay can therefore be highly effective, while barely affecting performance. For all the configurations we studied and for decay intervals of 64 K-cycles or larger, the reduction in branch prediction accuracy is always less than 1% and less than 0.2% for most benchmarks. For these same decay intervals, branch predictor leakage energy can be reduced by 40–60%.

Over all the configurations we explored, the best reductions in leakage power were achieved with the bimodal predictor (Section 4), but the power savings achieved with the "believe the chooser" policy for hybrid prediction were nearly as good. Since hybrid prediction has superior prediction accuracy and hence performance, hybrid prediction remains the best choice from both a performance and energy standpoint.

Some of our interesting results were specifically due to the fact that in the 21264's hybrid predictor, the chooser and GAg component were the same. This let us deduce useful facts about one based on state in the other. More generally, the influence of indexing on decay may suggest that the choice of predictor index is worth revisiting (yet again). In the past, indexing functions have been chosen to minimize aliasing, usually by spreading the state from branches in the same working set as widely as possible. In contrast, decay can be made more effective by clustering this state into rows, with the goal of making as many counters in a row as possible idle or active at any point in time. This observation therefore sets up a tradeoff between reducing aliasing and increasing decay opportunities. Although beyond the scope of this paper, seeking index functions that can balance the two factors is an area for future work. It may be possible to develop tractable index functions that cluster state into rows with minimal increase in aliasing. In addition, large predictors in particular may be able to accommodate index functions that boost row clustering.

The techniques in this paper applied fixed decay intervals. Kaxiras *et al.* [10] showed that for caches, even better decay rates can be achieved with adaptive decay intervals that adapt to changing program behavior. While studying adaptive branch predictor decay was beyond the scope of this paper, it is worth investigating since our data showed that some benchmarks suffered no performance loss even with a short, 8 K-cycle decay interval (and would get even more benefit from decay), while others suffer severely with such a short interval.

A further consideration is that banked and multi-table organizations provide substantial benefits in reduced *dynamic* energy, by reducing word- and bit-line lengths. Furthermore, when decay is applied to multi-table predictors, some tables can be left inactive, as in the "believe the chooser" policy. Another example arises in local-history prediction, where timing issues may require caching the predicted direction for each BHT entry in the BHT. As with "believe the chooser", decay might permit the PHT update and lookup in such a local-history organization to be omitted if that PHT entry is inactive and the cached direction was correct. Such a policy might be called "believe the BHT". Not only do these "believe" policies reduce leakage energy, they avoid the dynamic power associated with accessing that table. Our work here did not model these extra savings, but doing so would only make decay more valuable.

Another issue that warrants study is interference in branch predictors. In our experiment we observed interesting improvements in prediction rate with decay (see Section 4). This shows that decay (by setting the two-bit counters to a weak state) may have the effect of reducing destructive interferences, something we plan to quantify in our future work.

Leakage energy is expected to become a substantial portion of total energy expended in the processor in future CMOS generations. The results in this paper show that decay can mitigate these effects by dramatically reducing leakage energy expended in the branch predictor and BTB.

References

[1] T. M. Austin. SimpleScalar home page. http://www.simplescalar.org.

[2] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4), 1999.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architecture-Level Power Analysis and Optimizations. In *Proc. of the 27th Int'l Symp. on Computer Architecture*, ISCA 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] 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.

[6] K. Diefendorff. Pentium III = Pentium II + SSE. *Microprocessor Report*, Mar. 8 1999.

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

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

[9] R. W. Holgate and R. N. Ibbett. An analysis of instruction fetching strategies in pipelined computers. *IEEE Transactions on Computers*, C-29(4):325–329, Apr. 1980.

[10] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *Proc. of the 28th Int'l Symp. on Computer Architecture*, July 2001. To appear.

[11] 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.

[12] J. J. Losq. Generalized history table for branch prediction. *IBM Technical Disclosure Bulletin*, 25(1):99–101, June 1982.

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

[14] 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.

[15] 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.

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

[17] The Standard Performance Evaluation Corporation. WWW Site. http://www.spec.org, Dec. 2000.

[18] S.-H. Yang et al. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches. In *Proc. of the Seventh Int'l Symp. on High-Performance Computer Architecture*, 2001.

[19] 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, Nov. 1991.