Applying Decay Strategies to Branch Predictors for Leakage Energy Savings

Zhigang Hu[†] Philo Juang[†] Kevin Skadron[‡] Douglas Clark[*] Margaret Martonosi[†]

[†] EE Department, Princeton University, {hzg, pjuang, mrm}@ee.princeton.edu
[‡] CS Department, University of Virginia, skadron@cs.virginia.edu
[*] CS Department, Princeton University, doug@cs.princeton.edu

Abstract

With technology advancing toward deep submicron, leakage energy is of increasing concern, especially for large on-chip array structures such as caches and branch predictors. Recent work has suggested that even larger branch predictors can and should be used in order to improve microprocessor performance. A further consideration is that the branch predictor is a thermal hot spot, thus further increasing its leakage. For these reasons, it is natural to consider applying decay *techniques—already shown to reduce leakage energy for caches—to branch-prediction structures.*

Due to the structural difference between caches and branch predictors, applying decay techniques to branch predictors is not straightforward. This paper explores the strategies for exploiting spatial and temporal locality to make decay effective for bimodal, gshare, and hybrid predictors, as well as the branch target buffer. 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. Multi-component hybrid predictors offer especially interesting implementation tradeoffs for decay.

1 Introduction

As fabrication processes have worked to improve 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 [1]. 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 [8, 10, 19]. 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 can be effective in reducing leakage energy with little performance impact.

After caches, branch predictors are among the largest and most power-consuming array structures in current CPUs. Current predictors are 4–8 Kbytes in size, already the size of a small cache. They dissipate about 10% of the processor's total dynamic power dissipation [14]. Cycle-time, power-dissipation, and thermal concerns tend to keep predictors from growing larger. However, Jiménez *et al.* [9] pointed out that two-level predictors can avoid cycle-time constraints and that large second-level predictors can give substantial increases in prediction accuracy, resulting in predictors that could be as large and have the same substantial leakage as first-level caches. Futhermore, Skadron *et al.* [15] found that the branch predictor is a thermal hot spot, as it is typically accessed every cycle. This indicates that branch predictors expend much more leakage energy than their sizes would suggest, because leakage power increases exponentially with temperature.

Applying decay techniques to caches has proven effective, so 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. 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, branch predictor entries are too small to deactivate individually, so one must consider some larger collection, such as a row of predictor entries in the square array in which the predictor is likely implemented. The challenge here is that unlike the 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 [16], which 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 has not. The chooser might be designed to pick the non-decayed component in such situations. For other 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.

Contributions. Because branch predictors have behavior more nuanced than caches, cache decay methods cannot be directly applied. This paper shows that effective decay techniques can nevertheless be devised for branch predictors. 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% and the branch target buffer (BTB) by 90%.
- 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 simulation infrastructure and benchmarks used in this study. Section 3 briefly reviews the organization of typical branch predictors. Then in Section 4 and Section 5 decay strategies are evaluated for basic and hybrid predictors respectively. Section 6 concludes the paper and includes some suggestions for future work.

2 Experimental Setup

Simulations in this paper are based on the SimpleScalar 3.0 toolkit [3]. Our model processor has microarchitectural

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 target buffer	2048-entry, 4-way

Table 1: Configuration of simulated processor

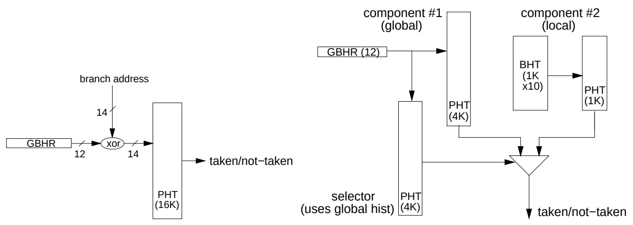


Figure 1: Gshare predictor in the Sun UltraSPARC-III (Left) and 21264-style hybrid predictor (Right).

parameters that most closely resemble the Intel PIII processor [5]. 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 [2].

Our results are evaluated using benchmarks from the SPEC2000 suite [18]. The benchmarks are compiled and statically linked for the Alpha ISA using the Compaq Alpha compiler with SPEC *peak* settings and include all linked libraries. We skip the first billion instructions of each program to avoid unrepresentative behavior at the beginning of the program's execution. We then simulate 500M (committed) instructions using the reference input set. To ensure reproducible results for each benchmark across multiple simulations, simulations are conducted with SimpleScalar's EIO traces.

3 Branch Predictors Studied

Although a wealth of dynamic branch predictors have been proposed, we focus on the effects of decay for the gshare and hybrid predictors, because they present the most interesting tradeoffs.

The gshare predictor, shown in the left-hand portion of Figure 1, tries to detect and predict sequences of correlated branches by tracking a global history (the global branch history register or GBHR) of the outcomes of the N most recent branches. In gshare, the global branch history and the branch address are XOR'd to reduce aliasing. 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 [17].

Instead of using global history, a two-level predictor can track local branch history on a per-branch basis. Local history 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 [4, 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. 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 [6] and is depicted in the right-hand portion of Figure 1.

Logically, branch predictors are arrays of counters that are typically just two bits wide. Physically, however, branch predictors are, like caches, implemented as square or nearly-square array structures. This helps to minimize the complexity of the row and column decoders and balance wordline and bitline length and delay. The predictor array is thus similar to a cache array, except that it needs no tags. For example, the 16K-entry gshare predictor discussed above can be laid out as a 128×128 array of 2-bit counters. Alternatively, it can be divided into 4 banks, each a 64×64 counter array. We refer to these two organizations as "unbanked" and "banked" respectively and will discuss their decay behavior in Section 4.2.

4 Decay with Basic Branch Predictors

Since branch counters are only 2 bits in size, a cost-effective choice for turning off these counters is at the granularity of rows in the array structure rather than individual entries. Our techniques have the following general structure. At regular intervals, all rows of predictor entries 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 to deactivate rows prematurely and induce extra mispredictions. Intervals long enough to mimimize extra mispredictions, on the other hand, result in the deactivation of fewer entries.

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; in the meantime, a default prediction is made. Upon activation, our experiments use a default of not taken and initialize all the counters to 01. Thus, 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); 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.

To evalulate the net effectiveness of decay for reducing leakage energy, we combine the reduced value of leakage energy that was observed with decay, and the overhead energy associated with the decay technique. We then compare this 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].

4.1 Spatial/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 32-256 counter entries. Fortunately, program branches are clustered rather than random, and across all the predictor organizations we examine, our experiments consistently show that some rows have heavy activity while others are idle and can be deactivated.

Clearly, programs exhibit spatial locality in the instruction cache. Over a short period of time, only one or a few small

	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 2: 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.

contiguous regions of the program are likely to be active, so branch instructions are likely to be close in terms of their PC. This also translates into spatial locality in branch-predictor accesses. For branch predictors, 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. 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.

Yet this is not useful if the active rows change rapidly, so temporal locality is also necessary. One immediate factor that creates temporal locality is the fact that many benchmarks have small static branch footprints (the number of unique branch instruction sites that are executed), as seen in Table 2. Decay will therefore clearly help bimodal predictors, because each static branch touches only one predictor entry and we know from the data in Table 2 that they are clustered.

Other predictor structures, however, may not do as well. With gshare, the branch address is XOR'd with the global branch history, so that one branch can touch many PHT entries. We evaluate decay for gshare predictors in the next subsection. Hybrid predictors use global- and local-history predictors as components, which brings more design choices. We explore the design space for hybrid predictors in section 5.

4.2 Results

Figure 2 shows the geometric mean of the active ratios across the benchmarks for both banked and unbanked 16K-entry gshare predictors and, as a reference, the 4K-entry bimodal predictor. As expected, the active ratios are quite small (*i.e.*, good from a decay point of view) for the bimodal predictor. For gshare predictors, the active ratios are larger. 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 2 for a banked version of gshare. 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 gshare

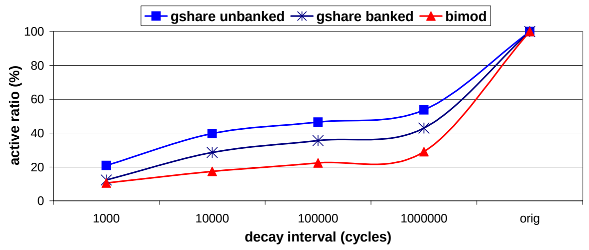


Figure 2: Mean active ratio for unbanked and banked gshare predictors and a bimodal predictor with different decay intervals. The rightmost label "orig" corresponds to non-decaying predictors.

is 15–35% smaller if it is broken into four banks of 4K entries each. Overall, these active ratios yield leakage energy savings of 40% for unbanked gshare and about 50% for banked gshare. Even greater savings can be achieved for structures directly indexed by PC: about 65% for bimodal predictors and 90% for the BTB. For more details, refer to [7].

5 Decay with Hybrid Predictors

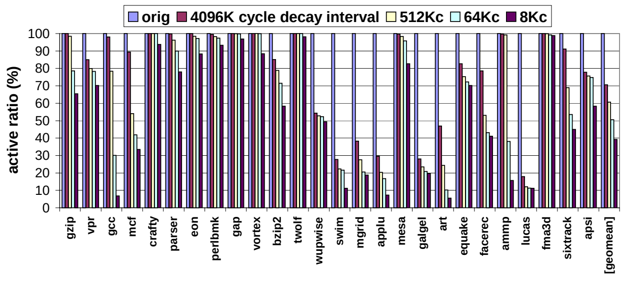
With two competing components (the global component and the local component), hybrid predictors exhibit many interesting design choices when implementing decay. In this section we will explore these design choices as well as their effect on decay in an Alpha 21264-style hybrid predictor.

Selection Policy The selection policy refers to the policy for choosing a prediction from one of the two component predictors. In a non-decaying 21264-style hybrid predictor, the chooser makes this decision using the global history; see Figure 1. However, when decay is enabled, it may happen that only one of the two components is active while the other is decayed. In this case, since the decayed component has lost its information, it is intuitively appealing to use the prediction from the active component, no matter what the chooser suggests. This policy is called "believe the active component", and is implemented in all our experiments. It may also happen that both the two components are decayed, in which case all components are reactivated and the branch is predicted as "weakly not taken", as in bimodal and gshare predictors.

Wakeup Policy The wakeup policy refers to the decision of whether to reactivate a decayed row when it is accessed by a branch instruction. A naive policy would always wakeup any rows that are accessed. In a hybrid predictor, a more elegant policy is possible: the decayed component will be reactivated only if the chooser wants to select it. We refer to this policy as "believe the chooser".

In the situation when the accessed row in the chooser is decayed, we know that the global component is also decayed in the 21264-style predictor. This is because the chooser and global predictor are indexed, accessed and thus decayed in exactly the same way; see Figure 1. In this case, the chooser has no useful information. If the local component is active, then we leave the chooser and the global component inactive and return the prediction from the local component. Otherwise (when the local component is also inactive), we reactivate all components and return a prediction of "weakly not taken".

Results Figure 3 details the active ratio and branch misprediction rate for naive decay, which always wakeup any rows that are accessed, with a 21264-style hybrid predictor. We see that even though the active ratios are higher than for bimodal or gshare predictors, decay has a negligible impact on the misprediction rate for intervals of 64K cycles or larger. Note that in order to compute active ratio sensibly on a multi-table structure, we compute it over all prediction and chooser bits in the structure. Overall, as Figure 4 shows, naive decay realizes strong reductions in energy savings—40% for a 64 K-cycle interval.



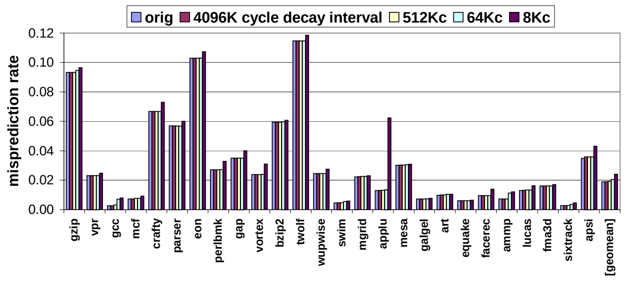


Figure 3: Active ratio (Left) and misprediction rate (Right) for 21264's hybrid predictor

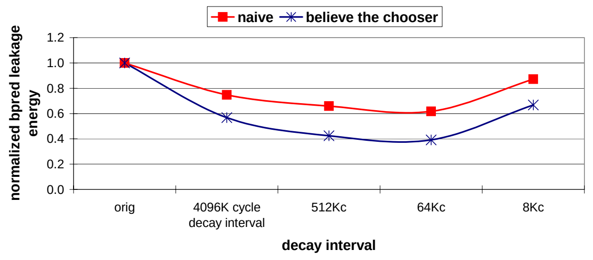


Figure 4: Normalized leakage energy of 21264-style Hybrid predictor with naive and "believe the chooser" wakeup policies

We can obtain even better energy savings by taking advantage of the "believe the chooser" wakeup policy. As shown in Figure 4, this more sophisticated policy leads to leakage power reductions about 50% better than for the naive policy.

6 Conclusion and Future Work

In this paper, we have explored the application of decay techniques to reduce leakage energy in branch predictor structures. Space constraints require that this paper focus on one particular decay technique, gated-Vdd, which loses state when deactivated. But the focus of this paper is on *policies*, not implementation techniques. The particular policies that we have developed (*e.g.*, "believe the chooser") apply to all non-state-preserving leakage-control mechanisms. The key contributions of this work are even more general, giving useful guidance to any work on leakage control. In particular, our locality analysis shows that all predictor organizations we examined have significant numbers of rows that are inactive for long periods of time. This makes leakage control in the branch predictor and BTB profitable, regardless of mechanism.

Non-state-preserving leakage-control mechanisms are well-suited for branch predictors, because the nature of the two-bit counters used in prediction mean that the random values that are present upon re-activation are sometimes correct. Our results show that for decay intervals of 64 Kcycles and larger, decay can be highly effective while barely affecting performance. The reduction in 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% while the BTB leakage can be reduced by about 90%.

Some of our interesting results were specifically due to the fact that there are two independent component predictors in a hybrid predictor. We showed that an intelligent policy that exploit this fact can achieve 50% more leakage energy savings than a naive policy.

One issue that warrants study further is interference in branch predictors. In some experiments we observed mild but interesting improvements in prediction rate with decay. This shows that decay (by setting the two-bit counters to a weak state) may have the effect of reducing destructive interference, 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. Futhermore, since many other prediction structures such as value predictors [12] and prefetch address predictors[11] are organized similarly to branch predictors, an interesting future effort will be to apply strategies found in this paper to these structures.

References

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

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architecture-Level Power Analysis and Optimizations. In *Proc. ISCA-27*, ISCA 2000.

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

[4] P.-Y. Chang, E. Hao, and Y. N. Patt. Alternative implementations of hybrid branch predictors. In *Proc. Micro-28*, pages 252–57, Dec. 1995.

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

[6] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, pages 11–16, Oct. 28, 1996.

[7] Z. Hu, P. Juang, K. Skadron, D. Clark, and M. Martonosi. Applying decay strategies to branch predictors for leakage energy savings. Tech. Report CS-2001-24, Univ. of Virginia.

[8] J. A. Butts and G. Sohi. A Static Power Model for Architects. In *Proc. Micro-33*, Dec. 2000.

[9] D. A. Jiménez, S. W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. In *Proc. Micro-33*, pages 67–77, Dec. 2000.

[10] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *Proc. ISCA-28*, July 2001.

[11] A. Lai, C. Fide, and B. Falsafi. Dead-Block Prediction and Dead-Block Correlating Prefetchers. In *Proc. ISCA-28*, July 2001.

[12] M. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proc. ASPLOS-VII*, pages 138–47, Oct. 1996.

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

[14] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan. Power issues related to branch prediction. In *Proc. HPCA-8*, pages 233–44, Feb. 2002.

[15] K. Skadron, T. Abdelzaher, and M. R. Stan. Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management. In *Proc. HPCA-8*, pages 17–28, Feb. 2002.

[16] J. E. Smith. A study of branch prediction strategies. In *Proc. ISCA-8*, pages 135–48, May 1981.

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

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

[19] S.-H. Yang et al. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches. In *Proc. HPCA-7*, 2001.