Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power

Stefanos Kaxiras
Circuits and Systems Research Lab
Agere Systems
kaxiras@agere.com

Zhigang Hu, Margaret Martonosi
Department of Electrical Engineering
Princeton University
hzg,mrm@ee.princeton.edu

Abstract

Power dissipation is increasingly important in CPUs ranging from those intended for mobile use, all the way up to high-performance processors for high-end servers. While the bulk of the power dissipated is dynamic switching power, leakage power is also beginning to be a concern. Chipmakers expect that in future chip generations, leakage's proportion of total chip power will increase significantly.

This paper examines methods for reducing leakage power within the cache memories of the CPU. Because caches comprise much of a CPU chip's area and transistor counts, they are reasonable targets for attacking leakage. We discuss policies and implementations for reducing cache leakage by invalidating and "turning off" cache lines when they hold data not likely to be reused. In particular, our approach is targeted at the generational nature of cache line usage. That is, cache lines typically have a flurry of frequent use when first brought into the cache, and then have a period of "dead time" before they are evicted. By devising effective, low-power ways of deducing dead time, our results show that in many cases we can reduce L1 cache leakage energy by 4x in SPEC2000 applications without impacting performance. Because our decay-based *techniques have notions of competitive on-line algorithms at their roots, their energy usage can be theoretically bounded at within a factor of two of the optimal oracle-based policy. We also examine* adaptive *decay-based policies that make energy-minimizing policy choices on a per-application basis by choosing appropriate decay intervals individually for each cache line. Our proposed adaptive policies effectively reduce L1 cache leakage energy by 5x for the SPEC2000 with only negligible degradations in performance.*

1 Introduction

Power dissipation is an increasingly pressing problem in high-performance CPUs. Although power used to mainly be a concern in battery-operated devices, thermal, reliability and environmental concerns are all driving an increased awareness of power issues in desktops and servers. Most power dissipation in CMOS CPUs is dynamic power dissipation, which arises due to signal transitions. In upcoming chip generations, however, leakage power (also known as static power) will become increasingly significant. Because leakage current flows from every transistor that is powered on, leakage power characteristics are different from dynamic power, which only arises when signals transition. As such, leakage power warrants new approaches for managing it.

This paper explores options for reducing leakage power by proactively discarding items from the cache, marking the lines invalid, and then putting the cache lines "to sleep" in a way that dramatically reduces their leakage current. Our policies for turning lines off are based on generational aspects of cache line usage [33]. Namely, cache lines typically see a flurry of use when first brought in, and then a period of dead time between their last access and the point where a new data item is brought into that cache location. Turning off the cache line during this dead period can reduce leakage, without introducing any additional cache misses and without hurting performance.

Contributions: We propose several policies for determining when to turn a cache line off. We begin with a time-based strategy, which we call *cache decay*, that turns a cache line off if a pre-set number of cycles have elapsed since its last access. This time-based strategy has the nice property that its worst-case energy behavior can be bounded using theories from competitive algorithms [17, 25]. It results in roughly 70% reduction in L1 data cache leakage energy. We also study adaptive variants of this approach, which seek to improve average case performance by adaptively varying the decay interval as the program runs. These adaptive approaches use an adaptation policy that approximates chip energy tradeoffs; as such, they automatically approach the best-case operating points in terms of leakage energy. While the time-based strategies have boundable worst-case behavior and quite good average-case behavior, they still miss out on further opportunities for turning off leakage power during idle times that are shorter than their decay interval. We present preliminary evaluations of profiling-based techniques for addressing some of these cases. In addition, the paper also explores cache decay techniques for multi-level or multiprogrammed hierarchies, and discusses the interactions of these techniques with other aspects of hierarchy design such as cache consistency.

Overall this paper examines leakage power in data caches with an eye towards managing it based on boundable techniques and self-tuning adaptive mechanisms. With the increasing importance of leakage power in upcoming generations of CPUs, and the increasing size of on-chip caches, we feel that these techniques will grow in significance over the next decade.

The structure of the paper is as follows. Section 2 gives an overview of our approach, with idealized data indicating cache decay's promise. Section 3 discusses our experimental methodology, simulator, benchmarks, and the energy estimations we use to evaluate our ideas. Section 4 discusses cache decay policies and implementations, including adaptive variants of our basic scheme. Section 5 looks into multi-level cache hierarchies and multiprogramming issues which can alter the basic generational characteristics of the programs. In Section 6, we examine profiling-based

policies for further reducing leakage energy. Finally, Section 7 touches on a number of issues that arise when considering implementing cache decay, and Section 8 offers our conclusions.

2 Problem Overview

2.1 Power Background

As CPU chips are more densely packed with transistors, and as clock frequencies increase, power density on modern CPUs has increased exponentially in recent years. Although power has traditionally mainly been a worry for mobile and portable devices, it is now becoming a concern in even the desktop and server domains. In CMOS circuits, the dominant form of power dissipation is "dynamic" or "switching" power. Dynamic power is proportional to the square of the supply voltage; for that reason, it has been common to reduce supply voltage to improve both performance and power. While effective, this optimization often has the side effect of increasing the amount of "static" or "leakage" power that a CMOS circuit dissipates. Static power is so-named because it is dissipated constantly, not simply on wire transitions. Static power is a function of the circuit area, the fabrication technology, and the circuit design style. In current chips, static power represents about 2-5% of power dissipation (or even higher [14]), but it is expected to grow exponentially in upcoming generations [2, 12, 27].

2.2 Leakage Power and Cache Generations

Because caches comprise much of the area in current and future microprocessors, it makes sense to target them when developing leakage-reducing strategies. Recent work by Powell et al. has shown that transistor structures can be devised which limit static leakage power by banking the cache and providing "sleep" transistors which dramatically reduce leakage current by gating off the V_{dd} current [24, 34].

Our work exploits these sleep transistors at a finer granularity: individual cache lines. In particular, a basic premise of our work is that, surprisingly often, cache lines are storing items that will not be used again. Therefore, any static power dissipated on behalf of these cache items is wasted. We aim to reduce the power wasted on dead items in the cache, without significantly worsening either program performance or dynamic power dissipation.

Figure 1 depicts a stream of references to a particular cache line. One can break this reference stream into generations. Each generation is comprised of a series of references to the cache line. Using the terminology from [33], the i-th generation begins immediately after the i-th miss to that cache line, when a new memory line is brought into the cache frame. This generation ends when this line is replaced and a new one is brought into the cache frame. Generations begin with zero or more cache hits. Following the last reference before eviction, the generation is said to have entered its dead time. At this point, this generation has no further successful uses of the items in the cache line, so the line is said to be dead. There is considerable prior evidence that dead cache lines comprise a significant part of the cache. For example, Wood, Hill, and Kessler showed that for their benchmark suite dead time was typically at least 30% on average [33]. Similarly, Burger et al. showed that most of the data in a cache will not be used in the future [6]. They found cache "efficiencies" (their term for fraction of data that will be a read hit in the future before any evictions or writes) to be around 20% on average for their benchmarks. Most interestingly, they noted that fraction of dead time gets *worse* with higher miss rates, since lines spend more of their time about to be evicted.

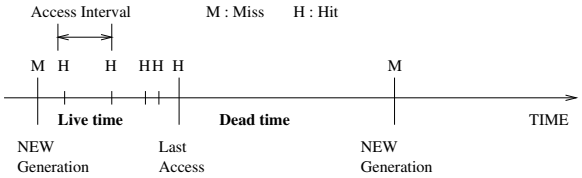

Figure 1. Cache generations in a reference stream.

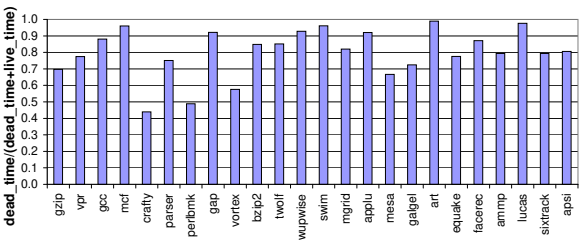

Figure 2. Fraction of time cached data are "dead."

Our goal is to exploit dead periods, particularly long ones, and to be able to turn off the cache lines during them. This approach reduces leakage power dissipated by the cache storing data items that are no longer useful.

2.3 Potential Benefits

To motivate the potential of our approach, we start by presenting an idealized study of its advantages. Here, we have run simulations using an "oracle" predictor of when dead time starts in each cache line. That is, we note when a cache item has had its last successful hit, before the cache miss that begins the next generation. We imagine, in this section only, that we can identify these dead periods with 100% accuracy and eliminate cache leakage during the dead periods.

Figure 2 illustrates the fraction of dead time we measured for a 32KB level-one data cache on our benchmark collection. This is the total fraction of time cache lines spend in their dead period.[1] We only count complete generations that end with a miss in the cache frame. The average across the benchmark suite is quite high: around 65% for integer benchmarks and even higher (80%) for FP benchmarks. Consider next an oracle predictor which knows precisely when a cache line becomes dead. With it, we could turn the cache line off with zero impact on cache miss rate or program performance. Such an oracle predictor would allow us to save power directly proportional to the shutoff ratio. If on average, 65% of the cache is shut off, and if we can implement this shutoff with negligible overhead power, then we can cut cache leakage power by one half or more.

Note that this oracle prediction is not necessarily an upper bound on the leakage power improvements to be offered by putting cache lines to sleep. Rather, the oracle predictor offers the best possible leakage power improvements *subject to the constraint that cache misses do not increase*. There may be cases where even though a line is live (i.e., it will be referenced again) the reuse will be far into the future. In such cases, it may be power-optimal to shut off the cache line early, mark it as invalid, and accept a moderate increase in the number of cache misses. Later sections will offer more realistic policies for managing these tradeoffs. On the other hand, real world attempts to put cache lines to sleep will also

[1] We sum the dead periods and the live periods of all the generations we encounter and we compute the ratio $dead/(dead + live)$. We do not compute individual dead ratios per generation and then average them, as this would skew the results towards short generations.

Processor Core	
Instruction Window	80-RUU, 40-LSQ
Issue width	4 instructions per cycle
Functional Units	4 IntALU,1 IntMult/Div, 4 FPALU,1 FPMult/Div, 2 MemPorts
Memory Hierarchy	
L1 Dcache Size	32KB, 1-way, 32B blocks, WB
L1 Icache Size	32KB, 1-way, 32B blocks, WB
L2	Unified, 1MB, 8-way LRU, 64B blocks,6-cycle latency, WB
Memory	100 cycles
TLB Size	128-entry, 30-cycle miss penalty

Table 1. Configuration of Simulated Processor

incur some small amounts of overhead power as we also discuss in the following sections.

3 Methodology and Modeling

3.1 Simulator

Simulations in this paper are based on the SimpleScalar framework [5]. Our model processor has sizing parameters that closely resemble Alpha 21264 [9], but without a clustered organization. The main processor and memory hierarchy parameters are shown in Table 1.

3.2 Benchmarks

We evaluate our results using benchmarks from the SPEC CPU2000 [29] and MediaBench suites [22]. The MediaBench applications help us demonstrate the utility of cache decay for applications with significant streaming data. The benchmarks are compiled for the Alpha instruction set using the Compaq Alpha compiler with SPEC *peak* settings. For each program, we follow the recommendation in [26], but skip a minimum of 1 billion instructions. We then simulate 500M instructions using the reference input set.

3.3 Evaluating Power Tradeoffs

A basic premise of our evaluations is to measure the static power saved by turning off portions of the cache, and then compare it to the extra dynamic power dissipated in our method. Our method dissipates extra dynamic power in two main ways. First, we introduce counter hardware to support our decay policy decisions, so we need to account for the dynamic power of these counters in our evaluations. Second, our method can dissipate extra dynamic power in cases where our decay policy introduces additional L1 cache misses not present in the original reference stream. These L1 misses translate to extra L2 reads and sometimes also extra writebacks. Turning off a dirty line results in an early writeback which is extraneous only if paired with an extra miss. For the rest of this paper, when we discuss extra misses we implicitly include associated extra writebacks.

Since both leakage and dynamic power values vary heavily with different designs and fabrication processes, it is difficult to nail down specific values for evaluation purposes. Rather, in this paper we focus on ratios of values. In this section, we describe our rationale for the range of ratio values we focus on. Later sections present our results for different ratios within this range.

A key energy metric in our study is "normalized cache leakage energy". This refers to a ratio of the energy of the L1 with cache decay policies, versus the original L1 cache leakage energy. The numerator in this relationship sums three terms. The first term is the improved leakage energy resulting from our policies. The second term is energy from counter maintenance or other overhead hardware for cache decay policies. The third term is extra dynamic energy incurred if cache decay introduces extra L1 misses that result in extra L2 cache accesses (reads and writebacks).

Dividing through by original cache leakage energy, we can use weighting factors that relate the dynamic energy of extra L2 accesses and extra counters, to the original cache leakage energy per cycle. Thus, the normalized cache leakage energy after versus before our improvements can be represented as the sum of three terms: $ActiveRatio + (Ovhd : leak)(OvhdActivity) + (L2Access : leak)(extraL2Accesses)$. ActiveRatio is the average fraction of the cache bits, tag or data, that are powered on. *Ovhd:leak* is the ratio of the cost of counter accesses in our cache decay method relative to the leakage energy. This multiplied by overhead activity (*OvhdActivity*) gives a relative sense of overhead energy in the system. The *L2Access:leak* ratio relates dynamic energy due to an additional miss (or writeback) to a single clock cycle of static leakage energy in the L1 cache. Multiplying this by the number of extra L2 accesses induced by cache delay gives the dynamic cost induced. By exploring different plausible values for the two key ratios, we present the benefits of cache decay somewhat independently of fabrication details.

3.4 Relating Dynamic and Static Energy Costs

Considering appropriate ratios is fundamental in evaluating our policies. We focus here on the *L2Access:leak* ratio. We defer policy counter overheads to Section 4 where implementations are covered.

We wish to turn off cache lines as often as possible in order to save leakage power. We balance this, however, against a desire to avoid increasing the miss rate of the L1 cache. Increasing the miss rate of the L1 cache has several power implications. First and most directly, it causes dynamic power dissipation due to an access to the L2 cache, and possible additional accesses down the memory hierarchy. Second, a L1 cache miss may force dependent instructions to stall, interfering with smooth pipeline operation and dissipating extra power. Third and finally, the additional L1 cache miss may cause the program to run for extra cycles, and these extra cycles will also lead to extra power being dissipated.

We encapsulate the energy dissipated due to an extra miss into a single ratio called *L2Access:leak*. The predominant effect to model is the amount of dynamic power dissipated in the level-two cache and beyond, due to the level-one cache miss. Additional power due to stalls and extra program cycles is minimal. Benchmarks see very few cycles of increased runtime ($< 0.7\%$) due to the increased misses for the decay policies we consider. In fact, in some situations, some benchmarks actually run slightly faster with cache decay techniques. This is because writebacks occur eagerly on cache decays, and so are less likely to stall the processor later on [10].

To model the ratio of dynamic L2 access energy compared to static L1 leakage per cycle, we first refer to recent work which estimates dynamic energy per L2 cache access in the range of 3-5nJ per access for L2 caches of the size we consider (1MB) [16]. We then compared this data to industry data by back-calculating energy per cache access for Alpha 21164's 96KB S-cache; it is roughly 10nJ per access for a 300MHz fabricated in a 0.5μ process [3]. Although the S-cache is about one-tenth the capacity of the L2 caches we consider, our back-calculation led to a higher energy estimate. First, we note that banking strategies typically employed in large caches lessen the degree by which energy-per-access scales

with size. Second, the higher 0.5μ feature size used in this older design would lead to larger capacitance and higher energy per access. Our main validation goal was to check that data given by the analytic models are plausible; our results in later sections are plotted for ratios varying widely enough to absorb significant error in these calculations.

The denominator of the *L2Access:leak* relates to the leakage energy dissipated by the L1 data cache. Again, we collected this data from several methods and compared. From the low-V_t data given in Table 2 of [34], one can calculate that the leakage energy per cycle for a 32KB cache will be roughly 0.45nJ. A simple aggregate calculation from industry data helps us validate this. Namely, using leakage power of roughly 2-5% of current CPU power dissipation, L1 cache is roughly 10-20% of that leakage [2], and CPU power dissipations are around 75W. This places L1 leakage energy at roughly 0.3nJ per cycle. Again, both methods of calculating this data give results within the same order-of-magnitude.

Dividing the 4nJ dynamic energy per access estimate by the .45nJ static leakage per cycle estimate, we get a ratio of 8.9 relating extra miss power to static leakage per cycle. Clearly, these estimates will vary widely with design style and fabrication technology though. In the future, leakage energy is expected to increase dramatically, which will also impact this relationship. To account for all these factors, our energy results are plotted for several *L2Access:leak* ratios varying over a wide range (5 to 100). Our results are conservative in the sense that high leakage in future technologies will tend to decrease this ratio. If that happens, it will only improve on the results we present in this paper.

4 Time-based Leakage Control: Cache Decay

We now examine possible policies for guiding how to use a mechanism that can reduce cache leakage by turning off individual cache lines. A key aspect of these policies is the desire to balance the potential for saving leakage energy (by turning lines off) against the potential for incurring extra level-two cache accesses (if we introduce extra misses by turning lines off prematurely). We wish to either deduce immediately at a reference point that the cache line is now worth turning off, or else infer this fact by watching its behavior over time, deducing when no further accesses are likely to arise, and therefore turning the line off. This section focuses on the latter case, which we refer to as time-based cache decay.

With oracle knowledge of reference patterns, Figure 2 demonstrated that the leakage energy to be saved would be significant. The question is: can we develop policies that come acceptably close to this oracle? In fact, this question can be approached by relating it to the theoretical area of competitive algorithms [19]. Competitive algorithms make cost/benefit decisions online (i.e., without oracle knowledge of the future) that offer benefits within a constant factor of an optimal offline (i.e., oracle-based) algorithm. A body of computer systems work has previously successfully applied such strategies to problems including superpage promotion for TLB performance, prefetching and multiprocessor synchronization [17], [25].

A generic policy for competitive algorithms is to take action at a point in time where the extra cost we have incurred so far by waiting is precisely equal to the extra cost we might incur if we act but guess wrong. Such a policy, it has been shown, leads to worst case cost that is within a factor of two of the offline optimal algorithm.[2]

[2] [25] includes a helpful example: the ski rent-vs.-buy problem. For example, if ski rental charges are $40 per day, and skis cost $400 to buy, then online approaches suggest that a beginning skier (who doesn't know whether they will enjoy skiing or not) would be wise to rent skis 10 times before buying. This equalizes the rental cost to the purchase cost, bounding total cost at two times the optimal offline approach.

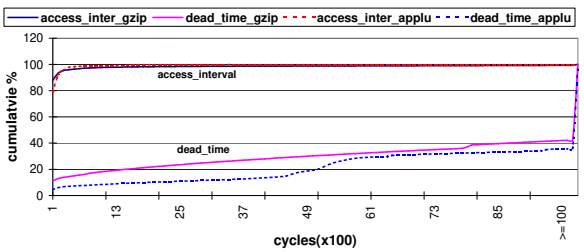

Figure 3. Cumulative distribution of Access Interval and Dead Time for gzip and applu.

For example, in the case of our cache decay policy we are trying to determine when to turn a cache line off. The longer we wait, the higher the leakage energy dissipated. On the other hand, if we prematurely turn off a line that may still have hits, then we inject extra misses which incur dynamic power for L2 cache accesses. Competitive algorithms point us towards a solution: we could leave each cache line turned on until the static energy it has dissipated since its last access is precisely equal to the dynamic energy that would be dissipated if turning the line off induces an extra miss. With such a policy, we could guarantee that the energy used would be within a factor of two of that used by the optimal offline policy shown in Figure 2.

As calculated in Section 3 the dynamic energy required for a single L2 access is roughly 9 times as large as the static leakage energy dissipated by whole L1 data cache. If we consider just one line from the L1 cache, then that ratio gets multiplied by 1024, since the cache we are studying has 1024 lines. This analysis suggests that to come within a factor of two of the oracle-policy (worst-case) we should leave cache lines turned on until they have gone roughly 10,000 cycles without an access. At that point, we should turn them off. Since the *L2Access:leak* ratio varies so heavily with design and fabrication factors, we consider a wider range of decay intervals, from 1K to 512K cycles, to explore the design space thoroughly.

The optimality of this oracle-based policy applies to the case where no additional cache misses are allowed to be added. In cases of very glacial reuse, however, it may be energy-beneficial to turn off a cache line, mark its contents invalid, and incur an L2 cache miss later, rather than to hold contents in L1 and incur leakage power for a long time period.

For the online approach (and its bound) to be of practical interest, the wait times before turning a cache line off must be short enough to be seen in real-life. That is the average dead times (Figure 1) seen in real programs must be long enough to allow the lines to be turned off a useful amount of the time. Therefore, we wish to characterize the cache dead times typically seen in applications, in order to gauge what sorts of decay intervals may be practical. Figure 3 shows cumulative distributions of access intervals and dead times for gzip (dotted lines) and applu (solid lines). The last point on the horizontal axis graph represents the tail of the distributions beyond that point. We use the term *access interval* to refer to the time between any two accesses during the live-time of a cache generation (see Figure 1). Dead time refers to the time between the last hit to an item in cache, and when it is actually evicted. Our experiments show that across the benchmark suite, there are a sizable fraction of dead times greater than 10,000 cycles. Thus, the

time range suggested by the online policy turns out to be one of significant practical promise.

Figure 3 also highlights the fact that there is a huge difference between average access interval and average dead time. For gzip, the average access interval during live time is 458 cycles while the average dead time is nearly 38,243 cycles. For applu, the results are similar: 181 cycles per access interval and 14,984 cycles per dead time. This suggests to us that dead times are not only long, but that they may also be moderately easy to identify, since we will be able to notice when the flurry of short access interval references is over.

Based on these observations, this section focuses on time-based techniques in which cache decay intervals are set between 1K and 512K cycles for the level-one cache. These intervals span broadly over the range suggested by both competitive algorithms and the dead time distributions. The following subsection details a particular way of implementing a time-based policy with a single fixed decay interval. Section 4.3 refines this approach to consider an adaptive policy whose decay interval automatically adjusts to application behavior.

4.1 Hardware Implementations of Cache Decay

To switch off a cache line we use the *gated* V_{dd} technique developed by Yang et al. [24]. The idea in this technique is to insert a "sleep" transistor between the ground (or supply) and the SRAM cells of the cache line. The stacking effect [7] of this transistor when it is off reduces by orders of magnitude the leakage current of the SRAM cell transistors to the point that leakage power of the cache line can be considered negligible. According to [24] a specific implementation of the gated V_{dd} transistor (NMOS gated V_{dd}, dual V_t, wide, with charge pump) results in minimal impact in access latency but with a 5% area penalty. We assume this implementation of the gated V_{dd} technique throughout this paper.

One way to represent recency of a cache line's access is via a binary counter associated with the cache line. Each time the cache line is accessed the counter is reset to its initial value. The counter is incremented periodically at fixed time intervals. If no accesses to the cache line occur and the counter saturates to its maximum count (signifying that the decay interval has elapsed) it switches off power to the corresponding cache line.

Our competitive algorithm bound and the dead time distributions both indicate that decay intervals should be in the range of tens of thousands of cycles. Such large decay intervals make it impractical for the counters to count cycles—too many bits would be required. Instead, it is necessary for the counters to tick at a much coarser level. Our solution is to utilize a hierarchical counter mechanism where a single global cycle counter is set up to provide the ticks for smaller cache-line counters (as shown in Figure 4).

Our simulations show that an infrequently-ticking two-bit counter per cache line provides sufficient resolution and produces the same results as a larger counter with the same effective decay interval. If it takes four ticks of the 2-bit counter to decay a cache line (figure 4), the resulting decay interval is—on average—3.5 times the period of the global counter.

In our power evaluations, we assume that the global counter will come for free, since many processors already contain various cycle counters for the operating system or for performance counting [8, 13, 36]. If such counters are not available, a simple N-bit binary ripple counter could be built with 40N + 20 transistors, of which few would transition each cycle.

To minimize state transitions in the local 2-bit cache-line counters and thus minimize dynamic power consumption we use Gray coding so only one bit changes state at any time. Furthermore,

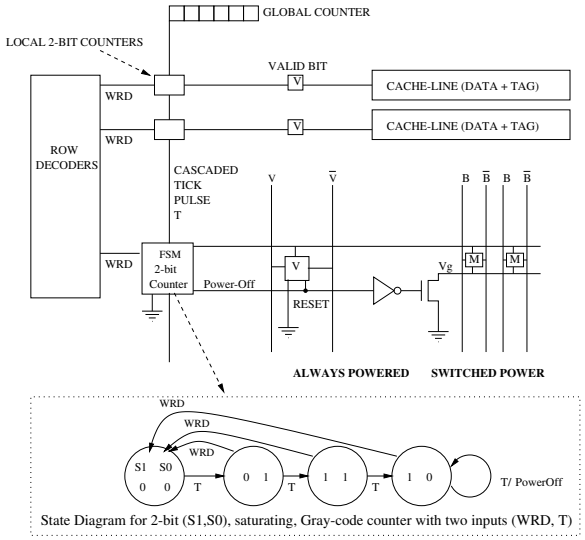

Figure 4. Hierarchical counters

to simplify the counters and minimize transistor count we chose to implement them asynchronously. Each cache line contains circuitry to implement the state machine depicted in Figure 4. The two inputs to the local counters, the global tick signal T generated by the global counter and the cache-line access signal WRD, are well behaved so there are no meta-stability problems. The output signal Power-Off, controls the *gated* V_{dd} transistor and turns off power when asserted. To avoid the possibility of a burst of write-backs with every global tick signal (if multiple dirty lines decay simultaneously) the tick signal is *cascaded* from one local counter to the next with a one-cycle latency. This does not affect results but it spreads writeacks in time.

All local counters change value with every T pulse. However, this happens at very coarse intervals (equal to the period of the global counter). Resetting a local counter with an access to a cache line is not a cause of concern either. If the cache line is heavily accessed the counter has no opportunity to change from its initial value so resetting it does not expend any dynamic power (none of the counter's transistors switch). The cases where power is consumed are accesses to cache lines that have been idle for at least one period of the global counter. Our simulation results indicate that over all 1024 2-bit counters used in our scheme, there are 0.2 bit transitions per cycle on average. Modeling each counter as a 2-bit register in Wattch [4], we estimate roughly .1pJ per access. Therefore, at an average of 0.02pJ per cycle, the power expended by all 1024 of these infrequently-ticking counters is roughly 4 orders of magnitude lower than the cache leakage energy which we estimate at 0.45nJ per cycle. For this reason, our power analysis will consider this counter overhead to be negligible from this point forward.

Switching off power to a cache line has important implications for the rest of the cache circuitry. In particular, the first access to a powered-off cache line should: (i) result in a miss (since data and tag might be corrupted without power) (ii) reset the counter and restore power to the cache line and (iii) delay an appropriate amount of time until the cache-line circuits stabilize after power is restored. To satisfy these requirements we use the Valid bit of the cache line as part of the decay mechanism (Figure 4). First, the valid bit is always powered. Second, we add a reset capability to the valid bit so the Power-Off signal can clear it. Thus, the first access to a power-off cache line always results in a miss regardless

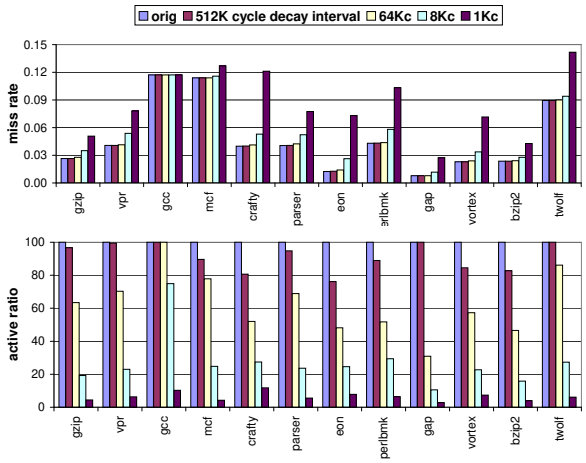

Figure 5. Miss rate and active ratio of a 32KB decay cache for SPECint 2000.

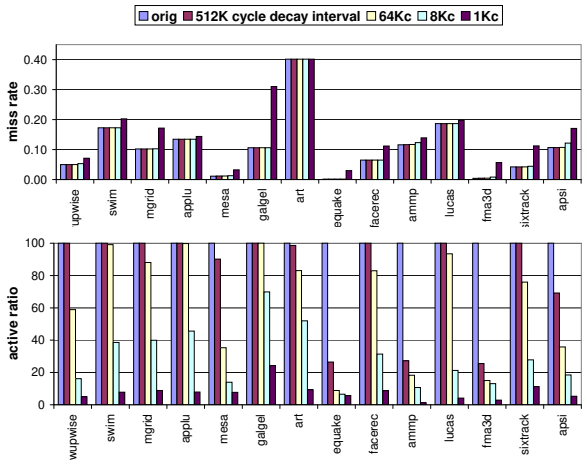

Figure 6. Miss rate and active ratio of a 32KB decay cache for SPECfp 2000.

of the contents of the tag. Since satisfying this miss from the lower memory hierarchy is the only way to restore the valid bit, a newly-powered cache line will have enough time to stabilize. In addition, no other access (to this cache line) can read the possibly corrupted data in the interim.

Analog implementation: Another way to represent the recency of a cache line's access is via charge stored on a capacitor. Each time the cache line is accessed, the capacitor is grounded. In the common case of a frequently-accessed cache line, the capacitor will be discharged. Over time, the capacitor is charged through a resistor connected to the supply voltage (V_{dd}). Once the charge reaches a sufficiently high level, a voltage comparator detects it, asserts the Power-Off signal and switches off power to the corresponding cache line. Although the RC time constant cannot be changed (it is determined by the fabricated size of the capacitor and resistor) the bias of the voltage comparator can be adjusted to different temporal access patterns. An analog implementation is inherently noise sensitive and can change state asynchronously with the remainder of the digital circuitry. Some method of synchronously sampling the voltage comparator must be used to avoid meta-stability. Since an analog implementation can be fabricated to mimic the digital implementation, the rest of this paper focuses on the latter.

4.2 Results

We now present experimental results for the time-based decay policy based on binary counters described in Section 4.1. First Figures 5 and 6 plot the active ratio and miss rate as a function of cache decay interval for a collection of integer and floating point programs. In each graph, each application has five bars. In the active ratio graph, the first bar is the active ratio for a traditional 32KB L1 data cache. Since all the cache is turned on all the time, the active ratio is 100%. Furthermore, we have determined that our benchmark programs touch the entirety of the standard caches for the duration of execution (active ratio over 99%). The other bars show the active ratio (average number of cache bits turned on) for decay intervals ranging from 512K cycles down to 1K cycles. Clearly, shorter decay intervals dramatically reduce the active ratio, and thus reduce leakage energy in the L1 data cache, but that is only part of the story. The miss rate graphs show how increasingly aggressive decay intervals affect the programs' miss rates.

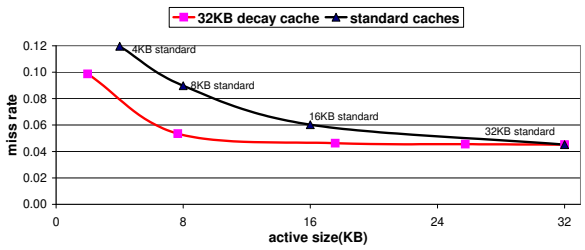

Figure 7. Comparison of standard 8KB, 16KB, and 32KB caches to a fixed-size 32KB decay cache with varying cache decay intervals. The different points in the decay curve represent different decay intervals. From left to right, they are: 1Kcycles, 8Kcycles, 64Kcycles, and 512Kcycles. Active size and miss rate are geometric means over all SPEC 2000 benchmarks.

Figure 7 plots similar data averaged over all the benchmarks. The upper curve corresponds to a traditional cache in which we vary the cache size and see the miss rate change. In a traditional cache, active size is just the cache size. The lower curve in this graph corresponds to a decay cache whose full size is fixed at 32KB. Although the decay cache's full size is fixed, we can vary the decay interval and see how this influences the active size. (This is the apparent size based on the number of cache lines turned on.) Starting at the 16KB traditional cache and dropping downwards, one sees that the decay cache with the same active size has much better miss rate characteristics.

Figure 8 shows the normalized cache leakage energy metric for the integer and floating point benchmarks. In this graph, we assume that *L2Access:leak* ratio is equal to 10 as discussed in Section 3. We normalize to the leakage energy dissipated by the original 32KB L1 data cache with no decay scheme in use. Although the behaviors of each benchmark are unique, the general trend is that a decay interval of 8K cycles shows the best energy improvements. This is quite close to the roughly 10Kcycle interval suggested for worst-case bounding by the theoretical analysis. For the integer benchmarks, all of the decay intervals — including even 1Kcycle for some —result in net improvements. For the floating point benchmarks, 8Kcycle is also the best decay interval. All but one of the floating point benchmarks are improved by cache decay techniques for the full decay-interval range.

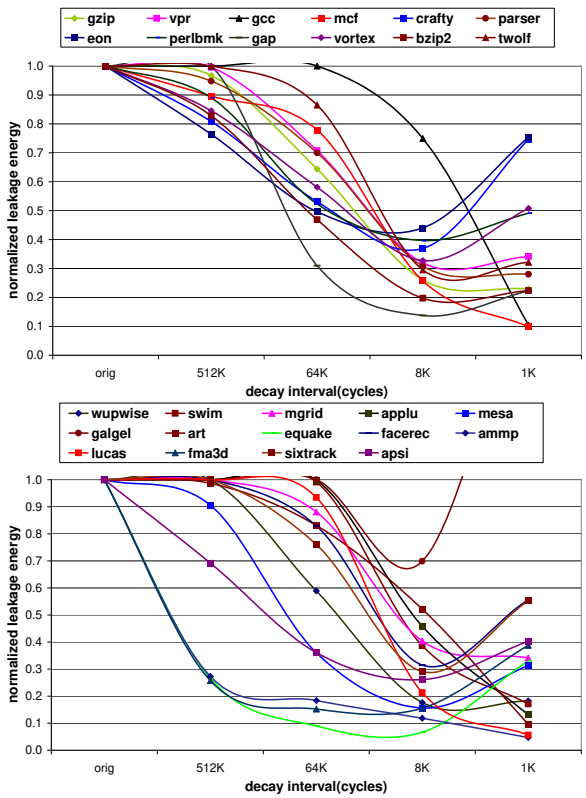

Figure 8. Normalized cache leakage energy for an *L2Access:leak* **ratio of 10. This metric takes into account both static energy savings and dynamic energy overhead. Top graph shows SPECint2000; bottom graph shows SPECfp2000.**

We also wanted to explore the sensitivity of our results to different ratios of dynamic L2 energy versus static L1 leakage. Figure 9 plots three curves of normalized cache leakage energy. Each curve represents the average of all the SPEC benchmarks. The curves correspond to *L2Access:leak* ratios of 5, 10, 20, and 100. All of the ratios show significant leakage improvements, with smaller ratios being especially favorable. When the *L2Access:leak* ratio equals 100, then small decay intervals (less than 8K cycles) are detrimental to both performance and power. This is because short decay intervals may induce extra cache misses by turning off cache lines prematurely; this effect is particularly bad when *L2Access:leak* is 100 because high ratios mean that the added energy cost of additional L2 misses is quite high.

To assess these results one needs to take into consideration the impact on performance. If cache decay slows down execution

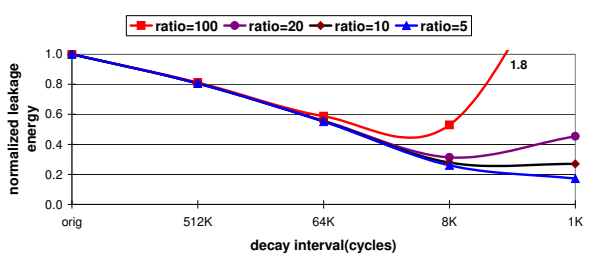

Figure 9. Normalized L1 data cache leakage energy averaged across SPEC suite for various *L2Access:leak* **ratios.**

because of the increased miss rate then its power advantage diminishes. For the decay scenarios we consider, not only we do not observe any slow-down but in fact we observe a very slight speed up in some cases, which we attribute to eager writebacks [10]. Beyond this point, however, cache decay is bound to slow down execution. For our simulated configuration, performance impact is negligible except for very small decay intervals: the 8Kcycle interval—which yields very low normalized leakage energy (Figures 8 and 9)—decreases IPC by 0.1% while the 1Kcycle interval—which we do not expect to be used widely—decreases IPC by 0.7%. Less aggressive processors might suffer comparably more from increased miss rates, which would make very small decay intervals undesirable.

In addition to the SPEC applications graphed here, we have also done some initial studies with MediaBench applications [22]. The results are even more successful than those presented here partly due to the generally poor reuse seen in streaming applications; MediaBench applications can make use of very aggressive decay policies. Since the working set of MediaBench can, however, be quite small (for gsm, only about 50% of the L1 data cache lines are *ever* touched) we do not present the results here. We have also performed preliminary sensitivity analysis on cache size and block size. Cache decay works well with a range of cache sizes but with smaller caches, smaller decay intervals are required. Smaller block sizes offer finer control, thus better behavior, but the effect is not very pronounced. Finally, we have also experimented with similar techniques on 32KB instruction caches. Across the SPEC suite, the data indicate that cache decay works at least as well in instruction caches as in data caches. An additional small benefit in this case is that instruction caches do not have any write-back traffic. These observations validate the results presented in [34].

4.3 Adaptive Variants of Time-based Decay

So far we have investigated cache decay using a single decay interval for all of the cache. We have argued that such a decay interval can be chosen considering the relative cost of a miss to leakage power in order to bound worst-case performance. However, Figure 8 shows that in order to achieve best average-case results this choice should be application-specific. Even within an application, a single decay interval is not a match for every generation: generations with shorter dead times than the decay interval are ignored, while others are penalized by the obligatory wait for the decay interval to elapse. In this section we present an adaptive decay approach that chooses decay intervals at run-time to match the behavior of individual cache lines.

Motivation for an adaptive approach: Figure 10 shows details about why a single decay interval cannot capture all the potential benefit of an oracle scheme. In this figure two bars are shown for each program: a decay bar on the left and an oracle bar on the right. Within the bar for the oracle-based approach, there are three regions. The lower region of the oracle bar corresponds to the lower region of the decay bar which is the benefit (the shut-off ratio) that comes from decaying truly dead cache lines. The two upper regions of the oracle bar represent benefit that the single-interval decay schemes of Section 4.2 cannot capture. The middle region of the oracle bar is the benefit lost while waiting for the decay interval to elapse. The upper region is lost benefit corresponding to dead periods that are shorter than the decay interval. On the other hand, the decay scheme can also mistakenly turn off *live* cache lines. Although this results in extraneous misses (*decay misses*) it also represents benefit in terms of leakage power. This effect is shown as the top region of the decay bars. For some

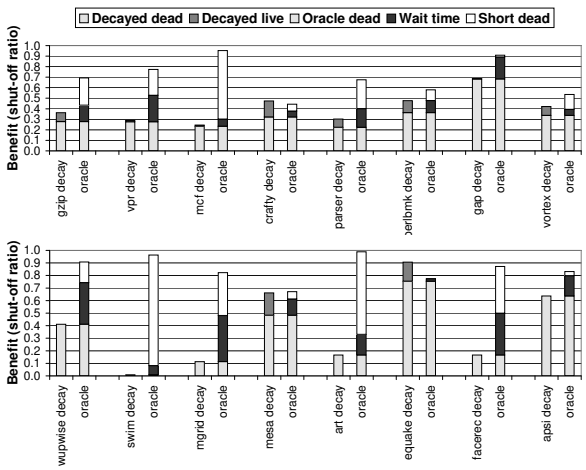

Figure 10. Lost opportunities for time-based decay (64Kcycle decay interval). SPECint2000 and SPECfp2000.

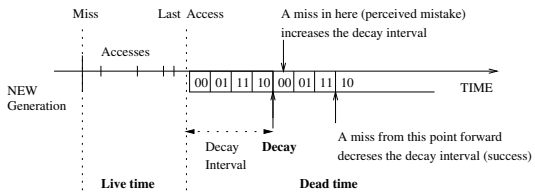

Figure 11. Adaptive decay.

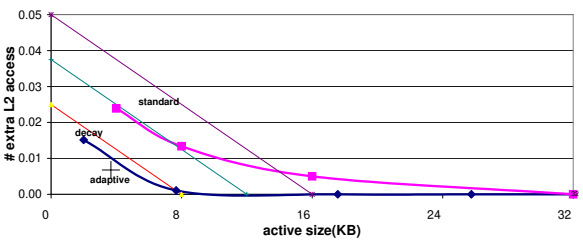

Figure 12. Effect of adaptive decay. Iso-power lines show constant power ($L2Access : leak = 10$). Results averaged over all SPEC2000.

SPECfp2000 programs the benefit from short dead periods is quite large in the oracle bars.

Implementation: An ideal decay scheme would choose automatically the best decay interval for each generation. Since this is not possible without prior knowledge of a generation's last access, we present here an adaptive approach to chose decay intervals per cache-line. In Section 6 we explore the possibility of deducing a generation's last access by other means (such as compiler techniques, prediction, and profiling).

Our adaptive scheme attempts to choose the smallest possible decay interval (out of a predefined set of intervals) individually for each cache-line. The idea is to start with a short decay interval, detect whether this was a mistake, and adjust the decay interval accordingly. A mistake in our case is to prematurely decay a cache-line and incur a decay miss. We can detect such mistakes if we leave the tags always powered-on but this is a significant price to pay (about 10% of the cache's leakage for a 32KB cache). Instead we opted for a scheme that infers possible mistakes according to how fast a miss appears after decay. We determined that this scheme works equally well or *better* than an exact scheme which dissipates tag leakage power.

The idea is to reset a line's 2-bit counter upon decay and then reuse it to gauge dead time (Figure 11). If dead time turns out to be very short (the local counter did not advance a single step) then chances are that we have made a mistake and incurred a decay-miss. But if the local counter reaches its maximum value while we are still in the dead period then chances are that this was a successful decay. Upon mistakes—misses with the counter at minimum value (00 in Figure 11)—we adjust the decay interval upwards; upon successes—misses with counter at maximum value (10)—we adjust it downwards. Misses with the counter at intermediate values (01 or 11) do not affect the decay interval.

We use exponentially increasing decay intervals similarly to Ethernet's exponential back-off collision algorithm but the set of decay intervals can be tailored to the situation. As we incur mistakes, for a cache line, we exponentially increase its decay interval. By backing-off a single step in the decay-interval progression rather than jumping to the smallest interval we introduce *hysteresis* in our algorithm.

Implementation of the adaptive decay scheme requires simple changes in the decay implementation discussed previously. We introduce a small field per cache line, called *decay speed field*, to select a decay interval. An *N*-bit field can select up to 2^N decay intervals. The decay-speed field selects different tick pulses coming from the same or different global cycle counters. This allows great flexibility in selecting the relative magnitude of the decay intervals. The value of this field is incremented whenever we incur a *perceived* decay miss and decremented on a *perceived* successful decay. We assume that higher value means longer decay interval.

Results: Figure 12 presents results of an adaptive scheme with 10 decay intervals (4-bit decay-speed field). The decay intervals range from 1K cycles to 512K cycles (the full range used in our previous experiments) and are successive powers-of-two. In the same figure we repeat results for the single-interval decay and for various standard caches. We also plot *iso-power* lines, lines on which total power dissipation remains constant (for $L2Access : leak = 10$). The adaptive decay scheme automatically converges to a point below the single-interval decay curve. This corresponds to a total power lower than the iso-power line *tangent to* the decay curve. This point has very good characteristics: significant reduction in active area and modest increase in miss ratio. Intuitively, we would expect this from the adaptive scheme since it tries to maximize benefit but also is aware of cost. This behavior is application-independent: the adaptive scheme tends to converge to points that are close or lower than the highest iso-power line tangent to the single-decay curve.

5 Changes in the Generational Behavior and Decay

In this section we examine how cache decay can be applied when multi-level cache hierarchies or multi-programming change the apparent generational behavior of cache lines. The upper level cache filters the stream that reaches lower levels, significantly changing their generational characteristics. With multiprogramming, cache conflicts introduced with context-switching tend to interrupt long generations and create more shorter generations.

5.1 Multiple Levels of Cache Hierarchy

Cache decay is likely to be useful at multiple levels of the hierarchy since it can be usefully employed in any cache in which

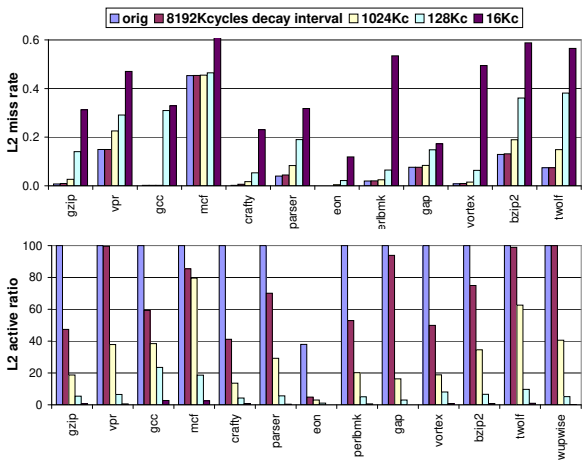

Figure 13. Miss rate and active ratio of a 1MB L2 decay cache for SPECint 2000. eon does not fully utilize the L2 cache.

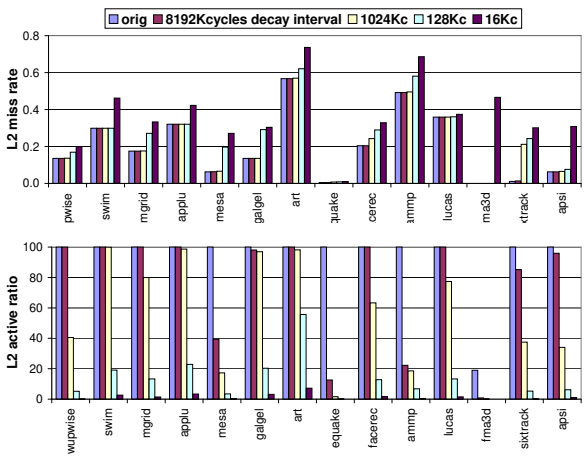

Figure 14. Miss rate and active ratio of a 1MB L2 decay cache for SPECfp 2000. fma3d does not fully utilize the L2 cache.

the active ratio is low enough to warrant line shut-offs. For several reasons, the payoff is likely to increase as one moves outward in the hierarchy. First, a level-two or level-three cache is likely to be larger than the level-one cache, and therefore will dissipate more leakage power. Second, outer levels of the cache hierarchy are likely to have longer generations with larger dead time intervals. This means they are more amenable to our time-based decay strategies. On the other hand, the energy consumed by any extra L2 misses we induce could be quite large, especially if servicing them requires going off chip.

The major difference between L1 and L2 is the filtering of the reference stream that takes place in L1 which changes the distribution of the access intervals and dead periods in L2. Our data shows that the average access interval and dead time for L2 cache are 79,490 and 2,714,980 cycles respectively. Though access intervals and dead periods become significantly larger, their relative difference remains large and this allows decay to work.

The increased access intervals and dead times suggest we should consider much larger decay intervals for the L2 compared to those in the L1. This meshes well with the competitive analysis which also points to an increase in decay interval because the cost of an induced L2 cache miss is so much higher than the cost of an induced L1 cache miss. As a simple heuristic to choose a decay interval, we note that since there is a 100-fold increase in the dead periods in L2, we will also multiply our L1 decay interval by 100. Therefore a 64Kcycle decay interval in L1 translates to decay intervals on the order of 6400K cycles in the L2.

Here, we assume that multilevel inclusion is preserved in the cache hierarchy [1]. Multilevel inclusion allows snooping on the lowest level tags only and simplifies writebacks and coherence protocols. *Inclusion bits* are used to indicate presence of a cache line in higher levels. For L2 cache lines that also reside in the L1 we can turn off only the data but not the tag. Figures 13 and 14 show miss rate and active ratio results for the 1MB L2 cache. As before, cache decay is quite effective at reducing the active ratio in the cache. Miss rates tend to be tolerable as long as one avoids very short decay intervals. (In this case, 128Kcycle is too short.)

It is natural to want to convert these miss rates and active ratios into energy estimates. This would require, however, coming up with estimates on the ratio of L2 leakage to the extra dynamic power of an induced L2 miss. This dynamic power is particularly hard to characterize since it would often require estimating power

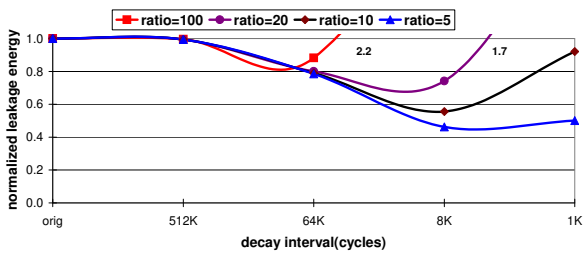

Figure 15. Normalized L1 data cache leakage energy for cache decay methods on a multiprogrammed workload.

for an off-chip access to the next level of the hierarchy. We are not that daring! Instead, we report the "breakeven" ratio. This is essentially the value of *L2Access:leak* at which this scheme would break even for the L2 cache.

In these benchmarks, breakeven *L2Access:leak* ratios for an 1Mcycle decay interval range from 71 to 155,773 with an average of 2400. For a 128Kcycle decay interval, breakeven *L2Access:leak* ratios range from 16 to 58,906 with an average of 586. The art benchmark tends to have one of the lowest breakeven ratios; this is because its average L2 access interval is very close to its average L2 dead time so cache decay is very prone to inducing extra misses.

5.2 Multiprogramming

Our prior results all focus on a single application process using all of the cache. In many situations, however, the CPU will be time-shared and thus several applications will be sharing the cache. Multiprogramming can have several different effects on the data and policies we have presented. The key questions concern the impact of multiprogramming on the cache's dead times, live times, and active ratios.

To evaluate multiprogramming's impact on L1 cache decay effectiveness, we have done some preliminary studies of cache live/dead statistics for a multiprogramming workload. The workload was constructed as follows. We collected reference traces from six benchmarks individually: gcc, gzip, mgrid, swim, vpr and wupwise. In each trace, we recorded the address referenced and the time at which each reference occurred. We then "sew" to-

gether pieces of the traces, with each benchmark getting 40ms time quanta in a round-robin rotation to approximate cache conflicts in a real system.

Multiprogramming retains the good behavior of the single-program runs. Average dead time remains roughly equal to the average dead times of the component applications. This is because the context switch interval is sufficiently coarse-grained that it does not impact many cache generations. In a workload where cache dead times are 13,860 cycles long on average, the context switch interval is many orders of magnitude larger. Thus, decay techniques remain effective for this workload. Multiprogramming also allows opportunities for more agressive decay policies such as decaying items at the end of a process time quantum.

6 Instruction-based leakage control

Time-based decay methods offer solid power improvements, but still experience leakage current during dead periods until the prescribed policy delay has elapsed. Other policy approaches could make quicker decisions about whether we are in a dead period or not. The potential additional benefit of turning off cache lines immediately after their last access (as opposed to waiting for the decay interval to elapse) is shown in Figure 10. In this figure the top two regions of the oracle bars represent the potential benefit from short dead periods and decay interval wait. Cache decay completely misses dead periods shorter than the decay interval and is penalized waiting for the decay interval. The potential benefit from short dead periods is especially pronounced in some of the floating point programs (Figure 10).

Knowing the last access to a cache line before replacement is a more difficult problem than the problem of predicting whether an instruction will miss in its next instantiation which other work has pursued for prefetching purposes [15]. Deducing whether an access is the last access to a cache line— consequently, the next access to the same cache frame would be a miss—can be approached by three methods: (i) **Compiler:** If the compiler could identify last-use for cache lines then the last access to the cache-line could also turn it off. For example, recent work has developed compiler algorithms to improve the performance of low associativity caches by marking cache lines as "evict-me-next" [31]. (ii) **Prediction:** A hardware structure could predict last hit before eviction on an access. The predictor, however, must be fairly accurate to be useful, and fairly small to avoid burning more power than it saves. These needs are likely to be contradictory. (iii) **Profiling:** If we can identify potential instructions whose access is consistently the last access before eviction then we can feed this information back to the compiler to generate annotated binaries.

To assess the feasibility of such methods we performed the following experiment to understand the "last-access before eviction" behavior of programs. For each load/store instruction we note the cache frame accessed by the instruction. We then track outcome of the next reference that accesses the same cache frame. Depending on the outcome of that next reference, we increment the `last-hit` (before eviction) or `other-hit` counter for the original noted load or store. The overall goal is to see if there are instructions which act as clear signals for last accesses. To visualize the data, Figure 16 shows the results as scatter plots for applu. Each point in the plots represents a single load or store instruction whose (x,y) coordinates are the `last-hit` and `other-hit` values (both axes in logarithmic scale.) An instruction on the horizontal axis is always a last hit (i.e., the next reference to the same cache frame is a miss). An instruction on the vertical axis is likewise always an ordinary hit and is followed by another

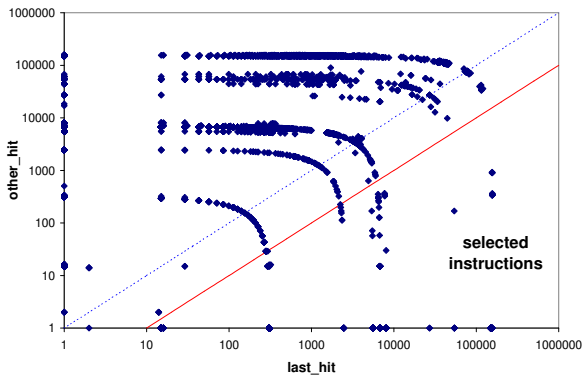

Figure 16. applu. Scatter plot of `last-hit`, `other-hit` per instruction.

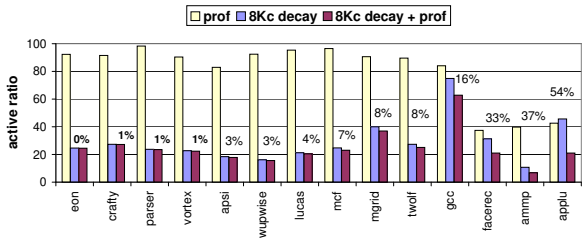

Figure 17. Active ratio for profiling, decay (8Kcycle) and profiling with decay.

hit. Instructions on or near the diagonal (dotted line) have about equal chances of being last hits or ordinary hits. In our case, the interesting instructions are located in the bottom right corner of the plots where the bulk of the numerous subsequent references turn out to be cache misses, so the instruction could be a good last-access predictor. Such instructions can be used to turn off the cache lines they access without unduly increasing miss rate, since the next access to these cache lines is probably a miss. Most integer programs (except twolf and gcc) do not have many instructions that can be used to turn off cache lines. On the other hand, some floating point codes exhibit some interesting behavior, with some instructions clearly biased towards last-access.

Based on this data, we looked into profiling-based approaches for exploiting the observed behavior. We profiled the programs using the *training input* and collected statistics (`last-hit`, `other-hit`) for the load/store instructions. Our profiling tool automatically selects the best candidates according to the following criteria: `last-hit` $> 10 *$ `other-hit` (shown as a solid line in Figure 16). We then annotate the codes and re-execute them with the *reference input*. Cache decay (8Kcycle) works on the remainder so cache lines that are not turned off by the profiled instructions are caught by the decay mechanisms. Figure 17 shows the active ratios that can be achieved by profiling alone, by 8Kcycle decay, and by profiling combined with decay. We only include programs for which profiling turns off more than 1% of the cache. The programs are sorted according to improvements over 8Kcycle decay. The improvement (decrease in active ratio) is shown as a percentage between the decay and profiling-with-decay bars. Profiling alone benefits significantly only some of the programs but in these cases its benefit is synergistic to decay.

These improvements come nearly "for free" in terms of power, since the profiling can occur only once and be amortized over

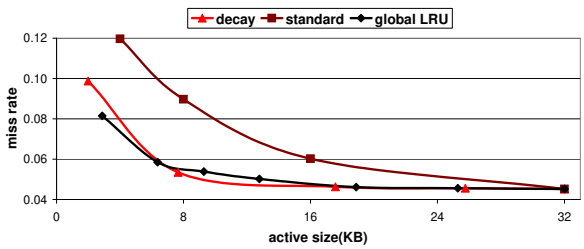

Figure 18. Global LRU decay vs. Working Set decay. Miss rates and active sizes as geometric means over all SPEC2000 programs.

many runs of the program. Furthermore, profiling *does not increase miss rate* by any perceptible level. A limitation of our simple annotation-based approach is that we classify events by program counter only and not by using any information about the address accessed. Additional addressing information might improve prediction accuracy [18] but would also be more complicated. Likewise, we considered a simple last-access predictor in hardware, but concluded that the energy expended in the predictor hardware outweighed any potential savings in many situations. On the software side, we expect that more elaborate compiler support may be useful in improving our approach [31].

7 Discussion

This section explores alternative policies to control decay, some of the interactions of the cache decay methods with other hardware structures and application of decay in other situations.

LRU Decay: Time-based decay is in essense a Working Set algorithm. Working Set and global LRU perform comparably in virtual memory paging [28] and this is also holds for cache decay. Global LRU mechanisms that have been proposed previously for cache management involve a separate structure to maintain LRU order of the cache blocks (e.g., Set-reference History Table [23], Indirect-Index Cache [11]). Such structures are likely to be expensive both in terms of transistor count and power consumption. The global LRU structure needs to be updated with every cache access thus expending dynamic power. In contrast, the local counters in the Working-Set implementation rarely switch when a cache line is accessed. To implement an LRU algorithm for cache decay we use a structure, similar to those proposed by [23] and [11], to maintain the LRU order of cache lines. Instead of controlling a decay interval, in the LRU implementation we directly control the active size of the cache, i.e., we can request half of the lines—the least recently used—to be turned off. In Figure 18 we compare the behavior of an idealized global LRU scheme with the Working Set decay and standard caches of various sizes (as in Figure 7). We control the LRU decay scheme by requesting 0% to 90% (in increments of 10%) of the cache lines to be turned off. Working Set decay shows a small advantage at the knee of the curve while LRU decay at the far left. The two schemes are very close in behavior and the decision on which one to use should be based on implementation costs.

Multiprocessor Cache Consistency: Another key issue in the realization of our cache decay mechanism is that also be usable in cache-consistent multiprocessor systems. Although we do not have quantitative results here, we feel that cache decay and multiprocessor cache consistency work well together. The key correctness issue to implement is that putting a cache line to sleep should be treated as any other cache eviction. If the line is dirty/exclusive, it should be written back to caches lower in the hierarchy or to memory. If the line is clean, then turning it off simply requires marking it as invalid. In directory-based protocols, one would also typically notify the directory that the evicting node is no longer a sharer of this line. Interestingly, cache decay may improve consistency protocol performance by purging stale info from cache (eager writebacks). Particularly in directory-based protocols, this can allow the system to save on stale invalidate traffic. From this aspect cache decay can be considered a poor man's predictor for **dynamic-self invalidation** [21, 20]. Invalidations arriving from other processors can also be exploited by cache decay methods. In particular, these invalidations can be used as an additional hint on when to turn off cache lines.

Victim Caches, Line Buffers and Stream Buffers: We also note that our cache decay schemes are orthogonal and synergistic with other "helper" caches such as victim caches or stream buffers. These other caching structures can be helpful as ways of mitigating the cache miss increases from cache decay, without as much leakage power as larger structures.

DRAM Caches: Some recent work has discussed the possibility of DRAM-based caches [32]. In such structures, there is a natural analog to the SRAM cache decay scheme we propose here. Namely, one could consider approaches in which lines targeted for shutoff do not receive refresh, and eventually the values decay away. A key point to note is that typically DRAM refresh is done on granularities larger than individual cache lines, so this strategy would need to be modified somewhat. At the extreme, one can have a DRAM cache with *no refresh*. In a decay DRAM cache, a mechanism to distinguish among decayed lines and valid lines is necessary. A simple solution is to guarantee that the valid bit, by fabrication, decays faster than the tag and data bits. Also, care must be taken not to lose any dirty data in a write-back cache. In this case, dirty lines, distinguished by the dirty bit, can be selectively refreshed, or "cleansed" with a writeback prior to decay. Savings in a decay DRAM cache include reduction in both static and refresh (dynamic) power. The refresh interval in DRAM memories is fairly large, and would correspond to decay intervals of millions of cycles in our simulations. Even for L2 caches such decay intervals do not increase miss rate significantly.

Branch Predictors: Cache decay can also be applied to other memory structures in a processor such as large branch prediction structures [35]. Decay in branch predictors has interesting implications in terms of cost. While we can expect savings for leakage power as in caches, the cost of a premature decay in a branch predictor is the possibility of a bad prediction rather than a miss. Thus, dynamic power expended in the processor must be considered in this case.

8 Conclusions

This paper has described methods to reduce cache leakage power by exploiting generational characteristics of cache-line usage. We introduce the concept of cache decay, where individual cache lines are turned off (eliminating their leakage power) when they enter a dead period—the time between the last successful access and a line's eviction. We propose several energy-efficient techniques that deduce entrance to the dead period with small error. Error in our techniques translates into extraneous cache misses and writebacks which dissipate dynamic power and harm performance. Thus, our techniques must strike a balance between leakage power saved and dynamic power induced. Our evaluations span a range of assumed ratios between dynamic and static power, in order to give both current and forward-looking predictions of

cache decay's utility.

Our basic method for cache decay is a time-based Working Set algorithm over all cache lines. A cache line is kept on as long as it is re-accessed within a time window called "decay interval." This approach is roughly equivalent to a global LRU algorithm but our proposed implementation is more efficient (in terms of transistor budget and power) than global LRU implementations proposed previously. In our implementation, a global counter provides a coarse time signal for small per-cache-line counters. Cache lines are "decayed" when a cache-line counter reaches its maximum value. This simple scheme works well for a wide range of applications, L1 and L2 cache sizes, and cache types (instruction, data). It also survives multiprogramming environments despite the increased occupancy of the cache. Compared to standard caches of various sizes, a decay cache offers better active size (for the same miss rate) or better miss rate (for the same active size) for all the cases we have examined.

Regulating a decay cache to achieve a desired balance between benefit and overhead is accomplished by adjusting the decay interval. Competitive on-line algorithm theory allows one to reason about appropriate decay intervals given a dynamic to static energy ratio. Specifically, competitive on-line algorithms teach us how to select a decay interval that bounds worst case behavior within a constant factor of an oracle scheme. To escape the burden of selecting an appropriate decay interval to optimize *average case behavior* for different situations (involving different applications, different cache architectures, and different power ratios) we propose adaptive decay algorithms that automatically converge to the desired behavior. The adaptive schemes involve selecting among a multitude of decay intervals per cache line and monitoring success (no extraneous misses) or failure (extraneous misses) for feedback.

Finally, this paper briefly explores a profile-based, rather than time-based, method to detect dead periods. By using a profile run to classify load/store instructions according to subsequent hit or miss events on the cache lines they access, one can further improve on time-based cache decay, mainly in floating point codes. Profiling captures benefit that time-based schemes miss because of their long decay intervals.

With the increasing importance of leakage power in upcoming generations of CPUs, and the increasing size of on-chip memory, cache decay can be a useful architectural tool to save power or to rearrange the power budget within a chip.

9 Acknowledgments

We would like to thank Girija Narlikar and Rae McLellan for their contributions in the initial stages of this work. Our thanks to Jim Goodman who turned our attention to adaptive decay techniques and to Alan J. Smith for pointing out LRU decay and multiprogramming. Our thanks to Nevin Heintze, Babak Falsafi, Mark Hill, Guri Sohi, Adam Butts, Erik Hallnor, and the anonymous referees for providing helpful comments.

References

[1] J. Baer and W. Wang. On the inclusion property in multi-level cache hierarchies. In *Proc. ISCA-15*, 1988.

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

[3] W. J. Bowhill et al. Circuit Implementation of a 300-MHz 64-bit Second-generation CMOS Alpha CPU. *Digital Technical Journal*, 7(1):100–118, 1995.

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

[5] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: the SimpleScalar tool set. Tech. Report TR-1308, Univ. of Wisconsin-Madison Computer Sciences Dept., July 1996.

[6] D. Burger, J. Goodman, and A. Kagi. The declining effectiveness of dynamic caching for general-purpose microprocessors. Tech. Report TR-1216, Univ. of Wisconsin-Madison Computer Sciences Dept.

[7] Z. Chen et al. Estimation of standby leakage power in CMOS circuits considering accurate modeling of transistor stacks. In *ISLPED*, 1998.

[8] J. Dean, J. Hicks, et al. Profileme: Hardware support for instruction-level profiling on out-of-order processors. In *Proc. Micro-30*, 1997.

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

[10] H.-H. Lee, G. S. Tyson, M. Farrens. Eager Writeback - a Technique for Improving Bandwidth Utilization. In *Proc. Micro-33*, Dec. 2000.

[11] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *Proc. ISCA-27*, June 2000.

[12] IBM Corp. Personal communication. November, 2000.

[13] Intel Corp. Intel architecture optimization manual.

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

[15] T. Johnson et al. Run-time Cache Bypassing. *IEEE Transactions on Computers*, 48(12), 1999.

[16] M. B. Kamble and K. Ghose. Analytical Energy Dissipation Models for Low Power Caches. In *ISLPED*, 1997.

[17] A. Karlin et al. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proc. SOSP*, 1991.

[18] S. Kaxiras and C. Young. Coherence communication prediction in shared-memory multiprocessors. In *Proc. HPCA-6*, Jan. 2000.

[19] T. Kimbrel and A. Karlin. Near-optimal parallel prefetching and caching. *SIAM Journal on computing*, 2000.

[20] A.-C. Lai and B. Falsafi. Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction. In *Proc. ISCA-27*, May 2000.

[21] A. R. Lebeck and D. A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proc. ISCA-22*, June 1995.

[22] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proc. Micro-30*, Dec. 1997.

[23] J. Peir, Y. Lee, and W. Hsu. Capturing Dynamic Memory Reference Behavior with Adaptive Cache Topology. In *Proc. ASPLOS-VIII*, Nov. 1998.

[24] M. D. Powell et al. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In *ISLPED*, 2000.

[25] T. Romer, W. Ohlrich, A. Karlin, and B. Bershad. Reducing TLB and memory overhead using online superpage promotion. In *Proc. ISCA-22*, 1995.

[26] S. Sair and M. Charney. Memory behavior of the SPEC2000 benchmark suite. Technical report, IBM, 2000.

[27] Semiconductor Industry Association. The International Technology Roadmap for Semiconductors, 1999. http://www.semichips.org.

[28] W. Stallings. *Operating Systems*. Prentice Hall, 2001.

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

[30] U.S. Environmental Protection Agency. Energy Star Program web page. http://www.epa.gov/energystar/.

[31] Z. Wang, K. S. McKinley, and A. L. Rosenberg. Improving replacement decisions in set-associative caches. Technical Report TR-01-02, University of Massachusetts, Mar. 2001. http://ali-www.cs.-umass.edu/.

[32] K. M. Wilson and K. Olukotun. Designing high bandwidth on-chip caches. In *Proc. ISCA-24*, pages 121–32, June 1997.

[33] D. A. Wood, M. D. Hill, and R. E. Kessler. A Model for Estimating Trace-Sample Miss Ratios. In *ACM SIGMETRICS*, pages 79–89, June 1991.

[34] 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.

[35] T. N. Yeh and Y. Patt. A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History. In *Proc. ISCA-20*, May 1993.

[36] M. Zagha, B. Larson, et al. Performance analysis using the MIPS R10000 performance counters. In *Proc. Supercomputing*, 1996.