Let Caches Decay: Reducing Leakage Energy via Exploitation of Cache Generational Behavior

ZHIGANG HU
Princeton University
STEFANOS KAXIRAS
Agere Systems
and
MARGARET MARTONOSI
Princeton University

Power dissipation is increasingly important in CPUs ranging from those intended for mobile use, all the way up to high-performance processors for highend servers. Although 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 article 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 having an impact on performance. Because our decay-based techniques have notions of competitive online 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.

Categories and Subject Descriptors: B.3.2 [**Bold**]: Memory Structures Design Styles—*Cache memories*

General Terms: Algorithms, Design, Measurement

Additional Key Words and Phrases: Cache memories, cache decay, generational behavior, leakage power

---

Authors' addresses: Z. Hu, M. Martonosi, Department of Electrical Engineering, Princeton University, Princeton, NJ 08544; email: {hzg,mrm}@ee.princeton.edu; S. Kaxiras, Circuits and Systems Research Lab, Agere Systems, 600 Mountain Ave., Murray Hill, NJ 07974; email: kaxiras@agere.com.

1. INTRODUCTION

Power dissipation is an increasingly pressing problem in high-performance CPUs. Although power used to be mainly 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 article 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 [Wood et al. 1991]. 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.

We propose several policies for determining when to turn off a cache line. We begin with a time-based strategy, which we call *cache decay*, that turns a cache line off if a preset 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 [Karlin et al. 1991; Romer et al. 1995]. 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 approaches use an adaptation policy that approximates chip energy trade-offs; as such, they automatically approach the best-case operating points in terms of leakage energy. The article also explores cache decay techniques for multilevel or multiprogrammed hierarchies, and discusses the interactions of these techniques with other aspects of hierarchy design such as cache consistency.

Overall this work 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 onchip caches, we feel that these techniques will grow in significance over the next decade.

The structure of the article 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 explores sensitivity of cache decay to cache geometries and looks into multilevel cache hierarchies and multiprogramming issues that can alter the basic generational characteristics of the programs. Finally, Section 6 touches on a number of issues that arise when considering implementing cache decay, and Section 7 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. Although 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 to 5% of power dissipation (or even higher [Butts and Sohi 2000]), but it is expected to grow exponentially in upcoming generations [Borkar 1999; IBM Corp. 2000; Semiconductor Industry Association 1999].

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 [Powell et al. 2000; Yang et al. 2001].

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 Wood et al. [1991], the ith generation begins immediately after the ith 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

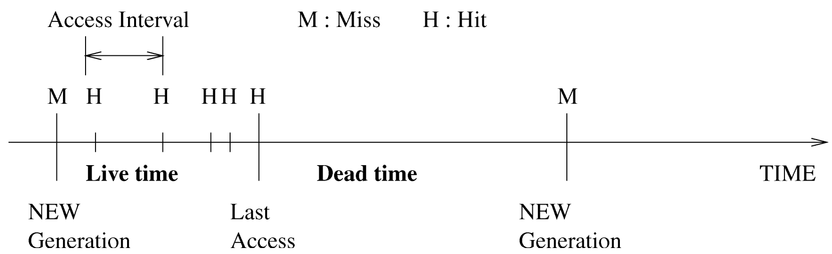

Fig. 1. Cache generations in a reference stream.

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 et al. [1991] showed that for their benchmark suite dead time was typically at least 30% on average. Similarly, Burger et al. [1995] showed that most of the data in a cache will not be used in the future. 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.

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 lines 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 32 KB 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 that 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.

[1]We sum the dead and live periods of all the generations we encounter and 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.

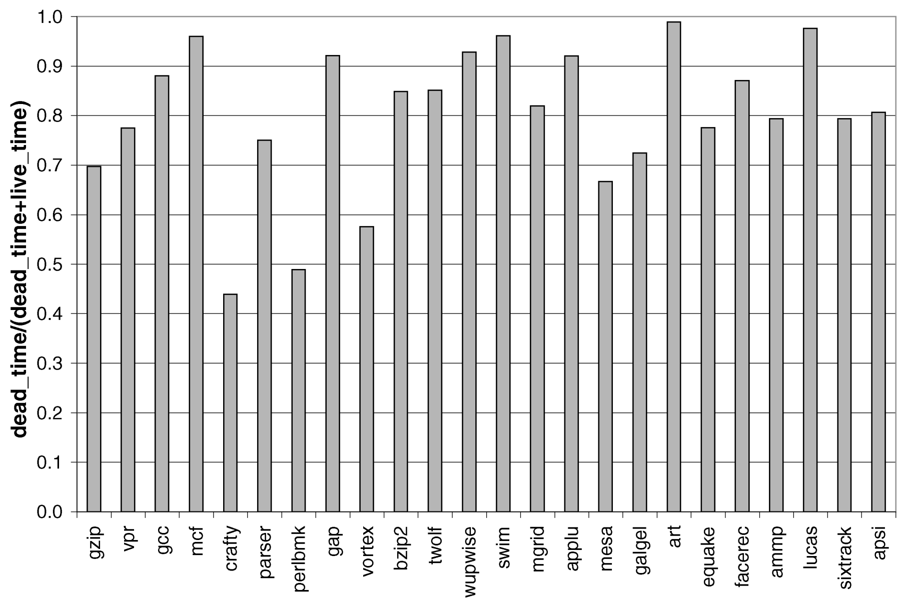

Fig. 2. Fraction of time cached data are "dead."

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 offer more realistic policies for managing these trade-offs. On the other hand, real-world attempts to put cache lines to sleep will also 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 article are based on the SimpleScalar framework [Burger et al. 1996]. Our model processor has sizing parameters that closely resemble Alpha 21264 [Gwennap 1996], but without a clustered organization. The main processor and memory hierarchy parameters are shown in Table I.

3.2 Benchmarks

We evaluate our results using benchmarks from the SPEC CPU2000 [The Standard Performance Evaluation Corporation 2000] and MediaBench suites [Lee et al. 1997]. The MediaBench applications help us demonstrate the utility of cache decay for applications with significant streaming data. The benchmarks

Table I. Configuration of Simulated Processor

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	32 KB, 1-way, 32 b blocks
L1 Icache Size	32 KB, 1-way, 32 b blocks
L2	Unified, 1 MB, 8-way LRU,
	64 B blocks, 6-cycle latency, WB
Memory	100 cycles
TLB Size	128-entry, 30-cycle miss penalty

are compiled for the Alpha instruction set using the Compaq Alpha compiler with SPEC peak settings. For each program, we follow the recommendation in Sair and Charney [2000], but skip a minimum of 1 billion instructions. We then simulate 500 M 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: 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; and 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 article, 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, we focus here on ratios of values. In this section, we describe our rationale for the range of ratio values on which we focus. 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: improved leakage energy resulting from our policies; energy from counter maintenance or other overhead hardware for cache decay policies; and 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 powered on cache bits, tag, or data. *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 decay 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, an 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 [Lee et al. 2000].

To model the ratio of dynamic L2 access energy compared to static L1 leakage per cycle, we first refer to recent work that estimates dynamic energy per L2 cache access in the range of 3 to 5 nJ per access for L2 caches of the size we consider (1 MB) [Kamble and Ghose 1997]. We then compare these data to industry data by back-calculating energy per cache access for Alpha 21164's 96 KB S-cache; it is roughly 10 nJ per access for a 300 MHz fabricated in a 0.5 μ process [Bowhill et al. 1995]. Although the S-cache is about one-tenth the capacity of the L2 caches we consider, our back-calculation leads 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 0.5 um feature size used in this older design would lead to larger capacitance and higher energy per access. Our main validation goal is 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 collect these data from several methods and compare. From the low-V_t data given in Table 2 of Yang et al. [2001], one can calculate that the leakage energy per cycle for a 32 KB cache will be roughly 0.45 nJ. A simple aggregate calculation from industry data helps us validate this. Namely, using leakage power of roughly 2 to 5% of current CPU power dissipation, the L1 cache is roughly 10 to 20% of that leakage [Borkar 1999], and CPU power dissipations are around 75 W. This places L1 leakage energy at roughly 0.3 nJ per cycle. Again, both methods of calculating these data give results within the same order of magnitude.

Dividing the 4 nJ dynamic energy per access estimate by the .45 nJ 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. In the future, leakage energy is expected to increase dramatically, which will also have an impact on 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 article.

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 turning the line off therefore. 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 [Kimbrel and Karlin 2000]. 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. Previously a body of computer systems work has successfully applied such strategies to problems including DRAM management, page allocation, superpage promotion for TLB performance, cache partitioning, prefetching, and multiprocessor synchronization [Karlin et al. 1991; Romer et al. 1995; Albonesi 1999; Delaluz et al. 2001; Lebeck et al. 2000].

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]

For example, in the case of our cache decay policy we are trying to determine when to turn off a cache line. 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 that 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 induced 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 nine times as large as the static leakage energy dissipated by the 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 1 K to 512 K 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

[2]Romer et al. [1995] include 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 she will enjoy skiing) 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.

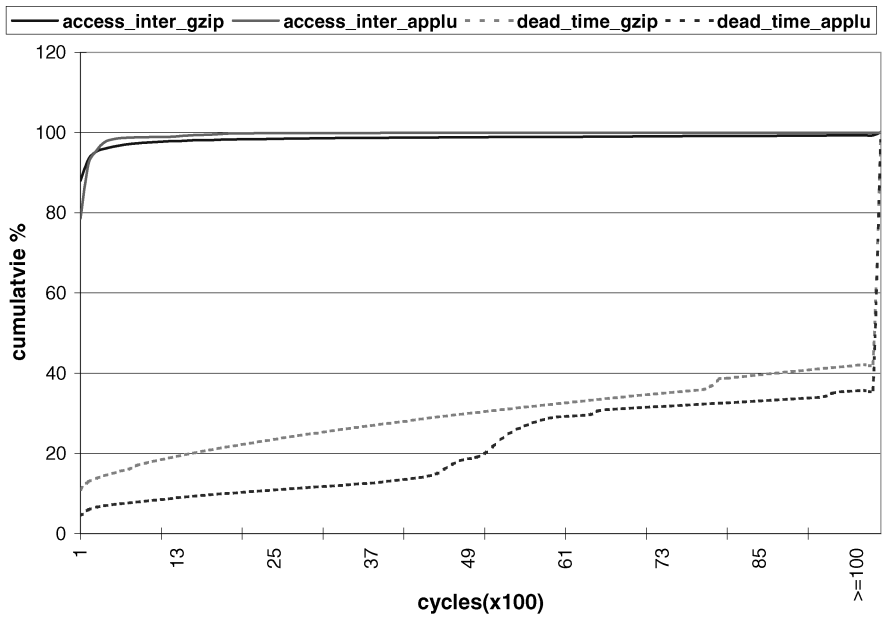

Fig. 3. Cumulative distribution of access interval and dead time for gzip and applu.

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 is 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 whereas 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 1 and 512 K 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 Powell et al. [2000]. 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

[Chen et al. 1998] 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 Powell et al. [2000] 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 article.

One way to represent currency 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 cacheline 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 two-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 [Dean et al. 1997; Intel Corp. 1997; Zagha et al. 1996]. 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 two-bit cacheline counters and thus minimize dynamic power consumption we use Gray coding so only one bit changes state at any time. Furthermore, to simplify the counters and minimize transistor count we choose 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 cacheline access signal WRD, are well behaved so there are no metastability 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 writebacks 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 writebacks 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

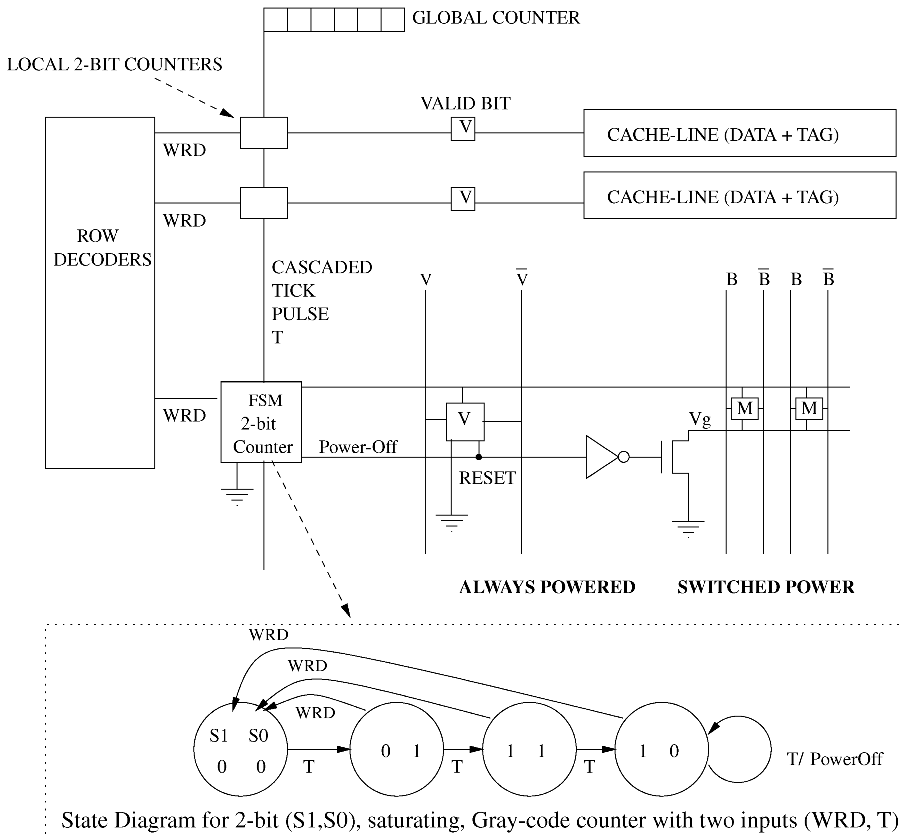

Fig. 4. Hierarchical counters.

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 two-bit counters used in our scheme, there are 0.2 bit transitions per cycle on average. Modeling each counter as a two-bit register in Wattch [Brooks et al. 2000], we estimate roughly .1 pJ per access. Therefore, at an average of 0.02 pJ per cycle, the power expended by all 1024 of these infrequently ticking counters is roughly four orders of magnitude lower than the cache leakage energy which we estimate at 0.45 nJ per cycle. For this reason, our power analysis considers 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: (1) result in a miss (since data and tag might be corrupted without power), (2) reset the counter and restore power to the cache line, and (3) delay an appropriate amount of time until the cacheline circuits stabilize after power is restored. To satisfy these requirements we use the valid bit of the cache line

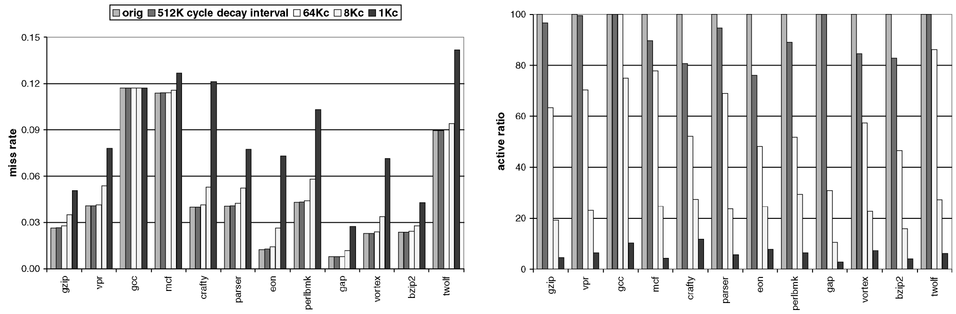

Fig. 5. Miss rate and active ratio of a 32 KB decay cache for SPECint 2000.

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 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.

4.1.1 *Analog Implementation.* Another way to represent the currency 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 metastability. Since an analog implementation can be fabricated to mimic the digital implementation, the rest of this article 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. Figures 5 and 6 plot the active ratio and miss rate as a function of the 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 32 KB 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 512 K cycles down to 1 K cycles.

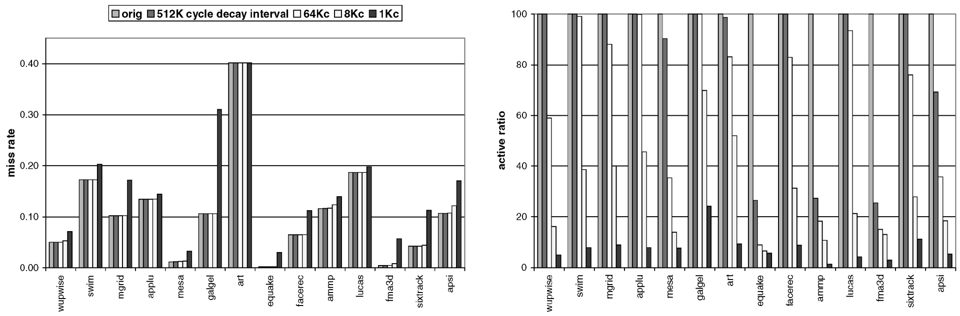

Fig. 6. Miss rate and active ratio of a 32 KB decay cache for SPECfp 2000.

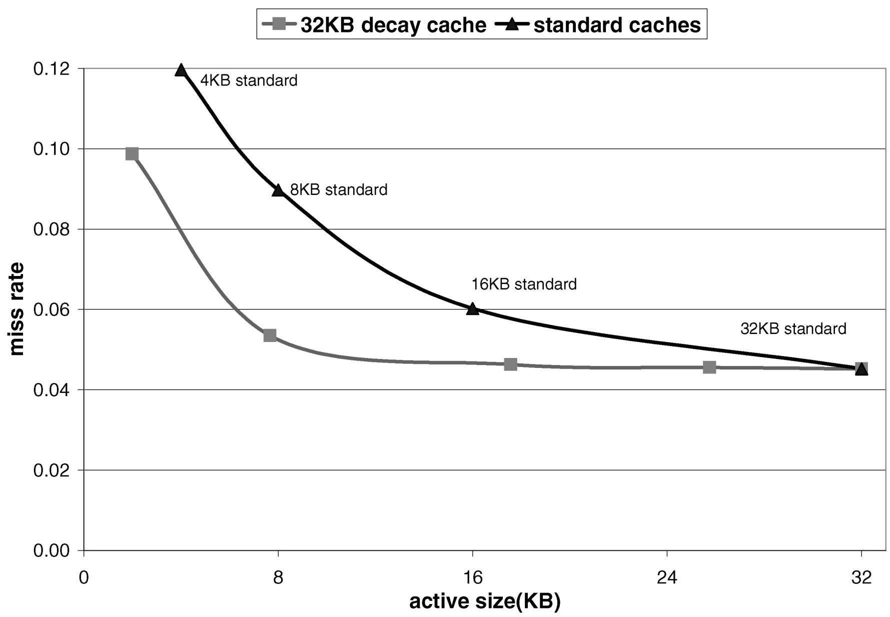

Fig. 7. Comparison of a fixed-size 32 KB decay cache with varying decay intervals to standard 4, 8, 16, and 32 KB caches. For the decay cache, the different points in the curve represent different decay intervals. From left to right, they are: 1, 8, 64, and 512 K cycles. Active size and miss rate are geometric means over all SPEC 2000 benchmarks.

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 plots similar data averaged over all the benchmarks. The upper curve corresponds to traditional nondecaying caches in which the miss rates change with the cache sizes. In a traditional cache without decay, active size is just the full cache size. The lower curve in this graph corresponds to a decay cache whose full size is fixed at 32 KB. For this cache, we can vary the decay interval and see how this influences the active size (i.e, the the number

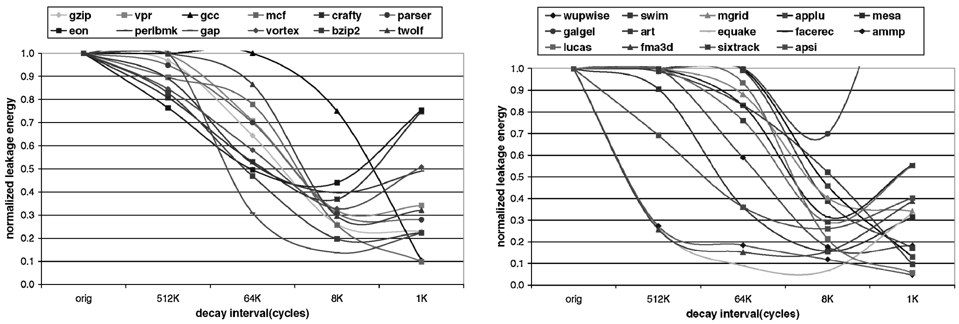

Fig. 8. Normalized cache leakage energy for an *L2Access:leak* ratio of 10. This metric takes into account both static energy savings and dynamic energy overhead. Left graph shows SPECint2000; right graph shows SPECfp2000.

of cache lines turned on). Starting at the 16 Kb traditional cache and dropping downwards, one sees that the decay cache has much better miss rate characteristics than standard caches with the same active size.

Figure 8 shows the normalized cache leakage energy metric for the integer and floating point benchmarks. In this graph, we assume that the *L2Access:leak* ratio is equal to 10 as discussed in Section 3. We normalize to the leakage energy dissipated by the original 32 KB 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 8 K cycles shows the best energy improvements. This is quite close to the roughly 10 K cycle interval suggested for worst-case bounding by the theoretical analysis. For the integer benchmarks, all of the decay intervals—including even 1 K cycle for some—result in net improvements. For the floating point benchmarks, the 8 K cycle 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.

We also wanted to explore the sensitivity of our results to different ratios of dynamic L2 energy versus static L1 leakage. Figure 9 plots four 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 8 K 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 because of the increased miss rate then its power advantage diminishes. For the decay scenarios we consider, not only do we not observe any slowdown but in fact we observe a very slight speedup in some cases, which we attribute to eager writebacks [Lee et al. 2000]. Beyond this point, however, cache decay is bound to slow down execution. For our simulated configuration, performance impact is negligible except for very

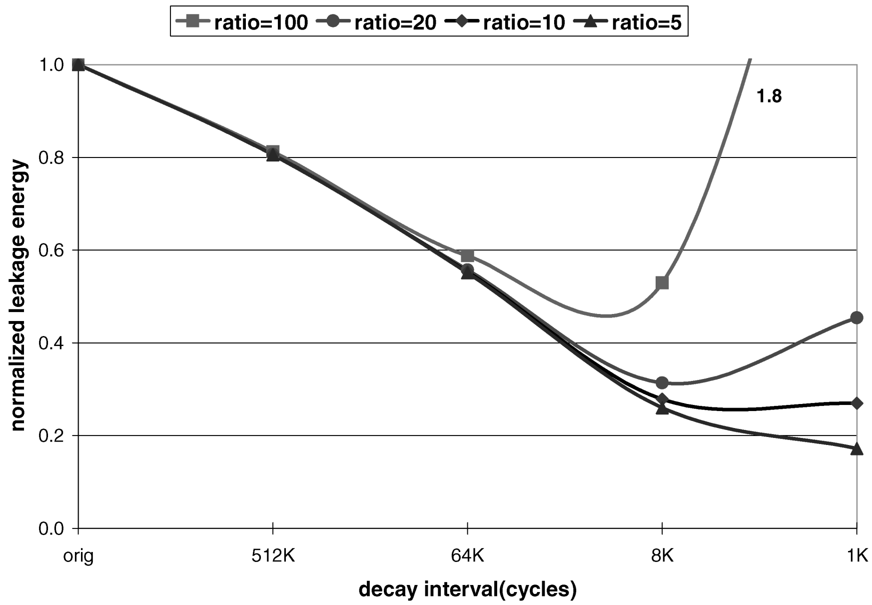

Fig. 9. Normalized L1 data cache leakage energy averaged across SPEC suite for various *L2Access:leak* ratios.

small decay intervals: the 8 K cycle interval, which yields very low normalized leakage energy (Figures 8 and 9), decreases IPC by 0.1% whereas the 1 K cycle 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 [Lee et al. 1997]. 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.

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, whereas 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.

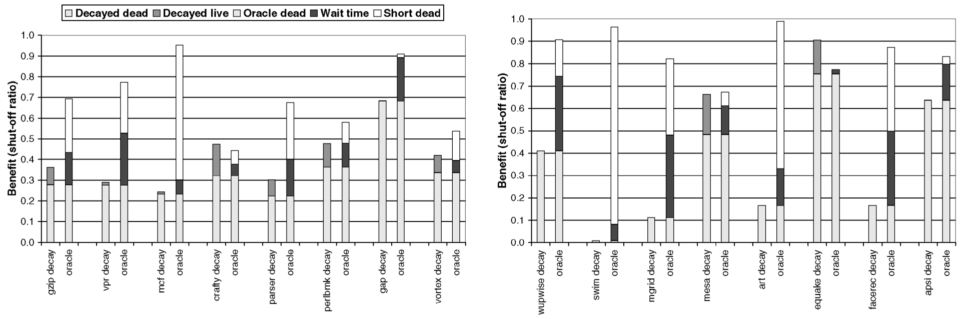

Fig. 10. Lost opportunities for time-based decay (64 K cycle decay interval). SPECint2000 and SPECfp2000.

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

4.3.2 *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 choosing decay intervals per cache line.

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 (up to 10% of the cache's leakage if we leave all the tag bits on). Instead we opt 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 two-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

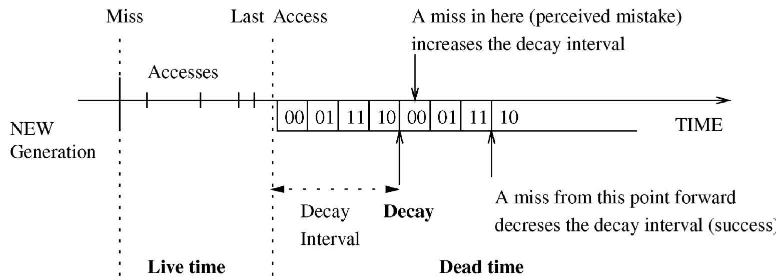

Fig. 11. Adaptive decay.

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 a longer decay interval.

4.3.3 *Results.* Figure 12 presents results of an adaptive scheme with 10 decay intervals (four-bit decay-speed field). The decay intervals range from 1 to 512 K 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 *isopower* 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 isopower 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

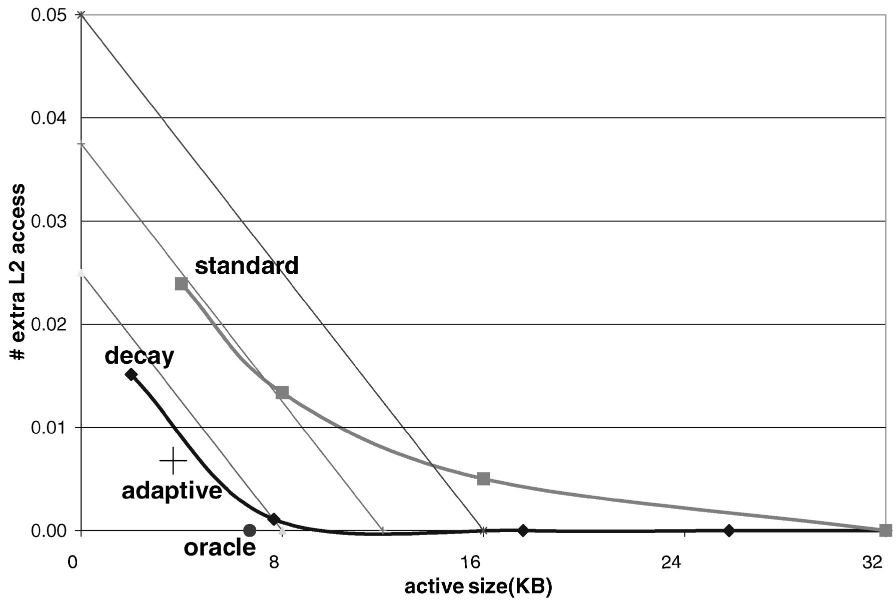

Fig. 12. Effect of adaptive decay. Isopower lines show constant power ($L2Access{:}leak = 10$). Results averaged over all SPEC2000.

tends to converge to points that are close to or lower than the highest isopower line tangent to the single-decay curve.

4.3.4 *Adaptive Decay for Set-Associative Caches.* Figure 13 demonstrates the effect of adaptive decay for a four-way 32 KB data cache. As in the previous section, the decay interval starts with an aggressively small value but increases in the event of a decay-caused miss. To identify a decay-caused miss, we keep part of the cache tag always powered on. If during a cache access, the lookup tag matches the partial powered-on tag but the data are decayed, then we declare it a decay-caused miss and modify the decay interval to a more conservative (larger) value. Since the powered-on partial tag leads to additional leakage power, we should choose it to be as small as possible. On the other hand, too few powered-on bits will lead to aliasing effects where the lookup tag matches the partial tag but does not match the whole tag. In our experiments, we found that a five-bit partial tag effectively removes aliasing effects with minimal additional leakage power.

5. CHANGES IN THE GENERATIONAL BEHAVIOR AND DECAY

In this section we first examine sensitivity of cache decay to cache size, associativity, and block size. Next we show the effectiveness of cache decay for the instruction cache. We then discuss how cache decay can be applied when multilevel cache hierarchies or multiprogramming change the apparent generational behavior of cache lines.

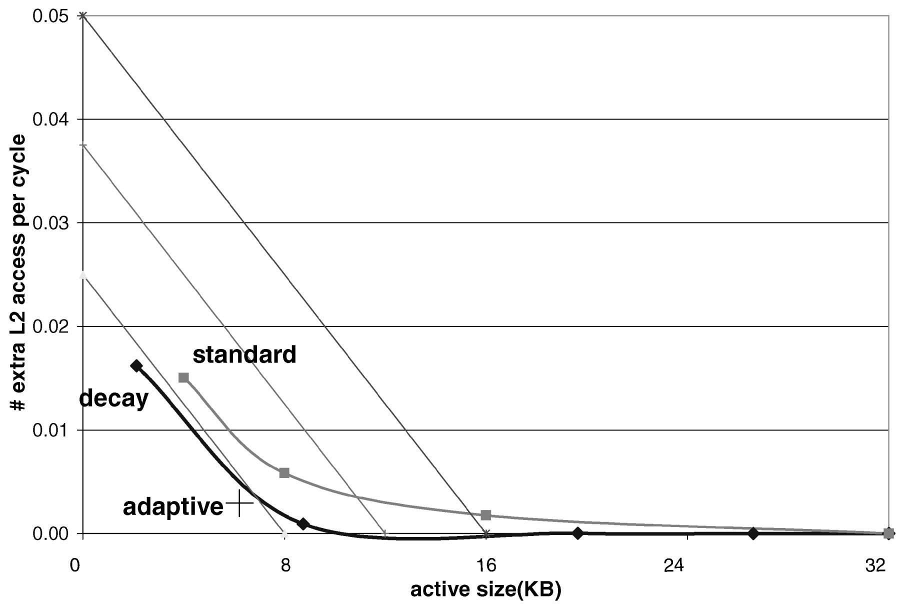

Fig. 13. Effect of adaptive decay for a four-way 32 KB cache. Isopower lines show constant power (*L2Access:leak* = 10). Results averaged over all SPEC2000.

5.1 Sensitivity to Cache Size, Associativity, and Block Size

Cache characteristics usually vary with different cache geometries, namely, cache size, associativity, and block size. In this section, we explore the effect of changing these parameters on cache decay behavior. Figures 14 and 15 plot ActiveRatio–MissRate curves of different cache size, associativity, and block size for the SPEC2000 benchmark suite. Across the configurations, we observe trends consistent with the 32 KB direct-mapped cache shown in the previous section. Cache decay constantly shows a benefit even for very small caches.

5.2 Instruction Cache

Cache decay can also be applied to instruction caches since they typically exhibit even more locality than data caches. In fact, compared to a data cache, instruction caches have the additional benefit that they do not have any write-back traffic. Figure 16 shows the normalized leakage energy for a 32 KB L1 instruction cache. Notice that even without decay, the instruction cache is not fully touched during our simulation period. The figure shows that decay works very well except for very small decay intervals.

5.3 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 the active ratio is low enough to warrant line shut-offs. For several reasons, the payoff is likely to increase as one moves outwards 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

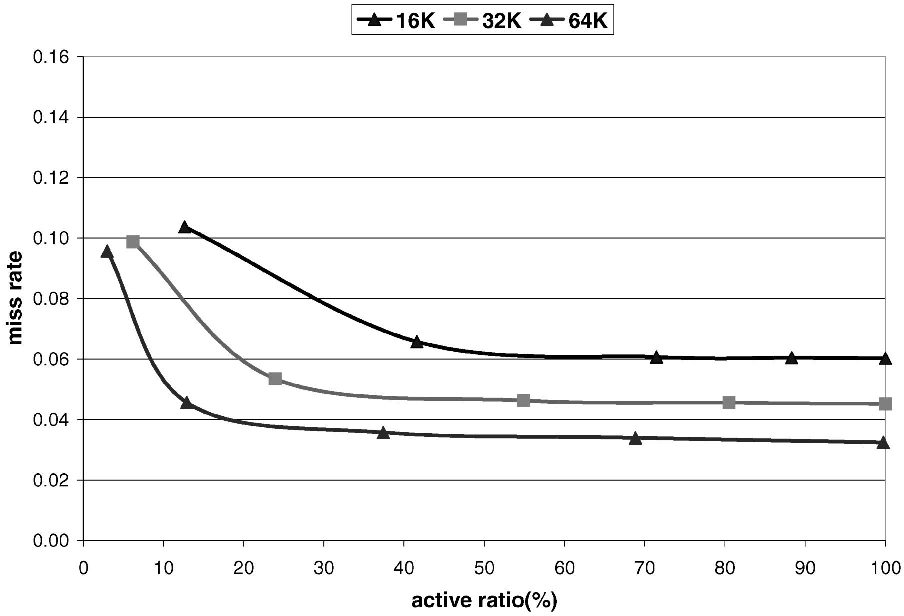

Fig. 14. ActiveRatio–MissRate curve for mcf for different sizes of L1 data cache.

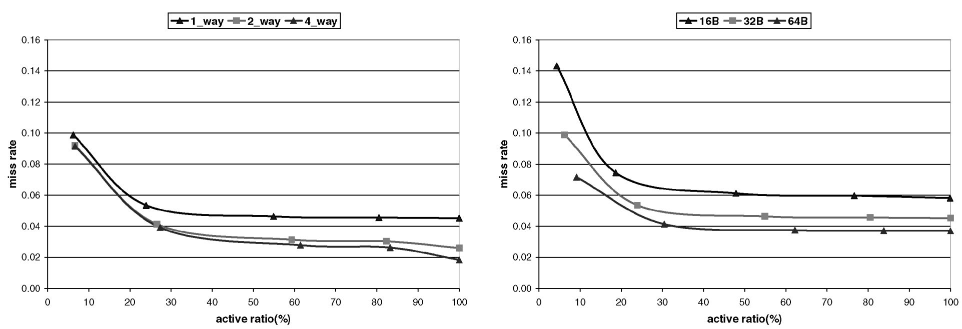

Fig. 15. ActiveRatio–MissRate curve for mcf for different associativity (left) and block size (right) of a 32 KB L1 data cache.

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 show that the average access interval and dead time for the L2 cache are 79,490 and 2,714,980 cycles, respectively. Although access intervals and dead periods become significantly larger, their relative difference remains large and this allows decay to work.

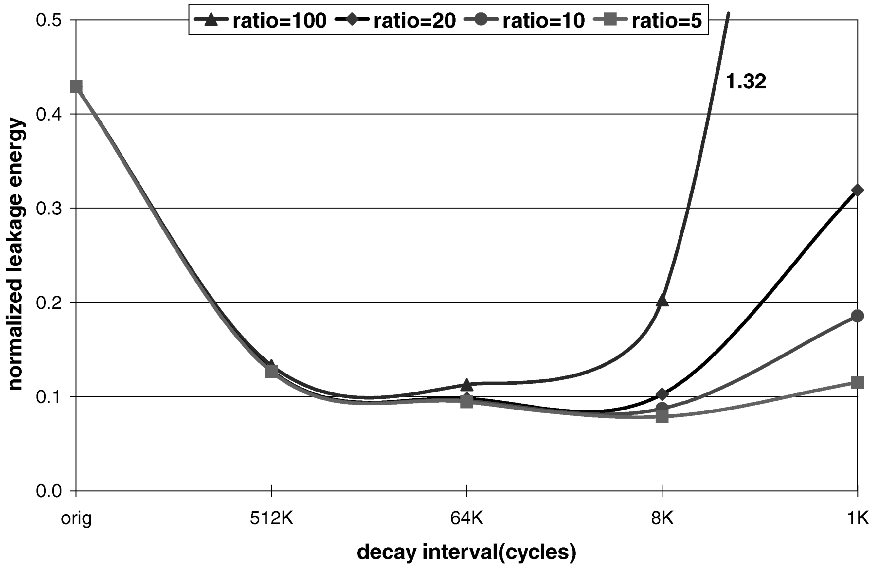

Fig. 16. Normalized L1 instruction cache leakage energy averaged across SPEC suite for various *L2Access:leak* ratios.

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 that 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 64 K cycle decay interval in L1 translates to decay intervals on the order of 6400 K cycles in the L2.

Here, we assume that multilevel inclusion is preserved in the cache hierarchy [Baer and Wang 1988]. 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. Figure 17 shows miss rate and active ratio results for the 1 MB 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, a 128 K cycle 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 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 1 M cycle decay interval range from 71 to 155,773 with an average of 2400. For a 128 K cycle

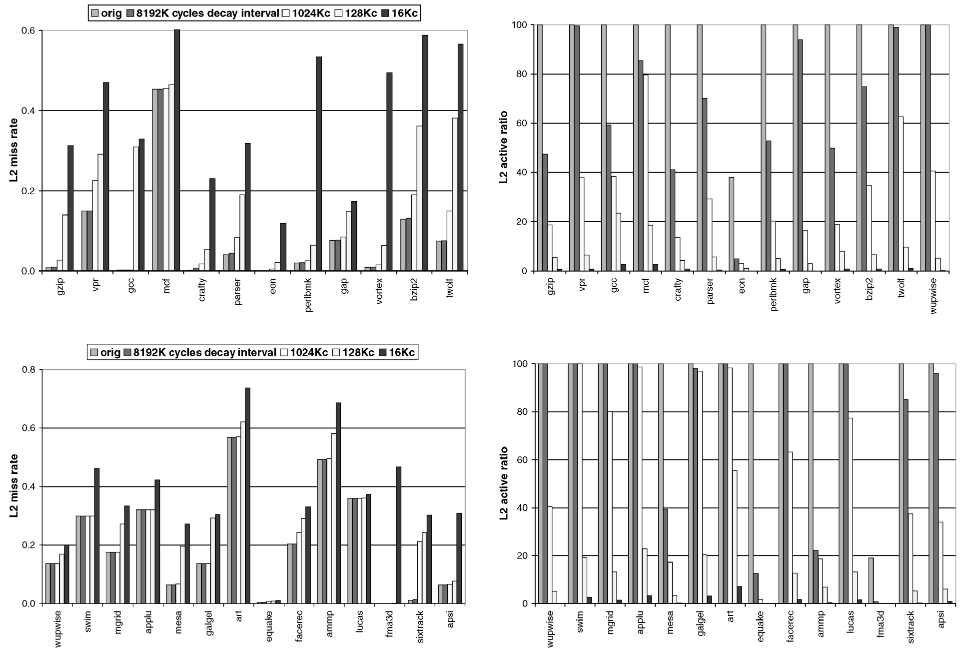

Fig. 17. Miss rate and active ratio of a 1 MB L2 decay cache for SPECint 2000 (upper) and SPECfp 2000 (lower).eon and fma3d do not fully utilize the L2 cache.

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.

5.4 Multiprogramming

Our prior results all focus on a single application process using all of the cache. In many situations, however, the CPU will be timeshared 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 "sewed" together pieces of the traces, with each benchmark getting 40 ms 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

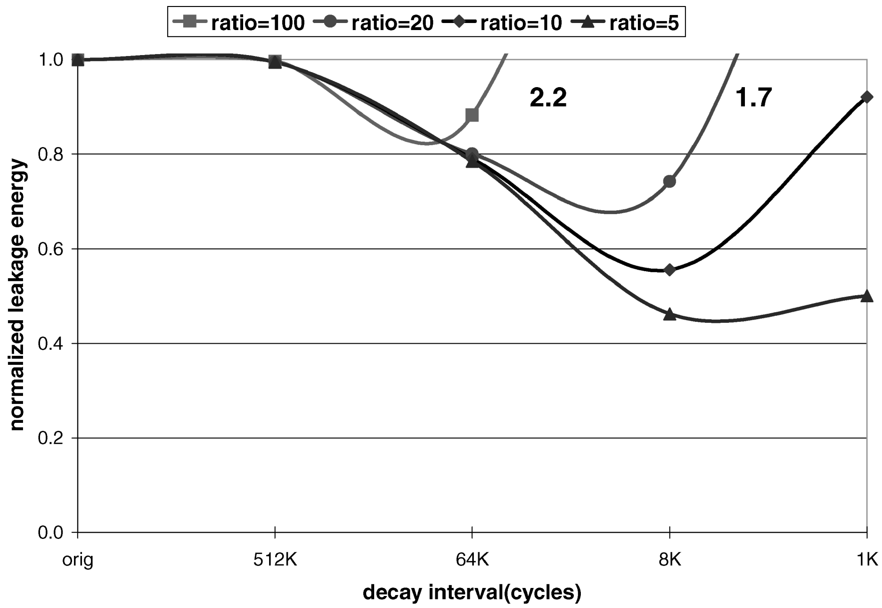

Fig. 18. Normalized L1 data cache leakage energy for cache decay methods on a multiprogrammed workload.

sufficiently coarse-grained that it does not have an impact on 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 aggressive decay policies such as decaying items at the end of a process time quantum. See Figure 18.

Although multiprogramming slightly increases the overall active ratio in the cache, it also gives opportunities to be more aggressive in turning off cache lines. In particular, one could turn off many lightly used lines at each context switch, since they are unlikely to survive in the cache until the next time this process gets to run. One could also consider per-process counter schemes. These schemes would allow processes that have recently awakened to ensure that they do not turn off any of their own cache lines; since these lines have been dormant so long, they would otherwise be key candidates for shut-off even now as they are likely to be used again. A solution to this situation would be to have global counters considered part of the process state, swapped in and out at context switches, and cacheline counters tagged with the process ID. It is unlikely, however, that this complexity would be warranted based on current or near-future values for dead periods, context-switch interval, and leakage.

6. 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.

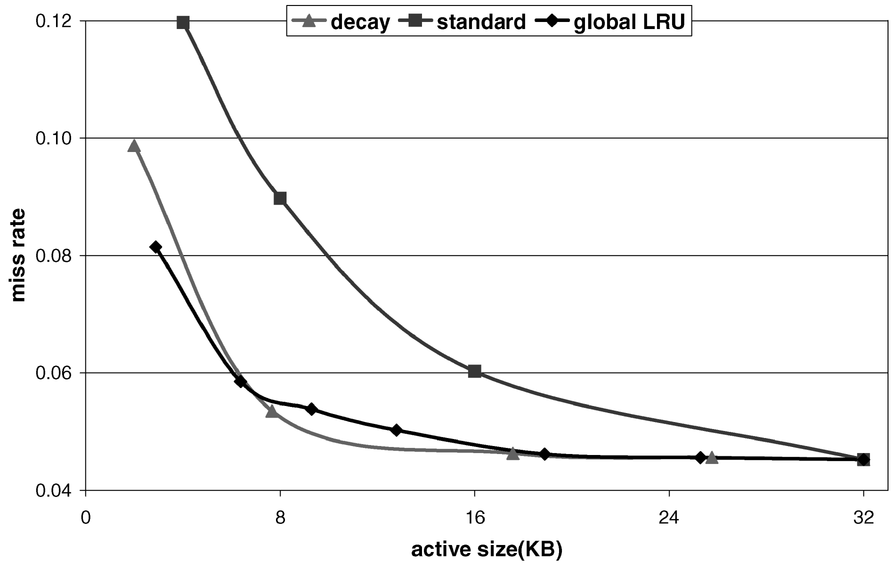

Fig. 19. Global LRU decay versus Working Set decay. Miss rates and active sizes as geometric means over all SPEC2000 programs.

6.1 LRU Decay

Time-based decay is in essence a Working Set algorithm. Working Set and global LRU perform comparably in virtual memory paging [Stallings 2001] and this 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 [Peir et al. 1998] and Indirect-Index Cache [Hallnor and Reinhardt 2000]). 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 Peir et al. [1998] and Hallnor and Reinhardt [2000], 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; that is, we can request half of the lines—the least recently used—to be turned off. In Figure 19 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 whereas LRU decays 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.

6.2 Multiprocessor Cache Coherence

Another key issue in the realization of our cache decay mechanism is that it also be usable in cache-coherent multiprocessor systems. Although we do not have quantitative results here, we feel that cache decay and multiprocessor cache coherence 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 coherence protocol performance by purging stale information from the 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 [Lebeck and Wood 1995; Lai and Falsafi 2000]. 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.

6.3 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.

6.4 DRAM Caches

Some recent work has discussed the possibility of DRAM-based caches [Wilson and Olukotun 1997]. 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 writeback 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.

6.5 Cache Decay with Four-Transistor-DRAM Caches

Cache decay is also a natural application in quasistatic memory technologies that share characteristics of both six-transistor static RAM (SRAM) and one-transistor dynamic RAM (DRAM) technologies. Here, we discuss a decay cache that uses four-transistor DRAM memory cells with no path to ground. Such cells

Table II. Hold Times (Nanoseconds) for 1.5 V and 3.3 V Versions of 4T Cells at Different Operating Temperatures

	1.5 V			3.3 V		
	25°C	85°C	125°C	25°C	85°C	125°C
NOM	18,000	1700	560	1,040,000	57,200	9400

readily replace ordinary six-transistor static cells in CMOS processes. Such 4T DRAM cells possess two characteristics fitting for decay: they are refreshed upon access and they decay over time if not accessed.

The 4T cells are similar to the ordinary 6T cells but lack two transistors connected to V_{dd} that replenish the charge that is lost via leakage. Their size is smaller than a SRAM cell because they only require four transistors and have no V_{dd} lines. 4T cells naturally decay over time without the need to switch them off as with SRAM memory cells. Once they lose their charge they stop leaking since there is no connection to V_{dd}.

In addition, 4T cells are automatically refreshed from the precharged bit lines whenever they are accessed. When a 4T cell is accessed, its internal nodes go to a full voltage swing refreshing the logical value stored in it; there is no need for a read-write cycle as in 1T DRAM. As the cell decays and leaks charge, the voltage difference of its internal nodes drops to the point where the sense amplifiers cannot distinguish its logical value. Below this threshold we have a decayed state, where reading a 4T DRAM cell may produce a random value, not necessarily a zero.

The decay interval in a 4T decay cache depends on the time it takes the voltage differential in the cells to drop below a safe detection threshold. This in turn depends on the leakage currents present in the 4T cell that are affected by variations in the process technology and by temperature. To study decay intervals for the 4T decay cache we used Agere's COM2 0.16 μ 1.5 V CMOS process and we simulated the appropriate circuits using Celerity tools. We studied hold times (time from last access to 100 mV differential voltage in the cell) for three temperatures, room temperature (25°C), operating temperature (85°C), and high temperature (125°C), as shown in Table II. Although there can be significant variations in the manufactured transistors we note that for most of production lots transistor behavior is close to nominal. Selecting I/O transistors (readily available in COM2 technology) for the 4T cells significantly extends hold times at the expense of increased cell area. Even in this case the 4T cell area is still less than that of the 6T cell. Table II lists the hold times in nanoseconds for the temperatures we studied. In many cases these hold times are appropriate for cache decay by not being exceedingly small.

Given that the hold time is large enough to be comparable to the suggested decay intervals in this article the only issue that remains is to distinguish among decayed and live 4T cells. For this we use decay counters and 6T SRAM valid bits. In this case, the decay counters are used not to switch off cache lines (since this is unnecessary in this design) but rather to indicate via the stable valid bits when the values of the 4T cells become unreliable because of their natural decay.

Since 4T cells discharge at a specific rate (as do the DRAM memory cells) there is no benefit in decaying and invalidating a cache line any sooner than its hold time. When the hold time is large (on the order of tens of thousands of machine cycles for GHz clocks) the local cacheline decay counters can be very coarse-grained with very low resolution. In this case, single-bit local cacheline decay counters can be used. The global counter ticks at a period half the hold time. Since the last access to a cache line in relation to the next global tick pulse is unknown, decay intervals range from one-half hold time to a full hold time (with an average of 3/4 of the hold time).

6.6 Branch Predictors

Cache decay can also be applied to other memory structures in a processor such as large branch prediction structures [Yeh and Patt 1993]. Decay in branch predictors has interesting implications in terms of cost. Although 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.

7. CONCLUSIONS

This article has described methods to reduce cache leakage power by exploiting generational characteristics of cacheline 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 that 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 reaccessed within a time window called a "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-cacheline counters. Cache lines are "decayed" when a cacheline 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 online algorithm theory allows one to reason about appropriate decay intervals given a dynamic to static energy ratio. Specifically, competitive online 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.

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

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

ALBONESI, D. H. 1999. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 32nd International Symposium on Microarchitecture*.

BAER, J. AND WANG, W. 1988. On the inclusion property in multi-level cache hierarchies. In *Proceedings of the Fifteenth International Symposium on Computer Architecture*.

BORKAR, S. 1999. Design challenges of technology scaling. *IEEE Micro 19*, 4.

BOWHILL, W. J., BELL, S. L., BENSCHNEIDER, B. J., BLACK, A. J., BRITTON, S. M., CASTELINO, R. W., DONCHIN, D. R., EDMONDSON, J. H., FAIR, III, H. R., GRONOWSKI, P. E., JAIN, A. K., KROESEN, P. L., LAMERE, M. E., LOUGHLIN, B. J., MEHTA, S., MUELLER, R. O., PRESTON, R. P., SANTHANAM, S., SHEDD, T. A., SMITH, M. J., AND THIERAUF, S. C. 1995. Circuit implementation of a 300-MHz 64-bit second-generation CMOS Alpha CPU. *Dig. Tech. J. 7*, 1, 100–118.

BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architecture-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture* (ISCA 2000).

BURGER, D., AUSTIN, T. M., AND BENNETT, S. 1996. Evaluating future microprocessors: The SimpleScalar tool set. Tech. Rep. TR-1308 (July), Univ. of Wisconsin—Madison Computer Sciences Dept.

BURGER, D., GOODMAN, J., AND KAGI, A. 1995. The declining effectiveness of dynamic caching for general-purpose microprocessors. Tech. Rep. TR-1216, Univ. of Wisconsin—Madison Computer Sciences Dept.

BUTTS, J. A. AND SOHI, G. 2000. A static power model for architects. In *Proceedings of the 33rd International Symposium on Microarchitecture* (Dec.).

CHEN, Z., JOHNSON, M., WEI, L., AND ROY, K. 1998. Estimation of standby leakage power in CMOS circuits considering accurate modeling of transistor stacks. In *ISLPED*.

DEAN, J., HICKS, J. E., WALDSPURGER, C. A., WEIHL, W. E., AND CHRYSOS, G. 1997. Profileme: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the Thirtieth International Symposium on Microarchitecture*.

DELALUZ, V., KANDEMIR, M., VIJAYKRISHNAN, N., SIVASUBRAMANIAM, A., AND IRWIN, M. J. 2001. Dram energy management using software and hardware directed power mode control. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture* (Feb.), 84–93.

GWENNAP, L. 1996. Digital 21264 sets new standard. *Microproc. Rep.* 11–16.

HALLNOR, E. G. AND REINHARDT, S. K. 2000. A fully associative software-managed cache design. In *Proceedings of the 27th International Symposium on Computer Architecture* (June).

IBM CORP. 2000. Personal communication. November.

INTEL CORP. 1997. Intel architecture optimization manual.

KAMBLE, M. B. AND GHOSE, K. 1997. Analytical energy dissipation models for low power caches. In *Proceedings of the International Symposium on Low Power Electronics and Design*.

KARLIN, A. R., LI, K., AND MANASSE, M. S. 1991. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*.

KIMBREL, T. AND KARLIN, A. 2000. Near-optimal parallel prefetching and caching. *SIAM J. Comput.*

LAI, A.-C. AND FALSAFI, B. 2000. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of the 27th International Symposium on Computer Architecture* (May).

LEBECK, A., FAN, X., ZENG, H., AND ELLIS, C. 2000. Power aware page allocation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, 105–116.

LEBECK, A. R. AND WOOD, D. A. 1995. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd International Symposium on Computer Architecture* (June).

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communication systems. In *Proceedings of the Thirtieth International Symposium on Microarchitecture* (Dec.).

LEE, H.-H., TYSON, G. S., AND FARRENS, M. 2000. Eager writeback—A technique for improving bandwidth utilization. In *Proceedings of the 33rd International Symposium on Microarchitecture* (Dec.).

PEIR, J., LEE, Y., AND HSU, W. 1998. Capturing dynamic memory reference behavior with adaptive cache topology. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (Nov.).

POWELL, M. D., YANG, S.-H., FALSAFI, B., ROY, K., AND T. N. VIJAYKUMAR. 2000. Gated-V_{dd}: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the International Symposium on Low Power Electronics and Design '00*.

ROMER, T., OHLRICH, W., KARLIN, A., AND BERSHAD, B. 1995. Reducing TLB and memory overhead using online superpage promotion. In *Proceedings of the 22nd International Symposium on Computer Architecture*.

SAIR, S. AND CHARNEY, M. 2000. Memory behavior of the SPEC2000 benchmark suite. Tech. Rep., IBM.

SEMICONDUCTOR INDUSTRY ASSOCIATION. 1999. The International Technology Roadmap for Semiconductors. Available at http://www.semichips.org.

STALLINGS, W. 2001. *Operating Systems*. Prentice-Hall, Englewood Cliffs, N.J.

THE STANDARD PERFORMANCE EVALUATION CORPORATION. 2000. WWW Site. http://www.spec.org.

U.S. ENVIRONMENTAL PROTECTION AGENCY. 2001. Energy Star Program Web page. http://www.epa.gov/energystar/.

WILSON, K. M. AND OLUKOTUN, K. 1997. Designing high bandwidth on-chip caches. In *Proceedings of the 24th International Symposium on Computer Architecture* (June), 121–132.

WOOD, D. A., HILL, M. D., AND KESSLER, R. E. 1991. A model for estimating trace-sample miss ratios. In *Proceedings of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* (June), 79–89.

YANG, S.-H., POWELL, M. D., FALSAFI, B., ROY, K., AND VIJAYKUMAR, T. 2001. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance I-caches. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*.

YEH, T. N. AND PATT, Y. 1993. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the Twentieth International Symposium on Computer Architecture* (May).

ZAGHA, M., LARSON, B., TURNER, S., AND ITZKOWITZ, M. 1996. Performance analysis using the MIPS R10000 performance counters. In *Proceedings of Supercomputing.*

Received April 2001; revised January 2002; accepted February 2002