

Efficient Energy Saving Scheme for On-Chip Caches

Abstract— With the reduction in feature size the static power component, such as the leakage power, dominates the dynamic power consumption in the on-chip caches. It has been observed that [2] all cache lines need not be kept active at all times. Only a very few lines during a given window of time need to be actively powered from the footprint, i.e., they are accessed during that time. Earlier research [2] has addressed the issue of how to determine the set of active lines and how long to keep them active (powered). Circuit techniques have also been developed to keep a cache line in low leakage state i.e., Drowsy State when the line is not being accessed or used. Such a cache is called drowsy cache. These circuit techniques try to achieve maximum reduction in the leakage power without losing the information content and with minimal performance penalty associated with power transitions. These techniques when used with optimal switching scheme, which decides when and what lines to drowse, results in maximum reduction in energy consumed. In this paper, we study the cache access pattern to evaluate them and arrive at an optimal scheme to implement the drowsy cache. We achieve energy reduction on the average of 88% of maximum gain achievable through the underlying circuit technique. We also compare the performance of our scheme with the earlier proposed schemes and show that we can achieve up to 6% of higher saving in cache energy for the benchmarks studied (with an average on 4% for all benchmarks with equal weights) without any additional performance penalty.

I. INTRODUCTION

Reducing the power and hence energy, is one of the important goals of today's embedded processor designers. Even in the case of processors targeting desktops and servers cache power reduction assumes importance in-terms of reducing the heating problem. Dynamic power, i.e., switching the transistors, is a major contributor to the power consumption in caches till recently. As the cache size increases and feature size decreases, the static power i.e., leakage component starts dominating the energy equation. Hence by reducing the static power, considerable improvement in energy usage can be achieved. Various circuit techniques have been researched to achieve static power reduction. Gated Vdd [3], ABB-MTCMOS [4], and dynamic Vdd scaling (DVS) [5] to name a few. Among these techniques the DVS technique achieves the twin target of not losing the cache state and reducing the leakage power. Moreover the penalty to transit from low power (drowsy) to normal state is minimal, both in-terms of time and energy. Hence the DVS scheme is claimed to achieve the maximum static power reduction of these techniques. Since cache memory occupies a significant portion of a processor chip and consumes almost 20% or more power, several suggestions have been made to save power in cache memory operation. A *cache line* or *block* is the smallest replaceable unit in the cache. In a

scheme suggested in [2], it had been proposed that power to individual cache lines be controlled. The power to a cache line is turned on when the cache line is accessed. A cache line can thus be in two states: *active* - the power to the cache is already on and no performance penalty is incurred when such a cache line is accessed; *drowsy* - power to the cache line is off and a performance penalty, whose value is determined by the underlying circuit technique, is incurred when a line in such a state is accessed. The instant at which a line must be transited from drowsy to active state is clear, i.e., when an access to such a line is made. However, it is not clear when the reverse operation, i.e., how long the power must be kept on must be carried out. In the scheme proposed in [2], power to all cache lines is turned off periodically. An optimal interval of 2000 cycle was shown to be effective if such a scheme were to be used. In this paper, we study the cache access pattern to evaluate them and arrive at an optimal scheme to implement the drowsy cache. We compare the performance of our scheme with the earlier proposed schemes and show that we can achieve up to (6%) higher saving in cache energy without any additional performance penalty.

A. Energy Vs Time tradeoff model

Any scheme that uses one of these techniques trades off performance for power reduction. Reduction in performance increases the execution time of an application and thus the whole system runs for longer duration. This performance reduction may lead to increased energy consumption if the tradeoff is not studied carefully. Let us consider a hypothetical system consisting of two components X and Y to understand the tradeoff and the *real gain* achieved by any energy reduction scheme. Suppose performance is traded off to reduce the power consumption of component Y . The power consumption of component X is not changed. In other word, component X represents the part of the chip whose power consumption remains unchanged when a particular scheme is deployed and component Y is the part of the chip whose power consumption is reduced. We use the following notation to derive the tradeoff relationship.

$\xi \Rightarrow$ Total energy of the system

$P \Rightarrow$ Total power consumed ($= P_x + P_y$)

$P_y \Rightarrow$ Power consumption of component $Y (= y * P)$

$P_x \Rightarrow$ Power consumption of component $X (= (1-y)*P)$

$\Gamma \Rightarrow$ Time of execution without performance reduction

$q \Rightarrow$ Additive Power reduction factor

$\rho \Rightarrow$ Additive Performance reduction factor

Then,

$$\xi = P * \Gamma = (P_x + P_y) * \Gamma$$

and the reduced power of component Y is given by

$$P_{yr} = (1 - q) * P_y = (1 - q) * y * P,$$

hence the energy consumption of the system after power reduction is

$$\xi_r = (1 - q) * P_y * \Gamma * (1 + \rho) + P_x * \Gamma * (1 + \rho) + K \quad (1)$$

where K is the energy consumed by the implementation of the scheme itself to achieve energy reduction. If we assume that K is negligible then *Gain* is given by

$$G = (\xi - \xi_r) / \xi = (1 + \rho) * q * y - \rho \quad (2)$$

The gain factor G is positive if condition

$$\rho / (1 + \rho) < q * y \quad (3)$$

is satisfied. The condition in Eqn. 3 provides us a mechanism to compare when to or not to use a scheme, or when the scheme is beneficial. And the gain factor G can also be used to choose between various schemes. In the case of energy reduction using *drowsy cache*, the following parameters control the tradeoff.

ρ is the penalty incurred due to accessing the cache line in drowsy state, which is characterized by the parameter *drowsy hit*.

q is the average leakage power reduction achieved through the scheme.

y is the factor of the total power consumed by caches in the system without power reduction.

B. Our Goal

Any scheme that manages the power consumption of caches should switch the cache lines from drowsy to active state and vice-versa. The power to a cache line may be switched between these two states by predicting a future access to it. A scheme may prove less efficient if the penalty for the transition from drowsy state to active state is very high and/or the power to a cache line is kept for long duration. The only reasonable point in time to make a cache line active seems to be the first access to it when it is in drowsy state, or just prior to that if it can be predicted. Thus the only real control can be exercised and more energy savings can be achieved by predicting when to switch the cache line back to drowsy mode from active mode.

In this paper we investigate relationship between cache access pattern and tradeoff parameters ρ and q , to evolve a scheme that maximizes G for a y that is fixed by the system. We again study the well-known characteristics of caches, namely, *Temporal Locality* and *Spatial Locality* for this purpose with specific interest in extracting the key parameters for the power control scheme.

Our goal is to find an optimal window size, which determines for how many cycles a cache line must be kept in active state once it is accessed. Once we know this, a timer can be associated with each cache line that enters the active state.

Upon expiry of the timer, the power to the cache line is switched back to the low power (drowsy state). The value of the timer must be chosen carefully so that the loss in performance while turning it on can be well compensated by the power gain obtained. The goal is that the most accesses to that line should happen during the time when the timer (and line) is active. In a variation to the timer scheme, the timer may be restarted every time the cache line is accessed again. In this case, the timer value is chosen so that a second access is most likely to occur during the duration of the timer and no access is likely to occur after the expiry of the timer. This is depicted in Fig. 1. We prefer the timer scheme with this variation, re-triggerable window, in this paper.

For a cache of size 32KB with 32B line size (which is typically the case), the above scheme would require 1024 timers. It is thus possible that the energy savings achieved by turning power on and off to cache lines using timers will be severely offset by the energy consumed by the timers. Hence we need to minimize the number of timers. One possibility is to only deploy a fixed number of timers, and associate these timers with active lines using a fully associative technique. Again, a complex control is required to realize this scheme and one has to remain concerned with power gain in cache lines vs power loss in timers and their control, and the overall energy saving. Another possibility is to use timers in a set associative manner, where a single timer is associated with a set of lines. The timer is active when any (one or more) of the lines in that set are active. Cache lines are put in active state individually as and when they are accessed. The same timer is activated for any line that belongs to the set. However, upon expiry of the timer, all cache lines in the set are transitioned to drowsy state. In order to determine the composition of such sets, we again study the spatial locality property with our specific interest in mind. The objective is to identify a group of cache lines that would be accessed together, and hence could be switched off at the same time.

C. Outline of the paper

With the above objectives in place, we organize the sections in the paper as follows. In Section 2 and 3 *temporal* and *spatial locality* of various benchmarks are analyzed. In Section 4, based on the analysis in Sections 2 and 3, outline of our new scheme is proposed for controlling the drowsy cache. Section 5 compares the performance of our scheme to the earlier proposed scheme. Section 6 includes some concluding remarks.

II. TEMPORAL LOCALITY

The simplest way to implement *Drowsy Cache* scheme is to make a cache line active only when it is accessed and turn it off. This would ensure that leakage power is minimum. But this would incur penalty for every cache access, and from Eqn. 1, ρ and K will have negative effect on the energy savings achieved from factor q . Hence it is necessary to understand cache access

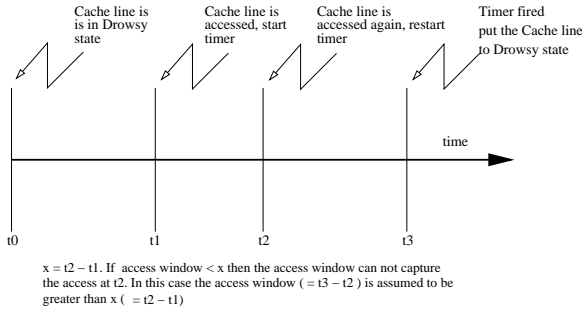


Fig. 1. Measurement of accesses in a re-triggerable window

pattern, and use this information to make an intelligent decision on how to drowse a cache line. The first property of caches that influences this decision is *Temporal Locality*. The temporal locality is a property of programs that refers to reuse of data that have used recently. This is one of the fundamental property that makes caching of data yield higher performance for most programs. Higher temporal locality implies higher likelihood of the same cache line being accessed many times within a short duration of time once it is accessed. This likelihood is a function of elapsed time since the previous access i.e., the probability of a line being accessed decreases inversely with the elapsed time since the previous access for certain duration. Hence a cache line, which is made active, can be put back to drowsy mode after the access probability has reduced considerably.

A. Measurements

The temporal locality characteristics can be measured by defining the parameter *Degree of Temporal locality*, δ_t , which is defined as

$$\delta_t = \#(temp - loc - mem - refs) / \#(windows).$$

A memory reference m to line l occurring at time t qualifies as a *temp-loc-mem-ref* if there is a memory reference m' to the same line l at time t' such that $t' - t \leq w$, where w is a small pre-set unit of time. We refer to w as the size of access window. The window that is used to measure δ_t - *access window* - should be chosen in such a way that the probability of a cache line being accessed is above certain threshold. *access window size* determines the number of cycles for which the cache line should be kept active once the line is accessed. The *access window size* is optimal if it captures most of the consecutive accesses, i.e., temporal locality is fully exploited. In the expression for δ_t , the numerator is the number of all such accesses that qualify as *temp-loc-mem-ref* during the execution of a program. The denominator is the number of windows observed during the execution of the program. Note that if several accesses occur consecutively such that all the consecutive accesses are within w , but the first to the last accesses are farther than w time apart, we still count it as one window. Thus, we measure

δ_t in the following way (refer to Figure 1). For every set of consecutive accesses that fall within the window, we count the number of all such accesses. We also count number of such window occurring (Figure 1 depicts one window and two such accesses). Then, δ_t is an average of number of accesses in a window. For example if accesses are made to line l in cycle 1, 13, 15, 526, 1101... then for access window of size 512, the timer is triggered by access at cycle 1 and re-triggered by accesses at 13,15 and 526. But before the access at 1101 the timer expires and hence δ_t for this window is 4. A new window is triggered by access at 1101 cycle. The timer is essentially a *retriggerable monostate timer*. In actual measurement, whenever a line is accessed it is made active for the next w cycles and the accesses to the line are counted as temp-loc-mem-refs. If the line is accessed within the period of w cycles, the timer is restarted and we continue to count the accesses to the line. We assume that the timers are implemented using counters that are initialized to a maximum value corresponding to the size of the access window chosen and the counter is decremented every cycle. Once the timer runs out we consider that as the end of the window. Thus δ_t for any window size indicates the savings in the drowsy penalty by keeping the line active for the access window size. The scheme based on other parameters of the system can decide on the best trade-off.

B. Observations

We use SimpleScalar 3.0 [6] in the configuration as shown in Table I to collect the statistics. All the benchmarks were run for 200 million instructions with fast-forwarding done for 1 billion instructions. For various benchmarks we measured δ_t and it is

TABLE I
SIMPLESCALAR SIMULATION PARAMETERS

Parameter	Value
Processor	Alpha out of order 4 way issue
L1 - DCache	32KB, 4 way, 32B line, LRU, 1 cycle latency
L1 - ICache	16KB, 1 way, 32B line, 1 cycle latency
L2 - Unified	256KB, 4 way, 64B line, LRU, 8 cycle latency

as shown in Figure 2. In the first set of graphs, we present the value of δ_t for different sizes of access window for various benchmark programs. From the Figure 2, we notice that for most of the benchmarks slope of δ_t is small i.e., increasing the access window size does not increase the number of accesses captured by the window significantly.

To make the above observation more clearly the number of accesses in each window is normalized to the number of accesses in the largest access window size, i.e., 2048 cycles. The results are depicted in the second set of graphs in Figure 2. These graphs illustrates the fact that for most of the benchmarks 256 cycles access window size performs almost as good as 1024 cycles. A few benchmarks also show the characteristic of linear increase with the increase in the access window size. For these benchmarks, most of the accesses recur after 1024 cycles. In such cases paying in a small penalty in performance

may be worth a while to save power as the accesses are widespread in time since the micro-architecture has more time to amortize the penalty incurred due to *Drowsy Hits*.

Another parameter to watch for here is the average number of cycles between the accesses. We define *Effectiveness* of an access window size as the average number of accesses per cycle in an access window. *Effectiveness* can be viewed as the reduction in drowsy percentage, thus energy savings, to reduce the impact of performance penalty. To compare this parameter for various benchmarks, we normalized it to *effectiveness* of the largest access window size, 2048 cycles. The normalized results are shown in the third set of graphs in Figure 2. From the graphs we can observe that the knee of the effectiveness curve is between 256 and 512 cycles access window and is on the average three times more effective than 1024 cycles access window. From the second set of graphs, we also inferred from the normalized access percentage that for most of the benchmarks 256 to 512 cycles access window size captures about 90% accesses in comparison to a 1024 cycles access window. Hence the sweet spot for access window size lies somewhere between 256 and 512 cycles.

III. SPATIAL LOCALITY

The spatial locality property of memory access suggests that, if a memory address is accessed then the probability of nearby addresses being accessed is very high. The cache line (block) size is also usually chosen using this property. However the increase in the cache line size beyond a certain size reduces the effectiveness of the caches because the fetch time for cache line increases, resulting in miss penalty, delays other bus accesses when a line is being fetched, and increases the conflict miss probability. Cache line of 32B has been shown to be most effective for most of the applications [1]. However, the applications have more spatial locality than exploited by the typical line size of 32B that can be exploited for our purpose in this paper. Various researchers have tried to exploit the spatial locality property of memory access [7]. In [8] Martin Kampe, et. al., claim to reduce the cache miss rate by 23% by exploiting the *Immediate Spatial Locality (IS)* in a 16KB cache with 32B line size. They also show that around 60% of the IS are more than one line size. From the above results it is clear that the line size is not the true representation of the available spatial locality. This interesting result motivated us to study the spatial locality in cache access pattern further to evaluate how we can use it to assign timers in a set associative manner.

A. Measurements

To measure the spatial locality we define a parameter *Degree of spatial locality*, $\delta_s = \text{deg-spat-loc}(g, w)$, as follows:

$$\delta_s = \#(\text{spac} - \text{loc} - \text{mem} - \text{refs}) / \#(\text{windows}).$$

A memory reference m to location l that occurs at time t , qualifies as a *spac - loc - mem - ref* if there is a memory

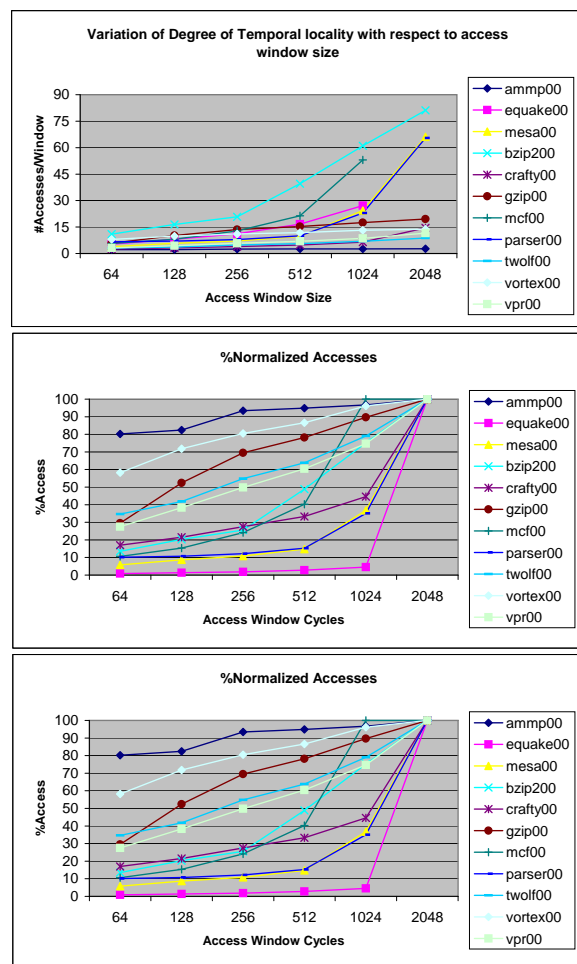
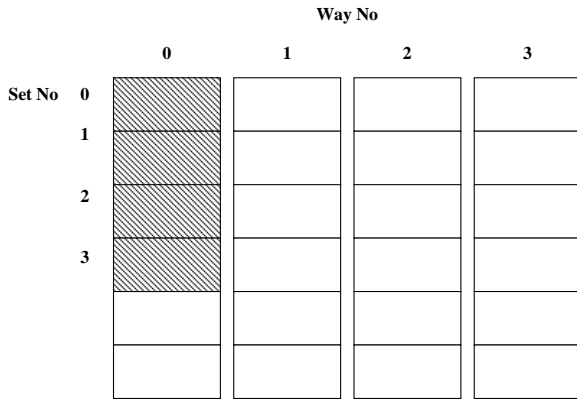


Fig. 2. Temporal Locality Characteristics

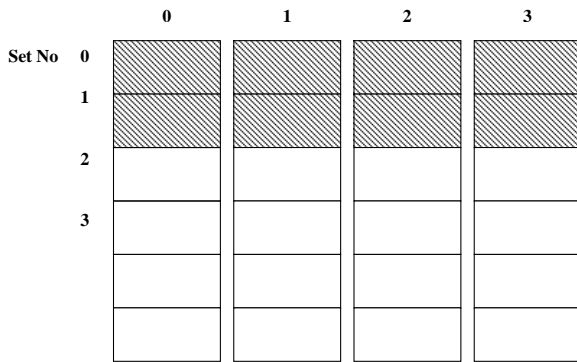
reference, m' to location l' at time t' , such that l and l' belong to a predefined group g and $t - t' \leq w$, where w is a preset unit of time. The grouping of lines into *Segment* can be achieved in many ways, taking cache organization into consideration we chose to group them into two modes.

- 1) *Way-Mode* - consecutive lines in a particular WAY in a set associative cache form a segment.
- 2) *Set-Mode* - all the lines in one or more consecutive sets form a segment.

These two modes are shown in Figure 3. In a segment consisting of 8 lines, the eight lines can be consecutive eight lines in one way (*Way - Mode*) or be the eight lines in two consecutive sets in a 4-way set associative cache (*Set - Mode*). In Section 2 we analyzed the impact of w . Even though δ_s is a function of g and w , the parameter w has very similar effect as in case of temporal locality, i.e., increasing w increases the degree of locality and vice-versa, as shown in Figure 4 for *gzip00* (spec2000 benchmark). All other benchmarks also follow similar pattern. Earlier, we also noticed that 256 or 512 cycles access window size provides a better trade-off for



Way Mode Segmentation
Lines belonging to the same way in consecutive sets are grouped



Set Mode Segmentation
All the lines belonging to one or more consecutive sets are grouped

Fig. 3. Two modes of segmentation in a 4-way set-associative cache

capturing more number of accesses with paying less penalty. We choose an access window size $w = 512$ cycles for our further study.

B. Observations

The parameter δ_s can be measured in a manner similar to δ_t except that instead of measuring accesses to one line, accesses to all the lines in a segment are measured. The average accesses per window gives the degree of spatial locality for any mode of *Segmenting*. We measure for various lines per segment for both the modes for various applications. We limited the segment size to be at most up to 16 lines. It is based on the belief that the spatial locality ceases to exist beyond that size. Our measurement results are shown in Figure 5 for the way-mode and in Figure 6 for the set-mode. In the first set of graphs in each case, we plot average # of accesses for different sizes of segments, i.e., 1, 2, 4, 8, and 16 for various benchmark applications. Notice that for the corresponding set-mode, the number of lines in a segment are four times more than in the way-mode. As we did in the case of temporal locality

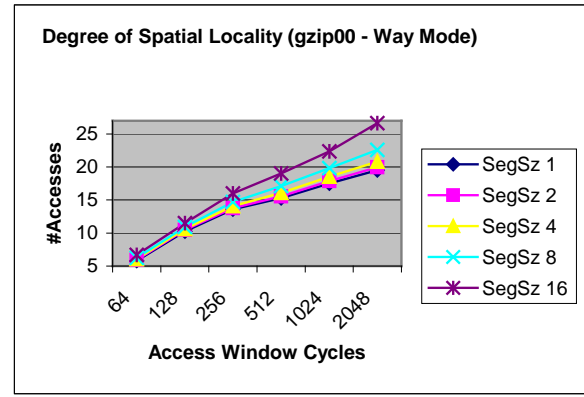


Fig. 4. Variation of spatial locality with respect to Access Window Size and Segment Size

measurement, we also plot normalized access percentage and normalized effectiveness, normalized with respect to measurements of the largest segment size, i.e., 16 for the way-mode and 64 for the set-mode as shown in second and third sets of graphs in Figures 5 and 6, respectively.

From Figures 5 for the way-mode we make the following observations. In the case of Way-Mode segmenting, four lines per segment captures more than 50% accesses with respect to 16 lines per segment. Also the effectiveness graph for most of the benchmarks is almost flat up to segment size of four. Therefore, in the way-mode segment size of four will perform better. This also confirms with the assumption with which we started i.e., some amount of spatial locality exists beyond 32B line size. Since our goal is to achieve a balance between reducing performance penalty by capturing more accesses and increasing energy saving by switching off the lines fast, the safe point to choose is segment size of four. It is to be noted that only the lines, which are actually accessed in a segment, are made active. Hence the lines that are not accessed in a segment would not incur any penalty for power consumption or reduce the drowsy percentage. The scheme will have higher benefits if the accesses to the lines within a segment have higher locality. The assumption in grouping the lines in the way-mode is that they have high spatial locality. But this may not be true for the set-mode as the replacement algorithm in a set associative cache and actual placement may have an impact on the mapping of consecutive lines. They may be mapped on different ways in the set-associative cache. Hence to validate our assumption, we grouped consecutive sets, which should capture all the spatial locality in that segment. However, this means that the segment would have four times (in general k times) more lines. The results for *SetMode* spatial locality, normalized with respect to segment size of 64 are as shown in Figure 6.

From Figure 6, it is clear that set-mode segmenting has very similar characteristics as that of the way-mode segmenting, i.e., we do not gain or loose much by grouping all the lines

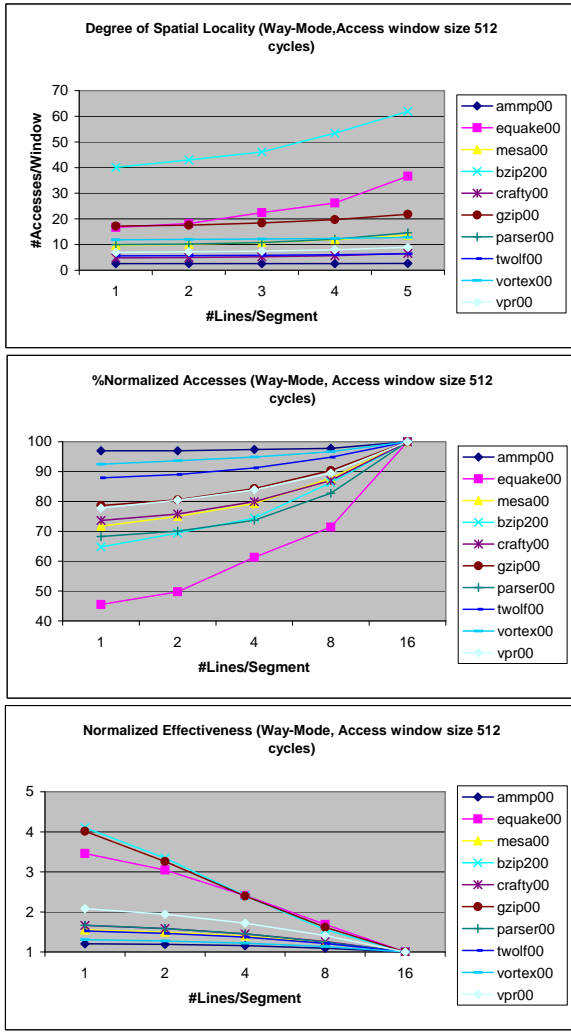


Fig. 5. Spatial Locality Characteristics (Way-Mode, Access Window 512 cycles)

in consecutive sets. However, if the accesses are not spatially located then the penalty to keep lines active for more time is higher in set-mode as there are four times more lines per segment compared to the way-mode and hence set-mode segmentation will be a disadvantage in power consumption. This fact is clearly illustrated in Figure 6. The normalized effectiveness decreases rapidly in the case of the set-mode when compared to the way-mode.

To compare the two modes of segmenting, we use number of sets that a segment encompasses in both modes, as the base, i.e., in a way-mode a segment size of 2 encompasses two lines from two consecutive sets, whereas in the set-mode segment size of eight encompasses lines in two complete consecutive sets. Thus if the assumption of way-mode capturing locality accesses in the same *way* is not accurate, then set-mode should perform better. In this comparison as the number of lines per segment in the set-mode is four times that of the way-mode,

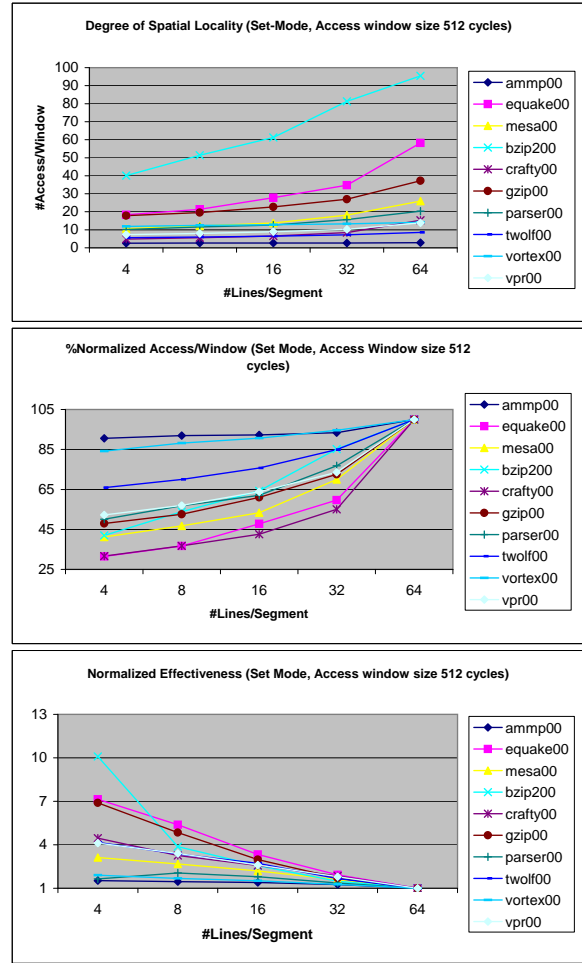


Fig. 6. Spatial Locality Characteristics (Set-Mode, Access Window 512 cycles)

we normalize the number of accesses in the way-mode with respect to the number of accesses in the set-mode. We also perform the same comparison for *effectiveness*. The two sets of results are shown in Figure 7. From the figures, we observe that the segment size of four in the way-mode captures almost 80% accesses in comparison to the set-mode that encompasses all lines from the same four sets (i.e., segment size of 16). The value of accesses captured in the way mode varies for different segment sizes. On comparing the normalized effectiveness parameter, which is shown in second set of graphs in Figure 7, we notice that effectiveness goes up as the segment size increases. However, when effectiveness results are combined with the results of the first set of graphs that captures percentages of accesses, it appears that the segment size of four performs the best overall as it is likely to capture most of the accesses with smaller window size.

IV. ANALYSIS OF OUR SCHEME

We started off with a goal of identifying the parameters which optimally defines when to switch the cache lines to

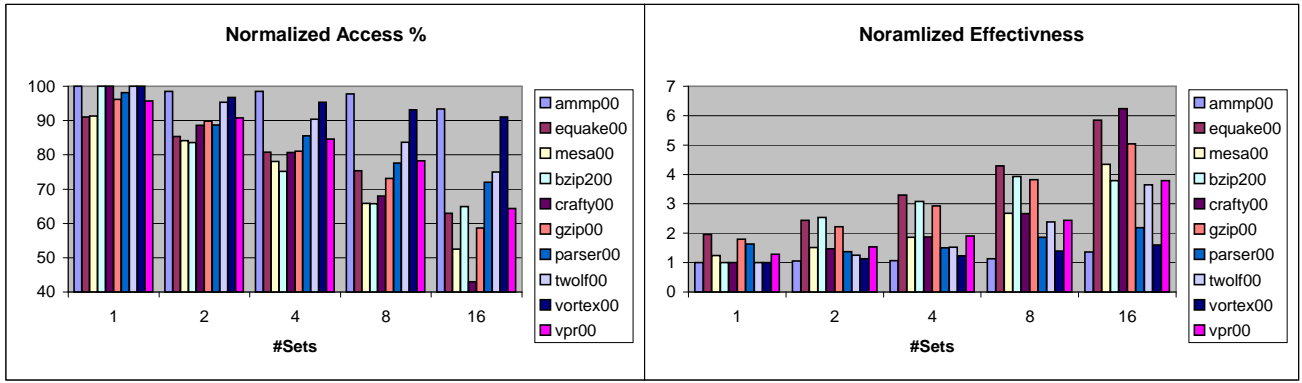


Fig. 7. Comparison of Set-Mode and Way-Mode (Access Window 512 cycles)

drowsy state and what lines need to be switched, that minimizes the power consumption for the implementation of our scheme. By analyzing spatial and temporal locality we have identified the parameters namely *access window size*, *segment mode*, and *segment size*. Moreover, we identified the need for timers that are used to switch the *segments* from active state to drowsy state. The number of timers is a parameter in our scheme. As mentioned earlier, we reduce the number of timers by using a set-associative technique. In this section, we study the effect of these parameters over energy *Gain* and arrive at an optimal set of parameters that would minimize the total energy.

We measure the effectiveness of our scheme using the parameter G . From Eqn. 1,

$$G = (1 + \rho) * q * y - \rho + K$$

where K is the energy consumed by the implementation of the scheme. When a drowsy line is accessed we need to pay a penalty both in terms of performance (access latency) and energy (transition energy from drowsy state to active state). The performance penalty is captured by ρ whereas the energy penalty is factored in parameter K . This transition energy from drowsy state to active state is directly proportional to the number of drowsy hits and is the characteristic of the underlying circuit technique. For the DVS technique, which we assume as our underlying circuit technique, proposed in [2], this is approximately one tenth of dynamic energy per cache access. Assuming dynamic energy of caches form 50% [7] of the total energy consumed by the caches, the equation of gain becomes,

$$G = (1 + \rho) * q * y - \rho - 0.5 * y * h/10$$

where h is percentage of cycles which incur drowsy hit. The component $\rho * q * y$ is very negligible as $\rho \ll y \ll q < 1$. Substituting the values of these parameters and rewriting the equation gives us,

$$G = (q - 0.05 * h) * y - \rho.$$

The parameter q is directly proportional to drowsy percentage (d). For the DVS scheme 60-65% reduction in total cache energy consumption is achievable through 100% drowsy caches. Thus factoring this into equation of G , it becomes,

$$G = (0.6 * d - 0.05 * h) * y - \rho.$$

The power consumption of 32KB on-chip caches is in the range of 15-20% [2], and substituting this value in the equation gives us,

$$G = (0.6 * d - 0.05 * h) * 0.2 - \rho.$$

For the scheme we implement, we measure the parameters, *Drowsy Hit*, *Drowsy Percentage* and *Performance Reduction* and use the above equation to find the gain achieved from the scheme. For our *scheme* we use the results of the analysis done in previous two sections. By analyzing *temporal locality* and *spatial locality* characteristics of cache access, we have derived three parameters for the *scheme*, namely *access window*, *segment mode* and *segment size*. By varying these parameters an optimal scheme can be arrived at which maximizes G . Moreover the results of access pattern analysis indicates the likely choice for these parameters. As we have already seen, for any access window increasing the segment size has similar characteristics, we can analyze the impact of these parameters independently. One more result we can use from our earlier analysis is that the way-mode clearly performs better than the set-mode. Hence we need to evaluate only the way-mode segmenting.

The impact of *access window* parameter on *Gain* is plotted in the Figure 8. The gain is normalized with respect to the maximum achievable gain $G_{max} = 0.6 * 0.2$. The gain variation graph depicts the variation in the gain from one access window size to the next. From Figure 8 it is clear that for most of the benchmarks, increasing the access window beyond 512 cycles reduces the gain. The *Gain* is maximum between 256 and 512 cycles window, which concurs with our results from our earlier analysis. The impact of *segment size* parameter on

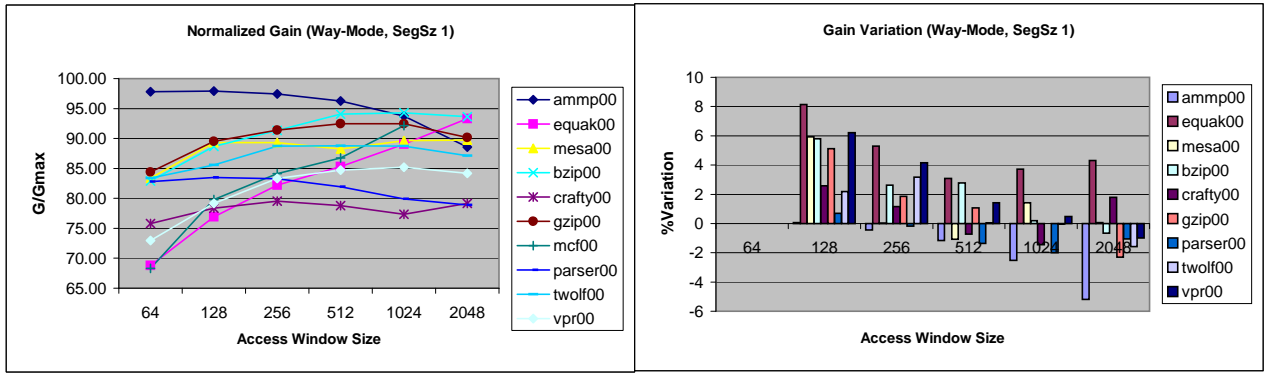


Fig. 8. Effect of Access Window Parameter on Gain

Gain is plotted in Figure 9. To measure the gain we used access window size of 256 cycles, at which for most of the benchmarks the gain is maximum. From the Figure 9 it is observed that beyond the segment size of four the gain starts decreasing, which also concurs with our earlier analysis.

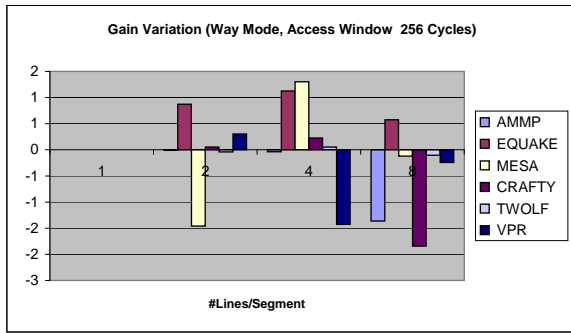


Fig. 9. Effect of Segment Size Parameter on Gain

One more interesting observation can be made from Figure 4, which helps us using the spatial locality effectively. From the figure it can be observed that increasing the segment size has the same effect as increasing the access window size. For example, the degree of spatial locality for the point (8, 1024) matches with the point (16, 512). This also explains the reason for achieving better performance with segment size of four. Using the above analysis and discussion, we suggest to use the following parameters.

- 1) Use segmented cache line architecture with a single timer for each segment.
- 2) Use the way-mode to segment the set-associative cache with a segment size of four.
- 3) A timer, *retriggerable monostate timer*, is associated with each segment which is triggered every time any line in the segment is accessed.
- 4) If no line in the segment is accessed within the *access window size* of 256 cycles all the lines in the

segment are set in the drowsy state.

Using the above set of parameters for a segmented timer-based cache drowsy control scheme the gain achieved for various benchmarks is listed in Figure 10. From the table it is clear that for all the benchmarks the drowsy percentage is above 95%. The drowsy percentage is the amount of lines kept in drowsy state averaged out over the whole execution time. The gain we have calculated is with the assumption that the power consumption of caches is 20% of the total power consumption of the chip. However in embedded processors this percentage may be as high as 40% and hence resulting in a higher gain.

	Drowsy%	Drowsy Hit%	Perf Red%	G/Gmax
ampp00	98.37	5.46	0.60	97.40
bzip200	96.85	4.24	0.40	93.13
crafty00	95.15	11.76	1.70	79.79
equake00	96.05	5.20	1.30	84.23
gcc00	96.98	7.53	1.19	86.40
gzip00	95.93	4.42	0.80	88.69
mcf00	96.31	5.22	0.68	90.20
parser00	96.85	7.40	2.30	92.82
twolf00	97.25	7.45	0.95	88.73
vpr00	96.78	7.91	1.40	84.50

Normalized gain achieved through Segment size 4, Access window size 256 and number of timers 256

Fig. 10. Performance results for various benchmarks

One observation from the table is that since on an average only less than 10% of cache lines are active at any point of time, further reduction in the number of timers is possible by grouping segments to form a *Super Segment*. The probability of accesses happening to multiple segments in the access window size determines the effectiveness of this system. The *SuperSegmenting* and associated timer sharing is explained in Figure 11. The segments sharing the timer are equally spaced in a *Way*. There can be more ways to group these segments and again we have chosen a simple and logical one. The impact of number of timers on *Gain* is as shown in Figure 12. For

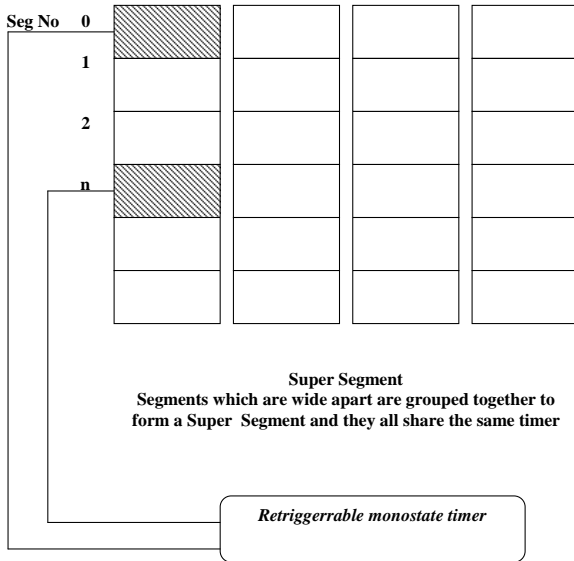


Fig. 11. A scheme to create super segments

example, if we choose to have 32 timers then segments 0,32, 64, ... will be grouped to form a *Super Segment* and will share a same timer.

As expected the reduction in number of timers by a factor of eight only reduced the gain by less than 2% for most of the applications. As we group more lines by the way of *super segment* the reduction in drowsy percentage is expected. And this reduction is overcome by the reduction in ρ . For example, for *crafty00* ρ reduces from 1.72% to 1.42%. For all the applications ρ lies within 1.5%. Thus just with 32 timers efficient energy management of cache lines can be done, with less performance penalty. Further reduction in energy consumed by the timers is possible by putting the timers themselves in *drowsy state*. The energy consumed by the timers is directly proportional to the average number of timers active and the average number of cycles for which they are active. And a timer is active only when the lines associated with it are active. Since the drowsy percentage achieved by 32 timers configuration is 90% (for most of the applications), only 10% of the timers 10% of timers contribute to the dynamic energy. The energy consumed by three 8 bit timers can be safely considered as negligible when compared with 1024 lines of 32B size. Our gain equation holds good and reflects the true gain achieved by the scheme.

V. RELATED WORK

In [2] the authors have proposed the DVS circuit technique for implementing drowsy caches and have also proposed a Policy, called *Simple Policy* to manage the drowsy caches. In this policy all the lines are switched off at equal intervals of time, which they have found to be 2000 cycles, to achieve good amount of drowsy percentage and less impact on performance. On comparing the performance of our scheme

Normalized gain achieved for various number of timers			
	256 Timers	64 Timers	32 Timers
ammp00	97.40	96.50	93.69
bzip200	93.12	91.17	87.73
crafty00	79.80	79.44	78.48
equake00	84.22	81.93	82.00
gcc00	86.40	86.20	85.98
gzip00	88.69	82.43	74.96
parser00	92.82	93.32	93.68
twolf00	88.73	88.63	88.38
vpr00	84.50	83.84	83.12
mcf00	90.20	88.01	80.40

Segment Size 4, Access Window size 256 cycles

Fig. 12. Performance results for various benchmarks for various number of timers

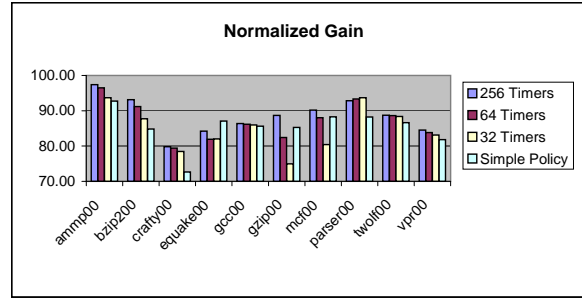


Fig. 13. Comparison of Normalized Gain achieved by our scheme with earlier proposed Simple Policy

with the *Simple Policy*, which is shown in Figure 13, we can infer that almost all the time 256 Timers configuration performs well, while 32 Timers configuration achieve less gain with some applications. The applications with which *Simple Policy* performs better than ours are those applications whose δ_t had steeper slope. The advantages of our scheme come from the fact that we exploit spatial locality together with the temporal locality. Also the random accesses tend to have lesser penalty in our scheme. as the line is kept active only for 256 additional cycles, whereas in the other scheme it may remain active for the whole window.

VI. CONCLUSION

Static power component, such as the leakage power, dominates the dynamic power consumption in the on-chip caches. Reducing the power and hence energy, is one of the important goals of today's embedded processor designers. This will be an issue even in the processor chips that are likely to be used for desktop and server systems as heat dissipation is becoming a big issue as well. As the cache size increases and feature size decreases, the static power i.e., leakage component starts dominating the energy equation. In this paper, we have studied the cache access pattern and evaluated them to arrive

at an optimal scheme to implement the drowsy cache that can reduce static power consumption. Earlier, researchers have observed that all cache lines need not be kept active at all times. Only a very few lines during a given window of time need to be actively powered from the footprint, i.e., they are accessed during that time. We significantly improve upon the earlier result by developing a better scheme for switching cache lines to drowsy state. We achieve energy reduction on the average of 88% of maximum gain achievable through the underlying circuit technique. We also compare the performance of our scheme with the earlier proposed schemes and show that we can achieve up to 6% of higher saving in energy for the benchmarks studied (with an average on 4% for all benchmarks with equal weights) without any additional performance penalty and with slightly more control hardware. This paper only lays out foundation for such a scheme. It is on-going work and need further evaluation for other cache parameters and we expect to be able to reduce the additional hardware. We will include some of these results in the final version of the paper.

Acknowledgements The authors would like to thank... ****
to be added in the final version ****

REFERENCES

- [1] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, CA, 2003.
- [2] Krisztian Falutner, Nam Sung Kim, Steve Martin, David Blaaw, Trevor Mudge - Drowsy Caches: Simple Techniques for Reducing Leakage Power, ISCA 2002
- [3] M. Powell, et. al. Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories. *Proc. of International Symposium on Low Power Electronics and Design*, 2000, pp. 90-95.
- [4] K. Nii, et. al. A low power SRAM using auto-backgate-controlled MT-CMOS. *Proc. of International Symposium on Low Power Electronics and Design*, 1998, pp. 293-298.
- [5] K. Flautner, et. al. Automatic Performance-setting for dynamic voltage scaling. *Proc. of International Symposium on Low Power Electronics and Design*, July 2001, pp. 260-271.
- [6] Doug Burger and Todd M. Austin, "The SimpleScalar Tool Set, Version 2.0", Computer Sciences Department Technical report # 1342, University of Wisconsin-Madison, June 1997.
- [7] Sanjeev Kumar, et. al. Exploiting Spatial Locality in Data Caches using Spatial Footprints. *Proc. of 25th Annual ACM/IEEE ISCA*, 1998.
- [8] Martin Kampe, et. al. Exploration of the Spatial Locality on Emerging Applications and the consequences for Cache Performance, published in *IPDPS* 2000.
- [9] Michael D. Powell, Amit Agarwal, T.N.Vijaykumar, Babak Falsafi, and Kaushik Roy, Reducing Set-Associative cache energy via Way-Prediction and Selective Direct-Mapping, *Proc. of 34th International Symposium on Microarchitecture*, 2001.