ARCHITECTURAL APPROACHES TO REDUCE LEAKAGE ENERGY IN CACHES

Shashikiran H. Tadas & Chaitali Chakrabarti
Department of Electrical Engineering
Arizona State University
Tempe, AZ, 85287.
tadas@asu.edu, chaitali@asu.edu

ABSTRACT

In this paper, we present two methods to reduce leakage energy by dynamically resizing the cache during program execution. The first method monitors the miss rate of the individual subbanks (in a subbanked cache structure) and selectively shuts them if their miss rate falls below a predetermined threshold. Simulations on SPECJVM98 benchmarks show that for a 64K I-cache, this method results in a leakage reduction of 17-69% for a 4 subbank structure and 18-75% for a 8 subbank structure when the performance penalty is <1%. The second method dynamically resizes the cache based on whether the macro-blocks (a group of adjacent cache blocks) are being heavily accessed or not. This method has higher area overhead but greater leakage energy reduction. Simulations on the same set of benchmarks show that this method results in a leakage reduction of 22-81% for the I-cache when the performance penalty is <0.1%, and 17-85% for the D-cache when the performance penalty is <1%.

1. INTRODUCTION

In recent microprocessor designs, a large part of the die size is devoted to memory components like instruction cache, data cache, buffers and tables. For instance, 30% of Alpha 21264 and 60% of Strong Arm chip area are devoted to caches and memory components. Thus a significant part of the leakage energy on a chip is due to caches. In fact, according to recent industry estimates, for 0.13um technology, 30% of L1-cache energy and 80% of L2 cache energy is due to leakage [1].

An effective way of reducing leakage energy in caches is by dynamically resizing it during application execution. The novel cache design, referred to as the *Dynamically ResIzable Instruction cache* (DRI-cache) [1] virtually turns off the supply voltage to the cache's unused sections to eliminate leakage. At the architectural level, the method exploits the variability in instruction cache usage and reduces the instruction cache size dynamically. At the circuit level, the method uses a mechanism called *gated-*V_{dd}, which reduces leakage by effectively turning off the supply voltage to the SRAM cells of the cache's unused block frames.

Another approach to reduce leakage energy is based on *selective-way* cache organization [2]. This method exploits the subarray partitioning of set associative caches to provide the ability to disable a part of the cache. This technique allows for varying the set associativity of the cache depending on the utilization of the cache *across* applications, while the cache size is set prior to the application execution.

A more recent approach operates at a lower granularity and selectively shuts off cache blocks when they have not been activated for a predetermined length of time [3]. This method can result in L1 leakage energy reducing by 4x for SPEC2000 benchmarks. Even greater reduction can be obtained by adapting the delay intervals during program execution.

In this paper, we present two promising approaches to reduce leakage energy in cache. The first approach is based on selectively shutting off the subbanks of an instruction cache while the second approach utilizes the access pattern of the macro-blocks to resize the cache (instruction or data). While the first approach has a lower area overhead, both
approaches achieve leakage energy reduction at the expense of a mild degradation in the performance. Simulations on the SPECJVM98 benchmarks show that the active portion size of the instruction cache is 32%-80% for a 4 subbank structure and 24%-78% for a 8 subbank structure when the performance penalty is < 1%. This results in a leakage energy reduction of 17-69% (average 38%) for the 4 subbank structure and 20-75% (average 52%) for the 8 subbank structure. Similar simulations on the access pattern based method show that the active portion size is 16%-67% for instruction cache when the performance penalty is 0.1% and 28%-78% for data cache when the performance penalty is 1.0%. This results in a 22-81% leakage energy reduction in the instruction cache and 17-65% in the data cache.

The rest of the paper is organized as follows. Sections 2 and 3 describe and validate the approaches based on selective disabling of the subbanks and monitoring of the access pattern of the macro-blocks. Section 4 concludes the paper.

2. DISABLING CACHE SUBBANKS

2.1 Leakage Energy Reduction mechanism

The main idea behind the proposed leakage energy reduction mechanism is to selectively shut down some of the subbanks in a subbanked cache structure based on the miss rate of the individual subbank. The scheme is implemented on a 4 as well as 8 subbank instruction cache. The mechanism is explained below. The miss rate for each subbank is calculated for every one million-access intervals. If the miss rate falls below the preset threshold value, then the least accessed subbank is disabled. Actually, the entire subbank is not disabled – the top 1K of the subbank is left active for future accesses. This is referred to as ADS or activated part of the disabled subbank. Future access (either read or write) to the disabled subbank are mapped into ADS. By leaving a fraction of the subbank active (1KB), we reduce the performance degradation significantly. To map accesses to the ADS, we mask the

higher four bits of the subbank address in the mapping function.

In order to calculate the miss rate of each subbank, we count the number of accesses to each subbank, and the number of *hit*/*misses* for that subbank. The threshold miss rate was chosen to be 1.5% implying a performance penalty of 1%. While the miss rate of the different programs varied, the average miss rate was 0.5%. The ADS size was chosen to be IK since this resulted in the largest reduction in active portion size compared to when the ADS was 0.5K or 2K. The change of cache size in our method is not as abrupt as in [1], where the cache size changes by a factor of 2 each time. For instance, a 64K cache organized into 8 subbanks can be resized into 57K, 50K, 43K, 36K, 29K, 22K, and 15K.

2.2 Overhead

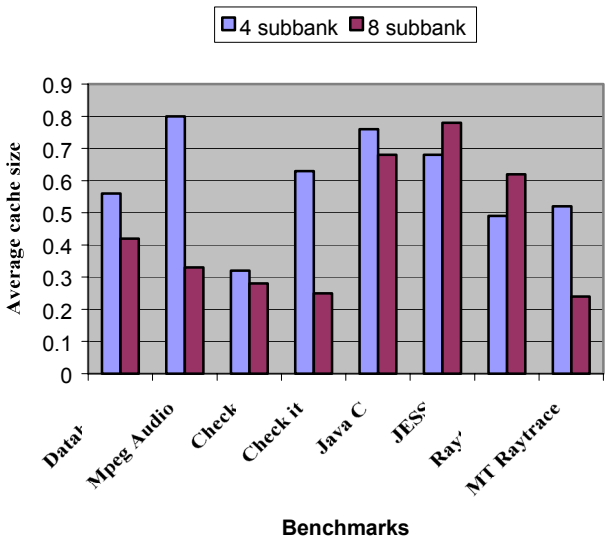

Fig 2: Average cache size for 4 and 8 subbank I-Cache for SPECJVM 98 benchmarks

Energy overhead*:* Dynamic energy is consumed in the L2 cache due to the extra L1 misses caused by the shutting down of the subbanks. However, energy reductions are gained by the subbanking structure due to selective precharging and sharing of sense amplifiers. The L1 leakage energy and the extra L2 dynamic energy are computed using the equations derived in [1].

*L1 leakage energy = active fraction * 0.91 * number of cycles*
Total energy overhead = extra L2 dynamic energy.
*extra L2 dynamic energy = 3.6 * extra L2 accesses.*

Access time overhead*:* Cache subbanking mechanism has a gate delay due to the masking bits in the mapping function, which impacts the cache lookup time. This is negligible compared to the access time reduction due the fast decoding logic in this method.

2.3 Results

Experimental setup: For cycle accurate simulation of cache behavior, we used the Shade cache simulator version 6 [4]. The base model consists of direct mapped 64KB instruction cache, 64KB data cache, and a 1MB unified L2 cache with set associativity of 2. We have used SPEC JVM98 benchmark for our experiments.

For the 4-subbank 64K instruction cache structure, if one subbank is disabled, the active portion size reduces by 23%.

Examples	Leakage Energy (J)		% Leakage
	Without Resizing	With Resizing (L1 Leakage +L2 Dynamic)	
Database	0.13	0.079 (0.073+0.06)	39
Mpeg Audio	0.34	0.282 (0.272+0.001)	17
Check	0.07	0.022 (0.022 +0)	69
Check it	1.06	0.698 (0.668+0.03)	34
Java C	0.28	0.221 (0.213+0.008)	21
JESS	0.27	0.196 (0.184+0.012)	27
Raytrace	1.18	0.597 (0.57+0.019)	49
MT Raytrace	1.08	0.581 (0.562+0.019)	46

Table 1: Leakage energy results for SPECJVM 98 benchmarks for 4 subbank I-cache.

This is because 1K of the 'disabled' subbank is still enabled and so the active portion reduces by 15/64 ≈ 23%. By keeping track of the number of subbanks disabled though the entire application execution, we obtain the average active portion size. For the 4 subbank structure, the active portion size ranged from 32% to 80% while for the 8 subbank structure, the active portion size ranged from 24% to 78%. Figure 1 describes the active portion size reduction obtained from 4 and 8 subbank cache structure. Note that in all the examples (with the exception of Jess and Raytrace), increasing the number of subbanks causes a greater reduction in the active portion.

Examples	Leakage Energy (J)		% Leakage Energy Reduction
	Without Resizing	With Resizing (L1 Leakage +L2 Dynamic)	
Database	0.12	0.054 (0.05+0.004)	55
Mpeg Audio	0.37	0.128 (0.122+0.006)	65
Check	0.07	0.022 (0.022 +0.002)	69
Check it	1.04	0.28 (0.26+0.0.02)	73
Java C	0.24	0.173 (0.163+0.0.1)	28
JESS	0.25	0.205 (0.195+0.01)	20
Raytrace	1.17	0.755 (0.725+0.03)	35
MT Raytrace	1.08	0.269 (0.259+0.01)	75

Table 2: Leakage energy results for SPECJVM 98 benchmarks for 8 subbank I-cache

Tables 1 and 2 describe the reduction in L1 cache leakage energy before and after resizing. The additional L2 cache dynamic energy overhead was found to be much smaller

than the leakage energy reduction in the L1 cache. As a result, the leakage energy is reduced by 17-69% for a 4 subbank structure and 18-75% for a 8 subbank cache structure. The leakage energy reduction was on the average higher for the 8 subbank cache structure --52% compared to 38% reduction obtained in the 4 subbank structure. All these results are for the case when the performance penalty is < 1.0%. Larger performance penalties result in larger reduction in the active area. However, the number of L2 misses increases as well and the dynamic energy overhead offsets the reduction in leakage energy.

3. MACROBLOCK ACCESS PATTERN BASED METHOD

A macroblock is a contiguous block of memory that is large enough so that the maintenance overhead is reasonable, but small enough so that the access pattern of the memory addresses within each macroblock is statistically uniform. The macroblock access pattern based method borrows some of the concepts from cache bypassing. Cache bypassing is a procedure to determine the data placement within the cache hierarchy based on the usage pattern of the memory locations accessed [5]. It uses a hardware mechanism called the Memory Address Table (MAT) to maintain and utilize the access patterns of the macroblocks to direct data placement. On a memory access, a look up in the MAT of the corresponding macroblock entry is performed in parallel with the cache access. If the cache access resulted in a hit, the access proceeds as normal. If the access resulted in a cache miss, then the cache controller must look up the MAT counter corresponding to the cache block that would be replaced to determine which data is more heavily accessed, and then predict which is more likely to be reused in cache.

For example, let the block 'j' residing in the cache have a spatial counter value of 'ctr2' in the MAT. Let the access to cache block 'i' result in a miss and now a decision has to be taken whether to bypass the access to block 'i' or replace it in the position of block 'j'. Let the spatial counter value for block 'i' be 'ctr1'. Then the counter value of the residing block, i.e. 'j', is decremented (i.e. ctr2--) and compared with that of the missing access. The actual comparison performed is ctr1 < f * ctr2. If this condition is satisfied, then the fetched data bypasses the cache, else it replaces the existing cached data. f is a fraction, which can be used to control the level of bypassing. A small 'f' prevents, while a large 'f' results in aggressive bypassing

3.1 Leakage energy reduction mechanism:

The leakage energy reduction mechanism uses the information in the MAT counters to resize the cache dynamically. The MAT counter values give an indication of whether the application is going through a heavily accessed region or not. So, if the MAT counter values are low, the cache can be downsized without significant degradation in performance. Similarly, if the MAT counter values are high, the cache should be upsized if possible. In our procedure, we introduce *sum_MATcntr* to keep the sum of the MAT counter values. If *sum_MATcntr* is less than the preset threshold value then the cache is downsized, else the cache is upsized. The preset threshold value is determined by the performance constraint. Use of the MAT counter values (or access profile) to resize makes this method effective even for applications with irregular access patterns.

Resizing the cache requires that we change the cache block lookup and mapping function dynamically. In conventional caches, there is a fixed set of index bits to locate the set to which a block maps and also a fixed number of tag bits. Since, we resize the cache, we require different number of tag bits for each of the different sizes. To meet these requirements, we maintain as many index bits to satisfy the largest possible cache size and as many tag bits to satisfy the smallest possible cache size. To find the right number of index bits, for a given cache size, we use a mask. Every time the cache downsizes, we shift the mask to the right to use a smaller number of index bits. The extra tag bits called as resizing tag bits are ignored for larger cache sizes. This mechanism is identical to the resizing method used in [1].

3.2 Overhead

The Macroblock method used here for leakage energy reduction uses hardware components to monitor and improve the cache performance. For instance, a 8-bit adder is used to increment, decrement or compare the spatial counters and a 16-bit adder is used to sum the spatial counters at the end of each interval to obtain *sum_MATcntr*. Thus there is area overhead when compared to the traditional caches.

Energy overhead*:* Energy is consumed in MAT as it maintains the monitored data from the cache. For a 1024 entry MAT with 12 tag bits, one valid bit and an 8 bit spatial counter, we need roughly 2Kb of memory. So, there is dynamic as well as leakage energy consumed in the MAT.

- For every access, energy is consumed in the 8-bit adder/subtractor, which does the increment operation, decrement operation or the compare operation. Energy is also consumed in the 16-bit sum_MATcntr adder.
- Energy is consumed by the extra L2 misses due to resizing of the cache.
- Also, dynamic energy is consumed due to the tag resizing bits maintained for resizing the cache.

We found that energy per access for the MAT structure to be 0.21nJ using Cacti [6]. We found the energy per addition for an 8-bit adder/subtractor to be 1.36pJ and that of a 16-bit adder to be 2.70pJ using SPICE for 0.18u technology.

Total energy overhead = energy due to MAT + energy due to 8-bit adder/subtractor + energy due to 16-bit adder + extra L2 dynamic energy + extra L1 dynamic energy.
*extra L1 dynamic energy = resizing bits * 0.0022 * L1 access.*
*extra L2 dynamic energy = 3.6 * extra L2 accesses.*

Access time overhead*:* In the cache bypassing mechanism, there will be a look up in the MAT for every cache access. There are three possibilities.
(a)Cache hit i.e., an entry is found. In this case, the spatial counter is incremented. This can be done in parallel with the cache access. So, this does not introduce extra time delay.
(b)Cache miss. In this case, a MAT entry has to be found for the missing access. Also, a MAT entry for the cache block, which would be replaced, has to be found. Then, a comparison operation has to take place, which will determine whether to bypass the cache or not. This

introduces a delay in the cache operation. Since, the miss rate for I-cache, which is under the study here, is very small (around 1% miss rate), this doesn't degrade the performance of the cache.

(c)No entry is found for that cache access, in which case a new entry is created. Since this can be done in parallel with the cache access, there will be no time delay.

3.3 Results

Experimental setup: This method was also implemented using the Shade cache simulator and simulated using SPECJVM98 benchmarks. The base model was the same as in section 2, namely, direct mapped 64K instruction cache and 64K data cache. The macroblock size was 1KB. The MAT had 1024 entries. Each MAT entry had one valid bit, 12 tag bits and an 8 bit spatial counter.

The preset threshold values were chosen experimentally. For instruction cache, when the performance penalty is 0.1%, the value used was 138. For data cache when the performance penalty is 1.0%, the value used was 141.

Table 4 shows the results obtained for the instruction cache with SPEC JVM98 benchmarks by using the cache bypassing method. The active portion of the instruction cache ranged from 16% to 67% for a performance penalty of 0.1%. The active portion size was reduced to as much as 16% for Check it, MT Raytrace and Raytrace applications. Note that the extra dynamic energy in L2 cache is very small because of the very small increase in miss rate. Since the dynamic energy overhead is quite small, the leakage energy reduction by this method scheme varies from 22% to 81% with an average of 60%.

Examples	Active size	L1 Leakage energy redn. (J)	Extra L2 dyn. energy (J)	Energy overhead due to MAT structure and adders(J)
Database	0.42	0.07	0.0007	0.025
Mpeg Audio	0.24	2.6	0.0008	0.060
Check	0.67	0.03	0.0004	0.012
Check it	0.16	0.90	0.003	0.04
Java C	0.29	0.17	0.001	0.047
JESS	0.28	0.18	0.001	0.048
Raytrace	0.16	1.00	0.008	0.027
MT Raytrace	0.16	0.90	0.001	0.029

Table 4: SPECJVM 98 benchmark results for I-cache using Macroblock method.

Examples	Leakage energy % Redn.	
	I-cache (penalty.1%)	D-cache (penalty 1%)
Database	37	37
Mpeg Audio	74	61
Check	22	17
Check it	80	63
Java C	51	49
JESS	52	52
Raytrace	81	64
MT Raytrace	81	65

Table 5: SPECJVM 98 benchmark results for I-cache using Macroblock method.

Table 5 shows the percentage reduction in I-cache and D-cache when this scheme is applied. Note that to obtain comparable leakage energy reductions, the performance penalty of D-cache was increased to 1% (compared to the 0.1% penalty in I-cache). This is to be expected since access patterns of D-caches are not as regular.

4. CONCLUSION

In this paper we presented two methods to dynamically resize caches. The first method shuts off the subbanks selectively and achieves reduction in leakage energy with very little extra hardware. Simulations with the SPECJVM98 benchmarks show that the active portion size of the instruction cache can be reduced to 32-80% for a 4 subbanked structure and to 24-78% for a 8 subbanked structure when the performance penalty is < 1.0%. The leakage energy reduction in L1 cache is significantly higher than the dynamic energy due to extra misses in the L2 cache resulting in the average leakage energy reducing by 38% for a 4 subbanked structure and by 52% for a 8 subbanked structure.

The second method uses counters to monitor the access patterns of the macroblocks and has higher hardware overhead. However, simulations show that the active portion of the instruction cache can be reduced from 16-67% by this method when the performance penalty is < 0.1% and that the active portion of the data cache can be reduced from 28-78%, when the performance penalty is <1.0%. As a result, the average leakage energy reduction is 60% for the instruction cache and 51% for the data cache. This is higher than the reduction obtained by the subbank based method.

Acknowledgements:

This work was carried out at the National Science Foundations's State/Industry/University Cooperative Research Centers' Center for Low Power Electronics (CLPE).

REFERENCES

[1] M. Powell, S.H.Yang B. Falsafi, K. Roy and T.N. Vijaykumar, "An energy efficient high performance deep-submicron instruction cache", IEEE Trans on VLSI, pp.77-89, Feb 2001.

[2].D.H. Albonesi, "Selective cache ways: On-demand cache resource allocation", Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 32), pp. 248-259, November 1999.

[3] S. Kaxiras, Z. Hu and M. Martonosi, "Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power", Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 34), pp. 240-251, 2001.

[4] R.F. Cmelik and D.Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling", in Sun Microsystems Labs Technical Report.

[5] T.L. Johnson, D. A. Connors, M. C. Merten and W.-M.W. Hwu, "Run-time Cache Bypassing," in IEEE Trans on Computers, pp.1338-1354, Vol.48, No.12, December 1999.

[6] Glenn Reinman and Norm Jouppi, "An integrated cache timing and power model", in Compaq Western Research Lab Technical Report.