Quantitative Evaluation of State-Preserving Leakage Reduction Algorithm for L1 Data Caches

Komiya, Reiko
Institute of Systems & Information Technologies/KYUSHU | Department of Electronics Engineering and Computer Science, Fukuoka University

Inoue, Koji
Department of Electronics Engineering and Computer Science, Fukuoka University

Moshnyaga, Vasily G.
Department of Electronics Engineering and Computer Science, Fukuoka University

Murakami, Kazuaki
Department of Computer Science and Communication Engineering, Kyushu University | Institute of Systems & Information Technologies/KYUSHU

出版情報：IEICE transactions on fundamentals of electronics, communications and computer sciences. E88-A (4), pp.862-868, 2005-04-01. The Institute of Electronics, Information and Communication Engineers
バージョン：
権利関係：

Quantitative Evaluation of State-Preserving Leakage Reduction Algorithm for L1 Data Caches

Reiko Komiya[†‡], *Non Member*, Koji Inoue[†], Vasily G. Moshnyaga[†],
and Kazuaki Murakami[‡§], *Regular Members*

Summary
As the transistor feature sizes and threshold voltages reduce, leakage energy consumption has become an inevitable issue for high-performance microprocessor designs. Since on-chip caches are major contributors of the leakage, a number of researchers have proposed efficient leakage reduction techniques. However, it is still not clear that 1) what kind of algorithm can be considered and 2) how much they have impact on energy and performance. To answer these questions, we explore run-time cache management algorithm, and evaluate the energy-performance efficiency for several alternatives.

Key words:
Low power, cache, leakage.

1. Introduction

Due to the popularization of battery operated devices such as laptop and hand-held computers, energy consumption has become a key constraint in microprocessor designs. Generally, the energy dissipation of CMOS circuits can be classified into two parts: dynamic and static. The dynamic energy is consumed by charging and discharging load capacitances in circuits, while the static energy is wasted by leakage current in non-ideal transistor operations, i.e., incomplete turning off. In the previous generation of CMOS technology, dynamic energy had large impact on total chip energy. However, with the increasing number of transistors employed in a chip and the continued reduction in threshold voltages of these transistors, leakage energy has become a major concern. For example, in Pentium4 microprocessor, 20% of total power originates in leakage [5].

A cache memory is a fast, small storage area placed on between CPU and the main memory. By concentrating the memory accesses into the on-chip cache, we can reduce the number of off-chip memory accesses, so that the processor performance is improved. The cache containing only instructions (or data) is called instruction cache (or data cache). In addition, recent high-end microprocessor tends to employ multi-level cache organization, in which the closest cache to the microprocessor is called the level 1 cache.

In state-of-the-art microprocessors, there is a tendency to increase the size of on-chip caches in order to compensate for low-speed off-chip memory accesses. Since the caches constitute a significant portion of the transistor budget of current microprocessors, leakage reduction of cache memories is especially important. For instance, in the case of a 0.07μm process technology, it has been estimated that leakage energy accounts for 70% of total cache energy [5].

To reduce the cache leakage, it is required to support at least two operation modes, **sleep** and **awake**, by circuit-level optimization [1][7]. In the awake mode the cache works with conventional SRAM leaky operations, while the sleep mode provides much less leakage but accessing to a sleep data forces some performance overhead. In order to exploit effectively the two operation modes, we need to make at least the following three decisions.

- **How is a target moved into the sleep mode?:** There are at least two options, non-state-preserving and state-preserving. The former gates the supply voltage to SRAM cells [7], thus large amount of leakage is reduced, but performance is negatively affected due to losing the data in the cells. While the latter can be implemented by optimizing the supply voltage as DVS [1] or the transistor threshold voltage as VT-CMOS. It can alleviate the performance impact, but leakage reduction is not as high as the non-state-preserving scheme.
- **When does a target change its operation mode?:** Although aggressive mode transition into the sleep mode makes significant leakage reduction, it affects negatively the performance due to an access penalty to sleeping data. Consequently, efficient algorithm to determine when the target data should be moved into the sleep or awake mode is required for improving energy-performance efficiency.
- **What is an appropriate granularity to apply the mode control?:** We need to consider the granularity to apply the low leakage mode. A coarse grain strategy transits the whole cache into sleep or awake mode at the same time. On the other hand, fine grain approaches may control word by word or line by line.

So far, many researchers have proposed efficient leakage reduction techniques [1][2][3][5][6][8][9] and they employ different alternatives for the above three parameters. Recent research has reported that state-preserving approach is better for L1 caches if L2 cache access has relatively large latency. Moreover, many

[†] The author is with Department of electronics engineering and computer Science at Fukuoka University , Fukuoka-shi, 814-0180,
[‡] The author is with Institute of Systems & Information Technologies/KYUSHU, Fukuoka-shi, 814-0001
[§] The author is with Department of Informatics at Kyushu University, Fukuoka-shi, 816-8580

approaches proposed before choose line by line fine-grain optimization. On the other hand, we can consider a number of strategies to manage the cache operation mode. Since the leakage reduction algorithm gives great impact on both energy and performance, analyzing and comparing the potential of mode control strategy are very worthwhile. In this paper, we focus on the cache management algorithm and classify them to explore the design alternatives of low leakage caches. Moreover, based on the classification, we evaluate the reduction approaches by performing trace-driven superscalar simulations. For fair evaluations, an assumption is used throughout the paper, line by line fine-grain optimization with state-preserving approach. Our goal is to guide processor designers to determine cache management strategy for making a good balance between leakage reduction and performance degradation.

This paper is organized as follows: Section 2 explains already proposed leakage reduction techniques briefly. In Section 3, we classify the cache management algorithm. Section 4 evaluates the classified design alternatives and discusses their energy-performance efficiency. Finally, in Section 5, we conclude this paper.

2. Cache Leakage Reduction Techniques

In this section, we briefly introduce leakage reduction techniques proposed before.

Ⅰ. DRI cache [8][9]

In the DRI cache, leakage energy consumption is reduced by means of cache resizing based on cache-performance requirements monitored at run-time. The number of cache misses is counted during a fixed time interval. If the miss count is less than a given threshold, we decide to reduce the cache size because the cache has quite enough capacity to maintain the performance. Otherwise, the cache size is increased to avoid performance degradation, i.e. improving cache-hit rates. The supply voltage to the SRAM cells unused is gated, i.e. non-state-preserving, thus reducing cache size contributes leakage elimination.

Ⅱ. Cache Decay [3]

When a cache miss takes place, the missed data is loaded from the next-level memory to the cache at line (or block) granularity. Generally, since programs have temporal and spatial locality of memory references, there is a tendency that the accesses to the missed line concentrate just after it is loaded into the cache and afterwards the line resides in the cache without any accesses until its eviction. In other words, each cache line survives in the cache regardless of its early last access, wasting unnecessarily a large amount of leakage energy. In the Cache Decay strategy, accesses to each cache line are monitored. If there are no accesses to a line during a given period, called decay interval, the cache predicts that the contents of the line are not expected to be reused. The supply voltage to the decayed lines is gated, i.e. non-state-preserving.

Ⅲ. Drowsy cache [1][5]

In the Drowsy cache, leakage is reduced by lowering periodically the supply voltage applied to SRAM cells. When a cache line in the sleep mode is accessed, its supply voltage is recovered to make the sleeping cache line wake up. Unlike the DRI cache or the Cache Decay, the drowsy approach employs the state-preserving optimization. Therefore, although we can not reduce the leakage energy per SRAM cell as well as the non-state-preserving schemes, cache-hit rates of conventional organization which does not employ any reduction approach can be maintained. In the drowsy cache, each line is transited into the sleep mode when a given time is elapsed and the sleeping line wakes up when it is accessed.

Ⅳ. Cache Hierarchy [6]

The Cache Hierarchy scheme reduces leakage energy consumption by exploiting the feature of memory hierarchical structure. In the case of a processor with L1 and L2 caches, if a miss occurs in the L1 cache, a copy of the target data which exists in the L2 cache is loaded into the L1 cache (we assume an L2 cache hit). After that, accesses to the loaded data hit to the L1 cache. Therefore, we can force the original data residing in the L2 cache to transit into the sleep mode, because it may be unnecessary any more. This kind of situation can be seen at other level memory hierarchies. This has originally been proposed for L2 cache leakage reduction.

3. Classifying Leakage Reduction Algorithm

In order to achieve cache leakage reduction without hurting microprocessor performance, it is very important to employ an efficient mode control algorithm. If the cache attempts to transit aggressively into the sleep mode, large amount of leakage is reduced in turn for significant performance degradation. On the other hand, where a large portion of the cache operates in the awake mode, we can not expect higher leakage reduction. In this section, we consider the possibility of mode control strategy and explore the design space of low leakage caches.

3.1 Switching to the Sleep Mode

Ideally, a cache line which is loaded from the next level memory should be turned off just after its last access. In order to approach to the ideal management, low leakage caches predict whether or not each cache line is to be reused. Table 1 summarizes the algorithm to change the cache-line state to the sleep mode. The table also presents

Table 1: Mode transition Algorithm to the Sleep Mode

Condition		Period	Validation
counter type (target)	counter ≦ threshold (cache misses[1], accesses)	$TW_{[1]}$	$synchronous_{[1]}$
	counter ≧ threshold (execution cycles[3])	$TW_{[3]}$	$synchronous_{[3]}$
	counter ≧ threshold (cache hits, no-access cycles[2])	TW	synchronous
			asynchronous
		$NTW_{[2]}$	---
event type	Load issue[4]	$NTW_{[4]}$	---

[1]DRI cache [2]Cache Decay [3]Drowsy cache [4]Cache Hierarchy

the relation of previously proposed techniques explained in Section 2. We need to consider at least the following three decisions.

- **What is a suitable condition?:** To improve the accuracy of the prediction, an appropriate condition to enter the sleep mode is required. If the condition is satisfied, the cache decides to turn-off the corresponding line. As shown in Table 1, there are two types of condition: **counter type** and **event type**. In the case of the counter type, the total number of events is counted. The condition is satisfied if the value of counter gets equal to or smaller (or larger) than a given threshold. Counting the number of cache-hits (or misses) can be used for exchanging the cache performance into the leakage reduction, while counting the number of accesses (or non-access cycles) attempts to capture the cache-access behavior. The simplest approach is to count just only the execution cycles, i.e. execution time elapsed. On the other hand, the event type does not consider previous history of events, thus the condition is satisfied just when the event occurs.

- **How long should the condition be monitored?:** The period to monitor the mode control condition effects leakage reduction efficiency. A short-period monitoring can correspond to quick transition of memory access patterns, but there is a possibility to make hasty decisions. We can consider two options for the monitoring period: **time-window (TW)** and **no-time-window (NTW)**. The TW scheme monitors the condition during a fixed time interval, while the NTW does not have any time period. The event type condition can not employ the TW scheme because it does not consider any history. For the counter type condition, the main difference between the TW and the NTW schemes is the timing for resetting the counter to zero. In the TW scheme, the counter is initialized at the end of each time window even if the condition is satisfied earlier. On the other hand, in the NTW, resetting the counter is performed just when the condition is satisfied. As presented in Table 1, the NTW scheme can not be applied to the condition which counts execution cycles elapsed, because it has a time window implicitly. Moreover, for the condition which is satisfied if the value of counter is equal to or less than the threshold, we need to select the TW scheme. Because this approach examines whether or not the condition is still satisfied after a certain time elapsed.

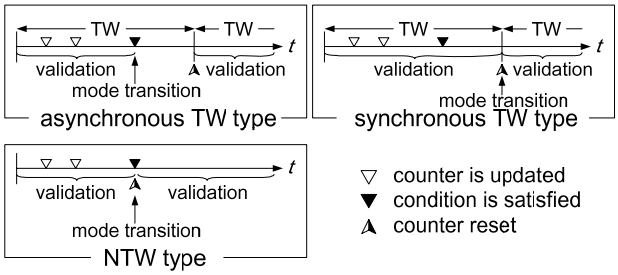

Figure 1: Mode Transition Examples

- **When is the conditional decision validated?:** For the NTW scheme, the cache transits its operation mode just when the condition is satisfied. However, for the TW type, we can consider two options: **synchronous** and **asynchronous**. The former transits the operation mode at the end of the time window, i.e. it waits for validating the conditional decision until the end of the current time window. The latter changes the operation mode soon when the condition is satisfied. The advantage of the asynchronous strategy is that it may reduce leakage energy effectively due to quick response, but it requires checking the value of counter at every counter up-date. On the other hand, although the synchronous approach causes some delay to enter the sleep mode, the frequency of conditional checking is lowered.

Table 2: Mode Transition Algorithm to the Awake Mode

Comdition		Period	Validation
counter type (target)	counter ≦ threshold (cache hits, no-access cycles)	TW	synchronous
	counter ≧ threshold (execution cycles)	TW	synchronous
	counter ≧ threshold (cache misss[1], accesses)	TW[1]	synchronous
			asynchronous[1]
		NTW	---
event type	Store issue[4]	NTW[2][3][4]	---
	Load/Store issue[2][3]		

[1]DRI cache [2]Cache Decay [3]Drowsy cache [4]Cache Hierarchy

Figure 1 gives examples for the combination of the above three parameters. In this figure, we assume that a counter type condition is employed. In this scenario, the cache has up-dated the counter two times (as marked by the white rectangles) until the value of the counter satisfies the condition (marked by the black rectangle). In the asynchronous TW type, the operation mode is changed when condition is satisfied and the counter is reset to zero at the end of time window. On the other hand, the synchronous TW type delays entering to the sleep mode until the end of time window. The NTW approach transits the operation mode and reset the counter to zero when the condition is satisfied.

3.2 Switching to the Awake Mode

The classification of the mode transition algorithm from the sleep mode to the awake mode is shown in Table 2. Fundamentally, the awake algorithm has complementary relation with the sleep mode transition algorithm except that the counter type condition with the execution cycle monitoring. Moreover, in order to take the locality of memory references into account, we added "Load or Store issue" as a candidate of events. Unlike Table 1, many leakage reduction techniques proposed before concentrate on the event type. This fact comes from the consideration of the temporal locality of memory reference. Furthermore, if accessing to a sleep data is not allowed, i.e. the accesses can be performed exactly after the target data is woken up, the event type should be selected. This is because the counter-type approaches need to keep counting the number of events during a monitoring period.

4. Evaluation

Based on the discussion in Section 3, we evaluate the energy-performance efficiency of leakage reduction algorithm. In order to perform fair comparison, we introduce the following assumptions.

- The mode control is applied at cache-line granularity. A number of techniques proposed before used this assumption [1][3][6].
- The state-preserving scheme is employed for sleep mode implementation, because it can maintain cache-hit rates.
- It is allowed to access to sleeping cache lines without wake-up transition. However, in this case, some penalty for cache-access time is caused. We call the overhead **Sleep-Hit-Penalty (SHP)**.
- Regardless of the algorithm, any cache misses wake up the target line. The line loaded from the next level memory to the L1 cache is initially set to the awake mode.
- Tag data are always in the awake mode, because it does not give large impact on the total leakage but affects significantly the cache-access time [1].

4.1 Leakage Energy Model

In this evaluation, total leakage energy consumed in a L1 data cache ***LEtotal*** is approximated as follows:

$$LEtotal = CC * LEline * Nline \quad (1)$$

$$CC = CCconv + CCextra, \quad (2)$$

$$LEline = SR * LEsline + (1-SR) * LEaline, \quad (3)$$

where *CC* is the total execution time in terms of clock cycles, *LEline* is the average leakage energy of a cache line consumed in one clock cycle, and *Nline* is the number of lines in the L1 data cache. *CC* can be presented by the total execution time without any cache-leakage optimization *CCconv* and the penalty caused by accessing to sleep-mode lines *CCextra*. On the other hand, *LEline* is given by introducing the sleep rate *SR* which is a rate of lines working in the sleep mode. *LEsline* and *LEaline* are the cache-line leakage energy dissipated in one cycle when it works on the sleep and awake mode, respectively. In a non-optimized conventional cache, both *CCextra* and *SR* are zero.

Table 3: Evaluated Models

Model	To Sleep Mode			To Awake Mode		
	Condition	Period	Validation	Condition	Period	Validation
EC-EC	Counter (EC)	TW	Sync.	Counter (EC)	TW	Sync.
EC-NAC	Counter (EC)	TW	Sync.	Counter (NAC)	TW	Sync.
EC-S	Counter (EC)	TW	Sync.	Event (S)	NTW	---
EC-LS	Counter (EC)	TW	Sync.	Event (LS)	NTW	---
NAC-EC	Counter (NAC)	NTW	---	Counter (EC)	TW	Sync.
NAC-NAC	Counter (NAC)	NTW	---	Counter (NAC)	TW	Sync.
NAC-S	Counter (NAC)	NTW	---	Event (S)	NTW	---
NAC-LS	Counter (NAC)	NTW	---	Event (LS)	NTW	---

EC: Execution Cycles NAC: No-Access-Cycles S: Store LS: Load/Store

4.2 Experimental Environment

In this evaluation, we developed a cache simulator to estimate the cache leakage energy. To obtain address trace information, we used the ZonC trace-driven SPARC64 simulator which models the detail of SPARC64 out-of-order execution and its error is only equal to or less than 5% compared to a real chip design [4]. If another instruction-set architecture such as X86 is assumed, we may have different results. This is because it will produce difference memory-access behavior. However, in this paper, each cache is evaluated with not absolute values but relative ones. Namely, the execution time and the amount of leakage reduced are normalized to those produced by the non-optimized cache organization, respectively. Therefore, we believe that our evaluation results can be useful even if we employ a difference processor architecture. Based on a SPARC64 design, 128 KB two-way set-associative data L1 cache with a 64B line size is assumed, thus *Nline* in equation (1) is 2K.

To measure *CC* and *SR*, we executed 12 of integer programs and 14 of floating-point programs from the SPEC2000 benchmark. In this simulation, the first 50 million instructions are skipped to capture stable execution behavior and the following 10 million instructions are used for measurements. On the other hand, for the leakage energy, we refer the reported value in [1] that the ratio of *LEaline* and *LEsline* is 100 to 8. This result was obtained by performing circuit-level simulation with a 0.07μm Berkeley Spice model. Moreover, the dynamic energy consumption accompanying cache accesses and mode transitions is not taking into consideration.

The cache models to be evaluated are shown in Table 3. Here, we choose two counter-type conditions for the sleep algorithm, one counts execution cycles (EC) and the other counts no-access cycles (NAC), because they are representative conditions in line-base leakage optimization techniques[1][3]. On the other hand, for the algorithm to wake lines up, two of event-type conditions, store (S) and load/store (LS), and two of counter-type conditions, EC and NAC, are considered. For the model notation, the left and right characters indicate the condition to enter the sleep and awake modes, respectively. We assumed that the threshold to satisfy the conditions is set to 4K cycles based on the reference [1].

In this evaluation, we focus only on L1 data caches. The energy efficiency of low-leakage caches depends on the memory access behavior. Therefore, we believe that the similar trends will be observed on next-level data caches, e.g. L2 data caches. On the other hand, for instruction caches, access behavior depends on the program control flow. Therefore, our evaluation results may be inapplicable to instruction caches.

4.3 Results

Figure 2 reports the normalized results to the same configuration conventional cache for the leakage reduction and performance overhead. The results are average of all benchmarks and the effects of SHP (Sleep Hit Penalty) are also evaluated.

As shown in Figure2, the cache models employing the event-type awake algorithm, EC-S, EC-LS, NAC-S, and NAC-LS, produce much better results than the caches with the counter-type awake strategy, EC-EC, EC-NAC, NAC-EC, and NAC-NAC. In order to clarify the advantage of the event-type awake algorithm, Figure 3 and Figure 4 present the energy reduction rates for all benchmark programs on EC-LS and EC-NAC, respectively. From the figures, we see that EC-LS is superior to EC-NAC for all benchmark programs. This is because the counter type approach wakes up all of the cache lines at every fixed interval without any consideration for access behavior. As a result, a number of lines whose last access has already been completed are woken up, degrading the efficiency of leakage reduction. On the other hand, a mode transition takes place only when a load or store instruction is issued

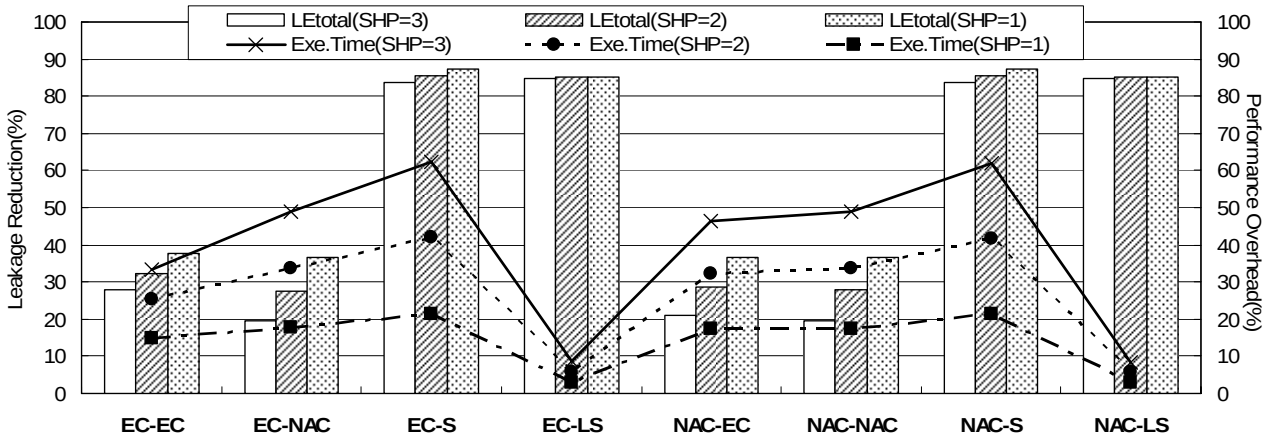

Figure 2: Leakage Energy and Performance (Average of all benchmarks)

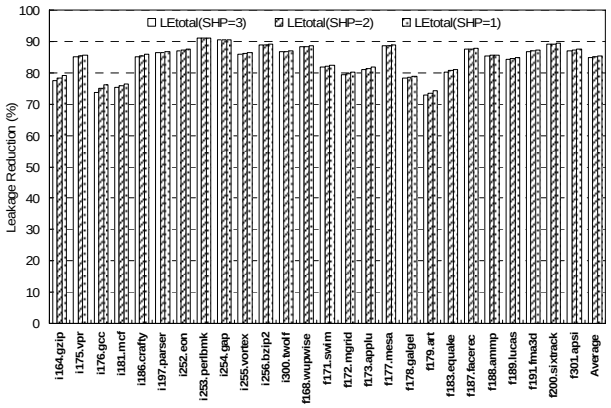

Figure 3: Leakage Reduction in EC-LS

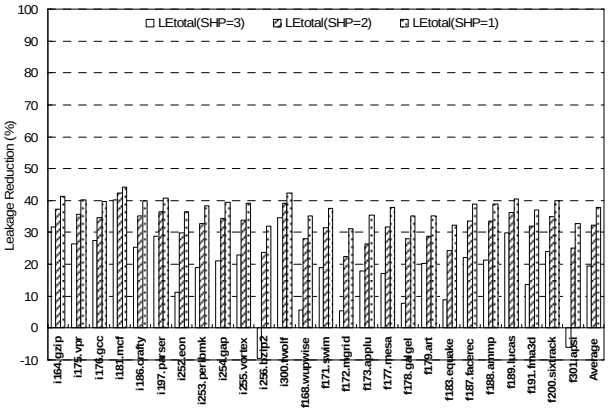

Figure 4: Leakage Reduction in EC-NAC

in the event type approaches, so that the majority of cache lines keep to residing in the sleep mode.

Next, we compare the caches with the event-type awake algorithm, EC-S, EC-LS, NAC-S, and NAC-LS, from the performance point of view. From Figure 2, we see that the performance overhead caused by the Load/Store awake strategy, EC-LS and NAC-LS, is smaller than that caused by the only-Store awake strategy, EC-S and NAC-S. For the detail consideration, in Figure 5, we show the performance overhead caused by EC-LS and EC-S for all benchmark programs. The result comes from the fact that memory references have temporal and spatial locality regardless of the access operations, load or store. This negative effect becomes clear with increase in the SHP. If the SHP is three clock cycles, the performance is degraded by about 20% on EC-S. So, it turns out that LS is the best awake algorithm from the viewpoint leakage and performance.

From the results discussed above, we conclude that EC-LS and NAC-LS approaches are the most promising

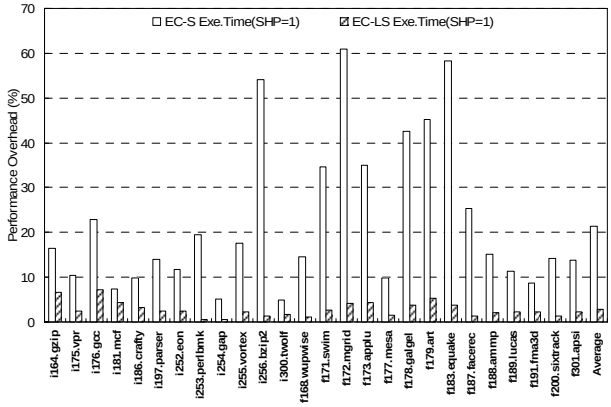

Figure 5: Performance Overhead in EC-S, EC-LS

algorithm to achieve low-leakage and low-performance-overhead L1 data caches. For all benchmarks, EC-LS achieves more than 70% of leakage reduction.

4.4 Complexity

In this section, we discuss the complexity of each cache model. For the sleep algorithm, we have two options; EC (Execution-Cycles) and NAC (Non-Access-Cycles) as showed in Table 3. As explained in Section 3.1, NAC counts the number of clock cycles elapsed without any accesses. In other words, this algorithm requires to monitoring whether or not each cache line is accessed. Therefore, a counter and a special circuit for the monitoring are needed per cache line. On the other hand, EC does not consider the cache-access behavior, i.e. only a counter per cache line is implemented, thus it is more complexity effective than the NAC approach. The awake algorithm has two counter-type options, EC and NAC, and two event-type options, L (Load) and LS (Load/Store). For the counter-type alternatives, EC is more complexity effective than NAC as explained above. In the event-type alternatives, the cache needs to know the type of current cache access, load or store. However, this information is originally provided from the microprocessor to the cache in order to perform the read or write operation. Therefore, hardware complexity caused by the event-type approach is trivial.

In Section 3.3, we have found that the Load/Store event-type awake algorithm, EC-LS and NAC-LS, is appropriate to low-leakage caches. In addition, as discussed above, EC awake algorithm is more complexity effective than NAC strategy. Therefore, we conclude that EC-LS approach, which is employed by the drowsy cache, is the best algorithm.

Table 4: Complexity

algorithm	implementation circuits
EC	counter
NAC	counter access-behavior monitor
S	access-behavior monitor
LS	access-behavior monitor

5. Conclusions

In this paper, run-time cache management algorithm to reduce leakage energy consumption has been classified. In addition, we have evaluated energy-performance efficiency of several models which employ line by line optimization. As a result, we have found that EC-LS model achieves the best performance-energy efficiency with relatively low complexity. This model makes each cache line enter to the sleep mode at every fixed interval and sleeping lines are transited to the awake mode when they are accessed. In this evaluation, the dynamic energy overhead caused by mode control units is not contained, and quantitative evaluation of complexity is omitted. Our future work is to evaluate with more accurate energy model including not only static energy but also dynamic energy consumed for the run-time cache management. Another future work is to discuss the complexity to manage cache operation based on real circuit designs.

Acknowledgments

We thank Masayuki Ikeda, Takumi Maruyama, Akira Katsuno and Mariko Sakamoto who gave us many advices. This research was supported in part by the Grant-in-Aid for Creative Basic Research, 14GS0218, 14702064, 14102027.

References

[1] K.Flautner, N.S.Kim, S.Martin, D.Blaauw, and T.Mudge, "Drowsy Caches: Simple Techniques for Reducing Leakage Power," *Proc. of the 29th Int. Symp. on Computer Architecture*, pp.148-157, May 2002.

[2] S.Heo, K.Barr, M.Hampton, and K.Asanovic "Dynamic Fine-Grain Leakage Reduction Using Leakage-Biased Bitlines," *Proc. of the 29th Int. Symp. on Computer Architecture*, pp.137-147, May 2002.

[3] S.Kaxiras, Z.Hu, and M.Martonosi, "Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power," *Proc. of the 28th Int. Symp. on Computer Architecture*, pp.240-251, June 2001.

[4] M.Sakamoto, A.Katsuno, A.Inoue, T.Asakawa, H.Ueno, K.Morita, Y.Kimura, "Microarchitecture and Performance Analysis of a SPARC-V9 Microprocessor for Enterprise Server Systems," *Proc. of the 9th Int. Symp. on High-Performance Computer Architecture*, pp.141-152, Feb.2003

[5] N.S.Kim, K.Flautner, D.Blaauw, and T.Mudge, "Drowsy Instruction Caches; Leakage Power Reduction using Dynamic Voltage Scaling and Cache Sub-bank Prediction," *Proc. of the Int. Symp. on Microarchitecture,* pp.219-230, Nov. 2002.

[6] L.Li, I.Kadayif, Y-F.Tsai, N.Vijaykrishnan, M.J.Irwin, and A.Sivasubramaniam, "Leakage Energy Management in Cache Hierarchies," *Proc. of the 11th Int. Conf. on Parallel Architectures and Compilation Techniques*, pp.131-140, Sep.2002.

[7] M. Powell, S. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories, " Int. Symp. on Low Power Electronic and Design, pp.90-95, July 2000.

[8] S.H.Yang, M.D.Powell, B.Falsafi, K.Roy, and T.N.Vijaykumar, "An Integrated Circuit / Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches," *Proc. of the 7th Int. Symp. on High-Performance Computer Architecture*, pp.147-157, Feb.2001

[9] S.H.Yang, M.D.Powell, B.Falsafi, and T.N.Vijaykumar, "Exploiting Choice in Resizable Cache Design to Optimize

Deep-Submicron Processor Energy-Delay," *Proc. of the 8th Int. Symp. on High-Performance Computer Architecture*, pp.151-161, Feb.2002.