Snug Set-Associative Caches

Reducing Leakage Power while Improving Performance [*]

Jia-Jhe Li
Department of Computer Science
National Taiwan Ocean University
Keelung 20224
Taiwan
jjli@pllab.cs.ntou.edu.tw

Yuan-Shin Hwang
Department of Computer Science
National Taiwan Ocean University
Keelung 20224
Taiwan
shin@cs.ntou.edu.tw

ABSTRACT

As transistors keep shrinking and on-chip data caches keep growing, static power dissipation due to leakage of caches takes an increasing fraction of total power in processors. Several techniques have already been proposed to reduce leakage power by turning off unused cache lines. However, they all have to pay the price of performance degradation. This paper presents a cache architecture, the *Snug Set-Associative* (*SSA*) cache, that does not only cut most of static power dissipation but also reduces execution times. The SSA cache reduces leakage power by implementing the *minimum set-associative* scheme, which only activates the minimal numbers of ways in each cache set, while the performance losses incurred by this scheme are compensated by the *base-offset load/store queues*. These two techniques are both developed based on the principle of locality and they work together nicely—experimental results show that the minimum set-associative scheme can cut static power consumption of the L1 data cache by 90% on average for SPECint2000 benchmarks, while the execution times are reduced by 3% when the default 8-entry load/store queue is modified to the base-offset design. Furthermore, the SSA cache can trim the leakage power of L2 data cache by 96% on average while still accomplishing a 3% reduction in execution times.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Cache Memories

General Terms: Algorithms, Design, Measurement

Keywords: Set-associative caches, Leakage power

1. INTRODUCTION

As the minimum feature size gets smaller and more transistors are packed densely onto processors, static power dissipation due to leakage takes an increasing fraction of total power in processors. It is estimated that leakage power will be the dominant form of power dissipation soon [6, 13]. Furthermore, nowadays on-chip caches comprise most of the transistor counts of processors. As a result, leakage power of caches accounts for a significant fraction of the overall chip energy.

[*]This research was supported in part by NSC grant NSC93-2213-E-019-001 and MOEA project 94-EC-17-A-01-S1-034

Several techniques have already been proposed to reduce leakage power of data caches [1, 7, 8, 10, 12, 23]. They all have developed polices and implementations to turn off or put to sleep data cache lines that are not likely to be reused. The rationale of this strategy is that cache lines usually have a period of "dead time" before their data are discarded, and the activity in data caches is only centered on a small subset of cache lines during a fixed period of time due to the property of locality. However, these methods all incur performance penalties since reloading data or waking up cache lines takes time and energy. To alleviate this problem, this paper presents a cache architecture, the *Snug Set-Associative* (*SSA*) cache, that does not only cut most of static power dissipation but also improves performance.

The SSA cache reduces leakage power by implementing the *minimum set-associative* (*MSA*) scheme, which only activates the minimal numbers of ways in every cache set. Similar to those proposed techniques, this theme turns off the cache lines that are idle over a pre-set number of cycles. However, this scheme does not activate an additional cache block right away once a cache miss happens, as done by these techniques. Instead, an additional way is activated only when the target block of the current miss has been referenced before within a certain time interval. On the other hand, if the data of the current miss had not been accessed recently, the data will be loaded to a cache line in the target cache set and the least recently used (LRU) cache block in the same cache set will be deactivated. In other words, the number of active ways remains the same. An additional way is allocated only to avoid thrashing in the same cache set. As a result, only the minimal numbers of ways in each cache set remain active. This scheme turns off more cache lines than these existing techniques, and consequently reduces more leakage power.

The performance losses caused by the MSA scheme will be compensated by the *base-offset load/store queue* (*BO-LSQ*) design. In addition to storing the effective address of a load or store instruction, the default load/store queue (*LSQ*) is modified to accommodate the offset and the content of the base register in the same entry as well. The design is developed based on locality: as data in the cache blocks of the current working set are accessed repeatedly, the same addresses are computed again and again. In other words, the load/store instruction currently being dispatched might find an entry in BO-LSQ with matched offset and base value. As a result, a couple of pipeline stages for address computation of some load/store instructions can be bypassed and hence execution times can be reduced.

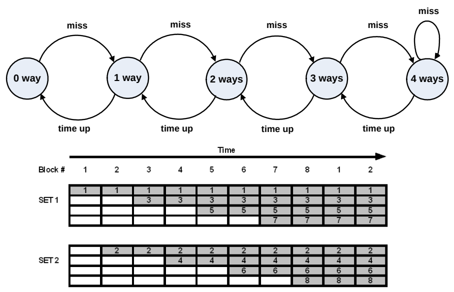
The main results of this paper are as follows:

- SSA can cut static power consumption of L1 caches by 90% for SPECint2000 benchmarks, while the execution times are reduced by 3% on average when the default 8-entry LSQ is modified to the BO-LSQ design. The average active ratio of cache blocks is less than 5%.

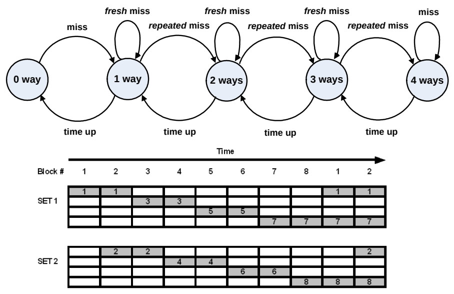
- SSA can trim the leakage power of L2 data cache by 96% on average, while still accomplishing a 3% reduction in execution times with an 8-entry BO-LSQ. This result is significant since nowadays L2 caches usually comprise much of the chip areas and transistor counts of processors.

2. MINIMUM SET-ASSOCIATIVE CACHE

Caches are very power-inefficient due to locality since on most of SPEC benchmarks the working set — the fraction of unique cache lines accessed during an interval of thousands of instructions — is small [7]. In addition, a cache block is not likely to be referenced in the future when it leaves the active window [4, 10]. Therefore, it is reasonable to deactivate cache blocks when they hold data not likely to reused.



(a) Set-Associative



(b) Minimum Set-Associative

Figure 1: Set-Associative vs. Minimum Set-Associative

2.1 Policy

The SSA cache reduces leakage power of caches by deploying the minimum set-associative (MSA) scheme, which tries to activate as few cache blocks as possible. The MSA scheme differs from the existing leakage-reducing techniques in the timing of when to activate an extra cache block in the target cache set. Both Cache Decay and Drowsy Cache allocate a new block when a miss happens and deactivate a block when its time window expires [7, 10]. As a result, the numbers of active ways in a 4-way set-associative cache will follow the state diagram shown in Figure 1(a). On the other hand, an extra way is allocated by MSA only when the new block and currently active blocks might be accessed in the near future. The reason is because it has been observed that the cache "efficiency" is low, which is the fraction of data that will be a read hit in the future before any evictions or writes [4]. In other words, cache blocks that are not actively accessed are not likely to be used in the future.

Since set-associative caches can avoid the thrashing problem of direct-mapped caches, MSA exploits this idea and activates more ways only when it can be identified the situation has occurred that a piece of code repeatedly alternates between accessing different locations that map to the same cache set [20]. In other words, MSA increases the number of active ways of a cache block only if the data of current miss has be accessed before within the time window. In order to determine whether the thrashing problem will occur if the number of active ways in the target cache set is not increased, MSA classifies all misses within a time interval into two categories:

- *repeated* — the miss to a block that was active before within the current window, and
- *fresh* — the miss that is not *repeated*.

When a *fresh* miss occurs, no extra way will be activated in the target cache set. Instead, an active block chosen by the LRU policy in the cache set will be deactivated and a new block will be allocated to accommodate the new data. On the other hand, when a *repeated* miss is identified, an extra way will be activated to avoid the thrashing problem. As as result, the numbers of active ways in a 4-way MSA cache will follow the state diagram shown in Figure 1(b).

Consider a 4-way set-associative cache with only 2 sets that is referenced by a sequence of addresses mapped to memory blocks 1 to 8 and then blocks 1 and 2. When block 3 is referenced, the set-associative scheme will activate another block in Set 1 and hence two cache lines are active to hold the data of blocks 1 and 3 (the shaded blocks), as shown in Figure 1(a). On the other hand, the MSA scheme considers it as a *fresh* miss, and then disables the currently active block after allocating a new cache line in Set 1 for block 3. As a result, there is still only one active way in Set 1 after time step 3, as depicted in Figure 1(b). After block 8 is accessed, all the cache block in the cache are active for the set-associative scheme, while MSA has only two working cache lines. When blocks 1 and 2 are referenced again, the set-associative scheme will do nothing since both cache lines are active. On the other hand, MSA will reactivate cache lines for blocks 1 and 2 as both are *repeated* misses.

2.2 Implementation

In order to differentiate *fresh* misses from *repeated* misses, there must be a way to tell if the cache block of the current miss was active within the current time interval. This paper solves this problem by placing a third state, called the *sleep* state, between the *active* and *off* states. In the *sleep* state, the data field of a cache block is inactive while the tag field remains active. Consequently, when a miss happens, the miss will be categorized as a *repeated* miss if the tag field of the request matches the tag fields of any *sleep* blocks in the target cache set. Otherwise, the miss will be classified as a *fresh* miss.

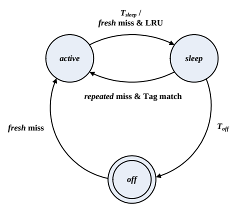


Figure 2: States of SSA Cache Blocks

Figure 2 depicts the state transitions of an SSA cache block under MSA. When a *fresh* miss happens, an *off* block will be activated and enters the *active* state. On the other hand, when a *repeated* miss takes place, the tag field of the request will match one of *sleep* blocks in the cache set, and the matched block will make the transition from the *sleep* state to the *active* state. An *active* block will be put to sleep when one of two conditions occurs: (1) when a *fresh* miss happens and this block is the LRU block in the set, or (2) when the pre-set time slot T_{sleep} expires. Finally, a block in the *sleep* state will be turned off when it is idle over a pre-set time window T_{off}.

These three states can be implemented by combining the techniques in Cache Decay [10] and Drowsy Cache [7], since they can be modeled as

- *active*: normal mode
- *sleep*: data field is put into a low-power drowsy mode so that it can be reactivated with only 1-cycle delay when necessary
- *off*: both tag and data fields are turned off as the decayed (off) mode in Cache Decay

Furthermore, Cache Decay can be implemented using the same circuit fabric as in Drowsy Cache [14]. Therefore, the dynamic voltage scaling (DVS) technique implemented in Drowsy Cache can be extended to supply three types of voltages: V_{DD} (1V), $V_{DD}Low$ (0.3V), and V_{SS} (0V), which correspond to *active*, *sleep*, and *off* states respectively.

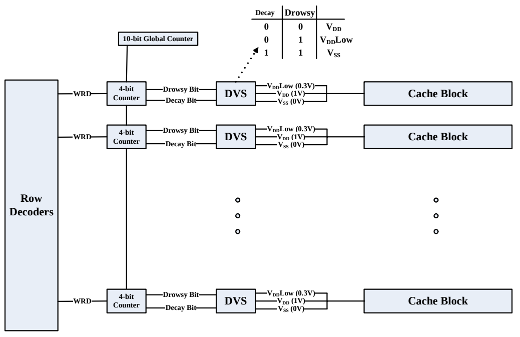


Figure 3: SSA Organization

Figure 3 displays the organization of SSA. The ticks for 4-bit cache-line counters are fed from a 10-bit global counter, which would come for free from on-chip hardware performance monitors [5, 22]. When a local counter reaches zero, one of two bits, *Drowsy* and *Decay*, will be set to signal a state transition. Both bits are cleared when the cache line is active, and the output of DVS will be V_{DD}. When T_{sleep} is up, the *Drowsy* bit will be set and the DVS output will be dropped to $V_{DD}Low$ to put the cache line into the drowsy mode. Finally, when T_{off} is up, the *Decay* bit will be set as well and the cache line will be turned off by lowering the DVS output to V_{SS}.

2.3 Energy

Static power dissipation of an SSA cache consists of leakage power of active blocks, leakage power dissipated by blocks in the *sleep* mode, and the energy consumption incurred by the extra L1 misses that are introduced when *off* cache lines are accessed. The leakage power of active blocks E_{active} is equal to the product of the number of all active blocks during the run of the program B_{active} and the static energy consumed by a cache line every clock cycle E_{block}, i.e.,

$$\begin{aligned} E_{active} &= B_{active} \times E_{block} \\ B_{active} &= \sum_{i=1}^{NewTotalCycles} ActiveBlocks_i \end{aligned}$$

where $NewTotalCycles$ is the total number of clock cycles for the program when MSA is applied and $ActiveBlocks_i$ is the number of active blocks at ith clock cycle.

Static energy of cache lines in *sleep* mode E_{sleep} is the sum of energy generated by the active tag fields and the drowsy data fields:

$$\begin{aligned} E_{sleep} &= B_{sleep} \times (F_{tag} + F_{data} \times R_{sleep}) \times E_{block} \\ B_{sleep} &= \sum_{i=1}^{NewTotalCycles} DrowsyBlocks_i \end{aligned}$$

where F_{tag} is the fraction of the tag fields in the cache line, F_{data} is the fraction of the data field ($F_{tag} + F_{data} = 1$), and R_{sleep} is the ratio of static energy of a drowsy block relative to an active block. In addition, turning off cache blocks introduces extra L1 misses and hence incurs some energy overhead $E_{L2access}$:

$$E_{L2access} = ExtraMiss \times R_{L2} \times E_{block}$$

R_{L2} relates the overhead due to an additional miss to static leakage energy of a single clock cycle in the L1 cache.

Performance of SSA caches will be evaluated by normalized leakage power, which uses the static power consumption of the original cache organization as the base:

$$\begin{aligned} Leakage &= \frac{E_{active} + E_{sleep} + E_{L2access}}{E_{origin}} \\ E_{origin} &= BlockNumber \times E_{block} \times TotalCycles \end{aligned}$$

where $BlockNumber$ is the number of cache blocks in the cache and $TotalCycles$ is the original number of clock cycles for the program.

3. BASE-OFFSET LOAD/STORE QUEUE

A dynamically scheduled processor usually embeds a load/store queue (LSQ) to provide the following functions: (1) buffering store addresses and values for in-order retirement, (2) forwarding in-flight store values to load, (3) detection of load/store ordering violations, and (4) detection of consistency violations [3, 17]. Recently, couples of techniques have been proposed to reduce power consumption or improve cache load time by buffering data in the LSQ [16, 18]. This section presents a design, the base-offset load/store queue (BO-LSQ), that can improve performance by modifying the base LSQ to accommodate the offsets and the contents of the base registers of load or store instructions. The design is developed based on the observation that store-load and load-load memory dependences exists in many programs [18]. The dependences can be correlated directly if the same base and offset are identified in any entry of the BO-LSQ, and then a couple of pipeline stages of load instructions can be bypassed to achieve speedup.

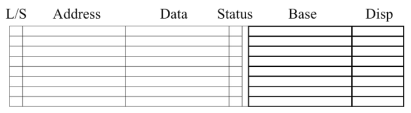


Figure 4: Base-Offset Load/Store Queue

Figure 4 depicts the details of the BO-LSQ organization. The *L/S* flag identifies the instruction as being a load or a store. The *address* field contains a memory address to access, the *data* field contains the data to store for store instructions, and the *status* bit contains the execution state of the instruction [16].

BO-LSQ adds two more fields to the base LSQ: the *base* field that keeps the content of the base register, and the *offset* field that holds the offset value of a load or store instruction. The design is based on the format of the load and store instructions of Alpha [11]:

```
load   Ra, disp (Rb)
store  Ra, disp (Rb)
```

where the effective address is computed by adding the value of *disp* and the content of the base register *Rb*. As a result, a load or store instruction is executed in two phases: address computation (*AGEN*) and memory access (*L/S*).

Figure 5 shows the pipeline stages of Alpha 21264 simulated by the SimpleScalar [2]. In the *Dispatch* stage, multiple instructions are decoded, dependences among instructions are detected, and the decoded instructions are renamed to reorder buffer (RUU) entries. In addition, the instructions will be put to the issue queue, and the values of register operands will be loaded in the *Dispatch* stage if they are available. The *Execute* stage performs the computation when operands are ready, and the result will be broadcasted in the *WriteBack* stage.

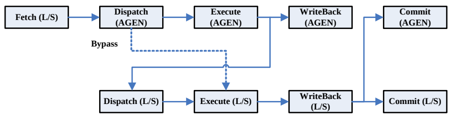


Figure 5: Pipeline Stages and Bypassing

When a load or store instruction enters the pipeline, the *AGEN* and *L/S* phases will be treated as two consecutive and dependent instructions. After the effective address is computed at the *Execute* stage, the result will be sent to the next pipeline stage and meanwhile be forwarded to the *Dispatch* stage of the *L/S* phase as well. Therefore, the *WriteBack* stage of *AGEN* and the *Dispatch* stage of *L/S* can then be executed simultaneously in the same clock cycle. In the *Execute* stage, the computed address will be compared with the addresses in the base LSQ and the tags of L1 cache to determine if the operand can be obtained from a normal store-to-load forwarding in LSQ or be read from a cache block.

The *Bypass* path in Figure 5 shows that a load or store instruction can jump to the *Execute* stage of its *L/S* phase from the the *Dispatch* stage of the *AGEN* phase if the effective address can be looked up from the BO-LSQ. Currently, two possible conditions are checked:

- if the offset and the content of the base register of the instruction match both the *offset* and *base* fields of any entry of the BO-LSQ, or
- if either the offset or the content of the base register of the instruction is equal to zero.

If one of these conditions is met, two pipeline stages can be bypassed and the execution of this instruction can be sped up by two clock cycles.

4. EXPERIMENTAL RESULTS

4.1 Setup

Experiments are conducted by executing the SPECint2000 benchmarks [19] on SimpleScalar v3.0d [2]. The simulated processor is a 21264-like processor [11], and the default parameters of the processor and memory hierarchy set by SimpleScalar are listed in Table 1. The SPECint2000 binaries with SPEC peak settings were downloaded from the University of Michigan through a link at the SimpleScalar home page [21]. Instead of just collecting a small trace of each program as done by other researchers, this paper chose to simulate the entire program of every SPECint2000 benchmark to avoid the pitfall that a program's locality behavior is not constant over the run of the entire program [9].

All the SPECint2000 benchmarks will be executed respectively on 1-way, 2-way, 4-way, 8-way, 16-way, and 32-way L1 instruction and data caches. Caches with up to 32-way associativity are simulated because modern embedded processors use data caches with high degrees of associativity in order to increase performance. In addition, both the Cache Decay and Drowsy Cache techniques have been implemented in the SimpleScalar as well, and their impact on static power consumption and execution times will be used as the baseline comparison. Table 2 lists the configurations used in this paper with different T_{sleep} and T_{off} settings to evaluate the performance of these three leakage saving techniques.

Processor Core	
Instruction Window	16 RUU, 8 LSQ entries
Issue Width	4 instructions/cycle
Functional Units	4 IntALU, 1 IntMult, 4 FpALU, 1 FpMult, 2 Memory Ports
Pipeline	5 Stages: Fetch/Dispatch/Execute/WriteBack/Commit
Memory Hierarchy	
L1 Dcache	32KB, 32B blocks, 1/2/4/8/16/32 ways, 1-cycle latency
L1 Icache	32KB, 32B blocks, 1/2/4/8/16/32 ways, 1-cycle latency
L2 cache	1MB, 64B blocks, 8-way LRU, 6-cycle latency
Memory	18-cycle latency

Table 1: Configuration of Simulated Processor

Name	Experiment Setup
Original	Set-Associative, 8-entry LSQ, $T_{sleep} = \infty$
MSA0116	MSA, 8-entry LSQ, T_{sleep} = 1K cycles, T_{off} = 16K Cycles
MSA0104	MSA, 8-entry LSQ, T_{sleep} = 1K cycles, T_{off} = 4K Cycles
SSA0116	MSA, 8-entry BO-LSQ, T_{sleep} = 1K cycles, T_{off} = 16K Cycles
Decay16	Cache Decay, 8-entry LSQ, T_{off} = 16K Cycles
Decay4	Cache Decay, 8-entry LSQ, T_{off} = 4K Cycles
Decay1	Cache Decay, 8-entry LSQ, T_{off} = 1K Cycles
Drowsy16	Drowsy Cache, 8-entry LSQ, T_{sleep} = 16K Cycles
Drowsy4	Drowsy Cache, 8-entry LSQ, T_{sleep} = 4K Cycles
Drowsy1	Drowsy Cache, 8-entry LSQ, T_{sleep} = 1K Cycles

Table 2: Setups of Experiments

4.2 Active Ratios

Since the MSA scheme of SSA activates an extra cache line only when a *repeated* miss has occurred, SSA should have fewer active blocks than Cache Decay and Drowsy Cache. Figure 6 shows on average only about 4.7% of cache blocks are active for SSA caches when the active window is set to $1K$ cycles, while the average active ratios for Cache Decay and Drowsy Cache with the same active window are 7.7% and 5.0% respectively.

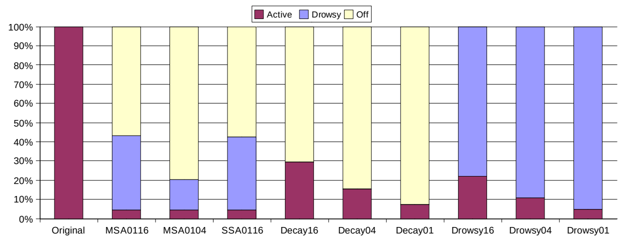


Figure 6: Average Active Ratios

4.3 Performance Evaluation

Static Energy

Figure 7 plots the normalized cache leakage dissipation for the SPECint2000 benchmarks. In this graph, the ratio of static energy of a drowsy block relative to an active block R_{sleep} is set to 0.08 [7], and the relative overhead due to an additional miss R_{L2} is 50 [10]. The bars in the graph are normalized to the static energy dissipated by the original 32B L1 data cache with 1-way, 2-way, 4-way, 8-way, 16-way, or 32-way organization.

SSA can significantly reduce the leakage power of L1 data cache, up to 90% when $T_{sleep} = 1Kc$ and $T_{off} = 16Kc$. When T_{off} is reduced to $4K$ cycles, the reduction can be further pushed to 93%, which is better than the best case of Cache Decay ($T_{off} = 1Kc$). Cache Decay usually saves more leakage energy than Drowsy Cache, since even drowsy cache blocks still produce leakage. When the time window is set to $1K$, $4K$, and $16K$ cycles, the normalized leakage powers of Drowsy Cache configurations are about 19%, 25%, and 39%, respectively. This figure reveals that even though a drowsy cache line dissipates only 8% of leakage of an active line, the overall effect is still substantial since most cache lines are drowsy.

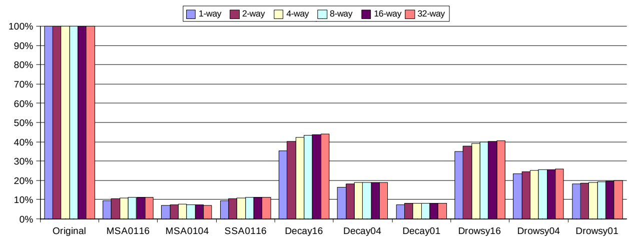


Figure 7: Average Normalized Leakage Power

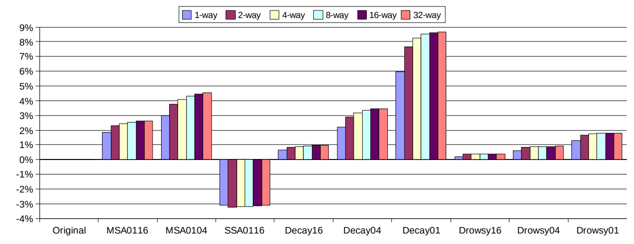


Figure 8: Average Run Time Impact

Run Time Impact

Turning off or putting to sleep cache lines to reduce leakage power will definitely incur performance penalties since reloading data or waking up cache lines takes time and energy. With $1Kc$ time window, Cache Decay causes the average execution times to increase over 6% and Drowsy Cache about 1.7%. Similarly, the MSA scheme of SSA alone degrades the performance by about 2.4% (the MSA0116 configuration). However, SSA deploys the BO-LSQ by modifying the default LSQ design to compensate this performance loss. Figure 8 shows the SSA cache with 8-entry BO-LSQ (i.e. SSA0116) can not only recover the 2.4% performance loss, but also improves the average run time by 3%.

Figure 7 and Figure 8 together demonstrate that SSA can outperform the original configuration by 3% while just dissipating 10% of leakage power. It provides an approach to improve both leakage power of data caches and performance. On the contrary, Cache Decay and Drowsy Cache reduce cache leakage power while compromising performance. Cache Decay can cut the cache leakage power down to 8%, but average execution time will rise over 8%. On the other hand, Drowsy Cache can contain performance penalties within 1%. However, the normalized leakage energy might reach 25%∼30%.

Miss Rates

Turning off cache lines definitely increases miss rates. However, the extra miss rates introduced by SSA are very small. Figure 9 shows the average miss rates of SPECint2000 benchmarks observed on the configurations listed in Table 2 with 1-way, 2-way, 4-way, 8-way, 16-way, and 32-way instruction and data caches. The average miss rates of the original configuration on 1-way, 2-way, 4-way, 8-way, 16-way, and 32-way caches are 6.4%, 5.3%, 5.1%, 5.1%, 5.0%, and 5.0%, respectively. SSA caches with $T_{sleep} = 1K$ cycles and $T_{off} = 16K$ cycles introduce an additional miss rate of about 0.5% to each case, and the average miss rates are 6.9%, 5.9%, 5.7%, 5.6%, 5.6%, and 5.6%. When T_{off} of SSA is reduced to $4K$ cycles, the extra miss rates grow to about 1.6%∼1.9%.

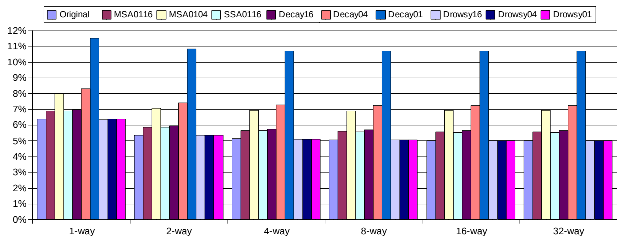


Figure 9: Average Miss Rates

Cache Decay with $T_{off} = 16K$ raises average miss rates about 0.7%, but the average miss rates are almost doubled when T_{off} is reduced to $1K$ cycles. On the other hand, Drowsy Cache does not increase miss rates since it does not turn off cache lines. Instead, it puts cache lines into a state-preserving drowsy state. However, drowsy cache lines still dissipate static power.

4.4 BO-LSQ Bypassing

The performance gain observed in the SSA0116 configuration comes from the bypassing of pipeline stages by BO-LSQ. The saving comes from two sources: (1) load or store instructions with zero offsets or bases, which account for 43%, and (2) load or store instructions with offsets or bases matched entries in BO-LSQ, which account for 5%. Consequently, the address computation phase of 48% of load or store instructions can be bypassed as shown in Figure 10.

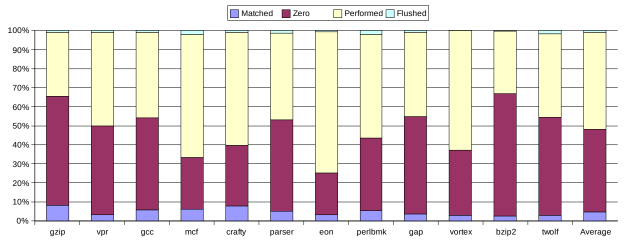


Figure 10: Percentage Breakdown of Load/Store Instructions

4.5 L2 Cache

SSA can also be used at L2 caches since the active ratios are usually very low. Figure 11 reveals that the normalized leakage power of the 1MB L2 SSA cache with $T_{sleep} = 1Kc$ and $T_{off} = 1024Kc$ is cut by 96%, which is significant since a 1M L2 cache would comprise much of the chip area and transistor counts of a CPU. Figure 12 demonstrates that L2 SSA caches can improve performance as well. Almost all the SPECint2000 benchmarks experience runtime reduction when SSA is applied to L2 cache, except for *twolf*. The reason is because the window size of $1024Kc$ is still too small for *twolf* and it causes many extra L2 misses. However, the average drop on execution times of SPECint2000 benchmarks still reaches 3% when the number of BO-LSQ entries is 8.

5. RELATED WORK

Selective cache ways proposed the first method of turning off unneeded ways to reduce cache energy [1]. It is a coarse-grain approach that can turn off entire cache ways of set-associative caches

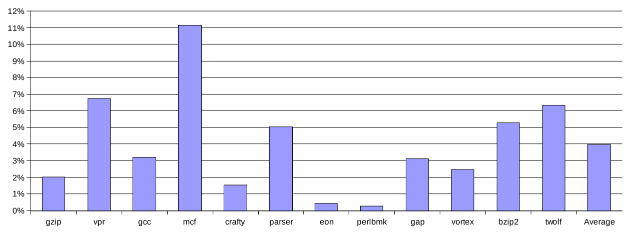


Figure 11: L2 SSA Cache Leakage Power

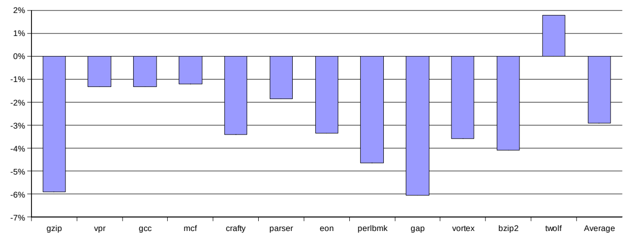


Figure 12: Average Run Time Impact (L2 SSA Cache)

for different programs. Recently, several fine-grain techniques have been developed that turn off or deactivate individual cache lines to reduce power consumption [7, 8, 10, 12, 23]. They all have developed polices and implementations to turn off or put to sleep cache lines that are not likely to be reused. However, these methods all incur performance penalties since reloading data or waking up cache lines takes time and energy. The MSA scheme of SSA caches follows the similar path to reduce leakage power dissipation and hence suffers minor performance degradation as well. Nevertheless, SSA caches deploy BO-LSQ, which can obtain enough performance gain that does not only recover the performance losses caused by MSA but still has an excess to reduce execution times.

Signature Buffer permits load and store instructions bypassing normal memory hierarchy for fast data communication [18]. However, it has to copy cache blocks into the signature buffer and must deal with data alignment and signature synonym problems. On the other hand, BO-LSQ in this paper skips the address computation phase by only comparing the base and offset of the dispatched instruction with those of instructions in BO-LSQ entries. Cached LSQ is another way to circumvent normal memory hierarchy that buffers data in the data field of LSQ entries [16]. This technique can be easily integrated in BO-LSQ to reduce accesses to cache blocks.

Recently Meng et. al. have developed a model to estimate the optimal leakage savings by combining Cache Decay and Drowsy Cache when the perfect knowledge of future trace is available [15]. The work is closely related to this paper, but the model does not assess the performance penalties when the optimal leakage savings are achieved. On the contrary, the SSA of this paper can improve performance while significantly cutting leakage dissipation of data caches.

6. CONCLUSIONS

This paper has presented a cache architecture, the SSA cache, that can reduce leakage power without incurring performance penalties. The SSA cache has implemented the MSA scheme, which only activates the minimal numbers of ways in every cache set, and it has cut static power consumption of SPECint2000 benchmarks by 90% on average. In addition, the performance losses caused by the MSA scheme has been compensated by an 8-entry BO-LSQ, and the average execution times of SPECint2000 benchmarks has been reduced by 3%. Furthermore, SSA can also trim the leakage power of L2 data cache by 96% on average, while still accomplishing a 3% reduction in execution times with an 8-entry BO-LSQ.

7. REFERENCES

[1] David H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–259, 1999.

[2] T. Austin, E. Larson, and D. Emst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[3] Lee Baugh and Craig Zille. Decomposing the load-store queue by function for power reduction and scalability. In *The First Watson Conference on Interaction between Architecture, Circuits, and Compilers (p = ac^2 Conference)*, 2004.

[4] Douglas C. Burger, James R. Goodman, and Alain Kägi. The declining effectiveness of dynamic caching for general-purpose microprocessors. Technical report 1261, University of Wisconsin-Madison Computer Science Department, 1995.

[5] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. Profileme: hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 292–302. IEEE Computer Society, 1997.

[6] B. Doyle, R. Arghavani, D. Barlage, S. Datta, M. Doczy, J. Kavalieros, A. Murthy, and R. Chau. Transistor elements for 30nm physical gate lengths and beyond. *Intel Technology Journal*, 6(2):42–54, 2002.

[7] Krisztian Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 148–157, 2002.

[8] H. Hanson, M.S. Hrishikesh, V. Agarwal, S.W. Keckler, and D. Burger. Static energy reduction techniques for microprocessor caches. In *Proceedings of 2001 International Conference on Computer Design*, pages 276–283, 2001.

[9] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantative Approach*. Morgan Kaufmann, 3rd edition, 2003.

[10] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage. In *Proceedings of the 28th annual International Symposium on Computer Architecture*, pages 240–251, 2001.

[11] R.E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[12] Nam Sung Kim, Krisztian Flautner, David Blaauw, and Trevor Mudge. Drowsy instruction caches: Leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture*, pages 219–230, November 2002.

[13] Nam Sung Kim, Krisztian Flautner, David Blaauw, and Trevor Mudge. Circuit and microarchitectural techniques for reducing cache leakage power. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(2):167–184, February 2004.

[14] Lin Li, Vijay Degalahal, N. Vijaykrishnan, Mahmut Kandemir, and Mary Jane Irwin. Soft error and energy consumption interactions: a data cache perspective. In *Proceedings of the 2004 international symposium on Low power electronics and design*, pages 132–137. ACM Press, 2004.

[15] Yan Meng, Timothy Sherwood, and Ryan Kastner. On the limits of leakage power reduction in caches. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, February 2005.

[16] D. Nicolaescu, A. Veidenbaum, and A. Nicolau. Reducing data cache energy consumption via cached load/store queue. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, pages 252–257, 2003.

[17] I. Park, C.L. Ooi, and T.N. Vijaykumar. Reducing design complexity of the load/store queue. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 411–422, 2003.

[18] Lu Peng, Jih-Kwon Peir, and Konrad Lai. Signature buffer: Bridging performance gap between registers and caches. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pages 164–175, 2004.

[19] Standard Performance Evaluation Corporation. SPEC CPU2000 v1.1, 2000.

[20] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice-Hall, 4th edition, 1999.

[21] Chris Weaver. SPEC 2000 binaries. http://www.eecs.umich.edu/∼chriswea/benchmarks/spec2000.html.

[22] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance analysis using the mips r10000 performance counters. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing*. ACM Press, 1996.

[23] Huiyang Zhou, Mark C. Toburen, Eric Rotenberg, and Thomas M. Conte. Adaptive mode control: A static-power-efficient cache design. *ACM Transactions on Embedded Computing Systems (TECS)*, 2(3):347–372, 2003.