Snug Set-Associative Caches: Reducing Leakage Power of Instruction and Data Caches with No Performance Penalties

YUAN-SHIN HWANG
National Taiwan Ocean University
and
JIA-JHE LI
National Tsing Hua University

As transistors keep shrinking and on-chip caches keep growing, static power dissipation resulting from 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 cuts most of static power dissipation of caches without incuring performance penalties. 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 caused by this scheme are compensated by the *base-offset load/store queues*. The rationale of combining these two techniques is locality: as the contents of the cache blocks in the current working set are repeatedly accessed, same addresses would be computed again and again. The SSA cache architecture can be applied to data and instruction caches to reduce leakage power without incurring performance penalties. Experimental results show that SSA can cut static power consumption of the L1 data cache by 93%, on average, for SPECint2000 benchmarks, while the execution times are reduced by 5%. Similarly, SSA can cut leakage dissipation of the L1 instruction cache by 92%, on average, and improve performance over 3%. Furthermore, when SSA is adopted for both L1 data and instruction caches, the normalized leakage of L1 data and instruction caches is lowered to 8%, on average, while still accomplishing a 2% reduction in execution times.

Categories and Subject Descriptors: B3.2 [**Memory Structures**]: Design Styles—*Cache Memories*

General Terms: Algorithms, Performance, Design, Experimentation

Additional Key Words and Phrases: Caches, cache decay, drowsy caches, leakage power

This research was supported in part by MOEA project 95-EC-17-A-01-S1-034 and NSC grants NSC94-2213-E-019-008 and NSC95-2221-E-019-012-MY3
Authors' addresses: Yuan-Shin Hwang, Department of Computer Science and Engineering, National Taiwan Ocean University, Keelung 20224, Taiwan; email: shin@cs.ntou.edu.tw.
Jia-Jhe Li, Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan; email: jjli@pllab.cs.nthu.edu.tw.

ACM Reference Format:
Hwang, Y.-S. and Li, J.-J. 2007. Snug set-associative caches: Reducing leakage power of instruction and data caches with no performance penalties. ACM Trans. Architec. Code Optim. 4, 1, Article 6 (March 2007), 28 pages. DOI = 10.1145/1216544.1216549 http://doi.acm.org/10.1145/1216544.1216549.

1. INTRODUCTION

As the minimum feature size gets smaller and more transistors are densely packed onto processors, static power dissipation resulting from leakage takes an increasing fraction of total power in processors. Since static power dissipation is a function of the number of transistors, the VLSI technology, and circuit design style [Butts and Sohi 2000], the 7.5 times increases in the total leakage current each generation result in the leakage power increasing about 5 times [Borkar 1999]. Consequently, leakage power will soon be the dominant form of power dissipation [Doyle et al. 2002; Kim et al. 2004a]. Furthermore, currently 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. For instance, it is projected that in a 70-nm CMOS process, leakage power will amount to more than 60% of power consumed in L1 caches, if left unchecked [Kim et al. 2004a].

Several techniques have already been proposed to reduce leakage power of caches [Albonesi 1999; Flautner et al. 2002; Hanson et al. 2001; Hu et al. 2003; Kaxiras et al. 2001; Kim et al. 2002; Zhou et al. 2003]. They all have developed polices and implementations to turn off or put to sleep 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 contents are discarded, and the activity in caches is only centered on a small subset of cache lines during a fixed period of time because of the property of locality. However, these methods all incur performance penalties, since reloading data to cache 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 preset number of cycles. However, this scheme does not activate an additional cache block right away once a cache miss occurs, 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 recently accessed, 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 number 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, redundant addresses computations are performed again and again. In other words, the load/store instruction currently being dispatched might find an entry in BO–LSQ with the same 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 SSA cache architecture can be applied to data and instruction caches, and this paper has exploited three possibilities: data caches only, instruction caches only, and both data and instruction caches. Experimental results show that SSA can cut static power consumption of the L1 data cache by 93%, on average, for SPECint2000 benchmarks, while the execution times are reduced by 5%. Similarly, SSA can cut leakage dissipation of the L1 instruction cache by 92%, on average, and improve performance over 3%. Furthermore, when SSA is adopted for both L1 data and instruction caches, the normalized leakage of L1 data and instruction caches is lowered to 8%, on average, while still accomplishing a 2% reduction in execution times.

Further improvement can be made if data of previous load operations are buffered in the unused data field of BO–LSQ entries as well, as done by cached LSQ [Nicolaescu et al. 2003]. In addition to bypassing pipeline stages of address computation, the *cached BO–LSQ* could load the data directly from matched entries and, hence, avoids accessing cache lines. As a result, it can further speedup execution as one more pipeline stage can be skipped. Experimental results demonstrate that an extra 1% of performance improvement can be obtained by the cached BO–LSQ. Meanwhile, average access rates of L1 data caches are reduced by 11%. The implication is that the dynamic power consumption of L1 data cache will be lowered as well, since dynamic power dissipation of caches is proportional to the access rates [Zhang et al. 2003].

The main results of this paper are as follows:

- The MSA scheme of SSA can significantly cut static power consumption of L1 data and L1 instruction caches by more than 92% for SPECint2000 benchmarks. The average active ratios of cache blocks are less than 4%.
- BO–LSQ can recover the performance losses caused by the MSA scheme. In fact, performance gains of 2 to 5% have been observed when the default 32-entry LSQ is modified to the BO–LSQ design.
- Cached BO–LSQ can further extend the performance improvement by about 1%. Meanwhile, average access rates of L1 data caches are reduced by 11%. As a result, the dynamic power consumption of L1 data cache will be lowered as well, since dynamic power dissipation of caches is proportional to the access rates [Zhang et al. 2003].

The rest of this paper is organized as follows. Section 2 introduces the minimum set associative scheme and Section 3 presents the modifications made to implement the base-offset load/store queues. Experimental results will be presented in Section 4, and Section 5 explores the possibilities and assesses the potential of applying some new leakage-reduction techniques to SSA. Section 6 surveys the related work and Section 7 concludes this paper.

2. MINIMUM SET-ASSOCIATIVE SCHEME

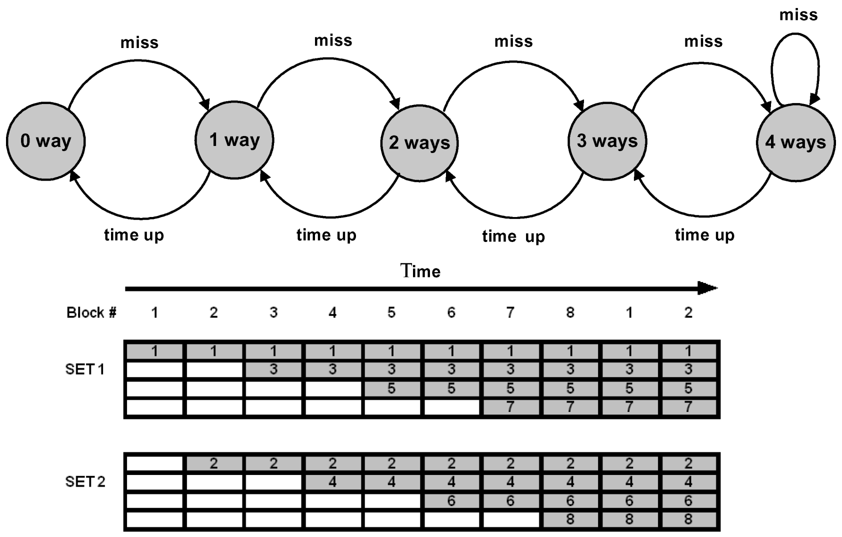
Locality is a property that is shared by most programs: programs tend to reuse data and instructions they have used recently [Hennessy and Patterson 2003]. Because of locality and higher speed of cache memories, caches can bridge the processors-memory performance gap and, hence, substantially improve performance of processors. On the other hand, caches are very power inefficient because of 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 [Flautner et al. 2002]. In addition, a cache block is not likely to be referenced in the future when it leaves the active window [Burger et al. 1995; Kaxiras et al. 2001]. Therefore, it is reasonable to deactivate cache blocks when they hold data not likely to reused.

2.1 Policy

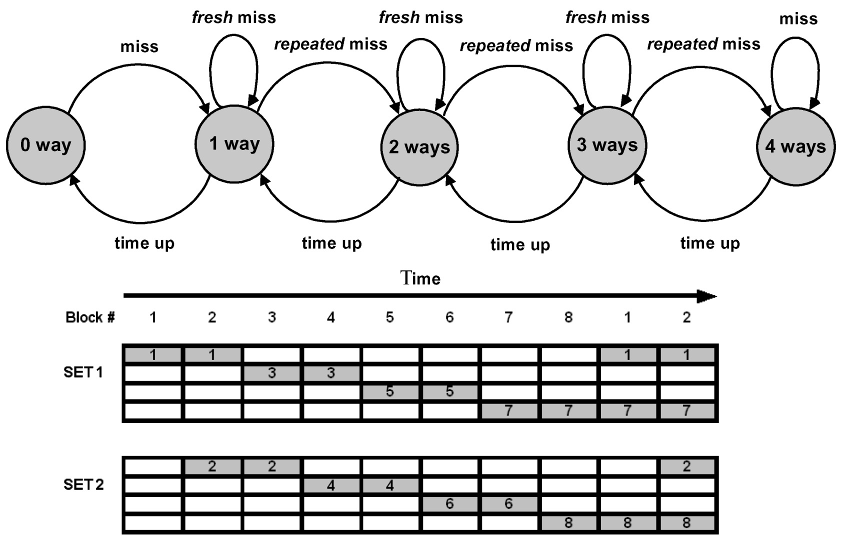
The SSA cache reduces leakage power of caches by deploying the minimum set-associative (MSA) scheme, which tries to activate as few cache blocks as possible. The MSA scheme differs from the existing leakage-reducing techniques in the timing of when to activate an extra cache block in the target cache set. Both cache decay and drowsy cache allocate a new block when a miss occurs and deactivate a block when its time window expires [Flautner et al. 2002; Kaxiras et al. 2001]. As a result, the numbers of active ways in a four-way set-associative cache will follow the state diagram shown in Fig. 1a. 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 [Burger et al. 1995]. In other words, cache blocks that are not actively accessed are not likely to be used in the future.

Since set-associative caches are introduced to solve 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 [Tanenbaum 1999]. In other words, MSA increases the number of active ways of a cache block only if the data of current miss has been accessed before within the time window. In order to determine whether the thrashing problem might occur, 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



(a) Set-Associative



(b) Minimum Set-Associative

Fig. 1. Set-associative versus minimum set-associative (four-way).

- *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 four-way MSA cache will follow the state diagram shown in Fig. 1b.

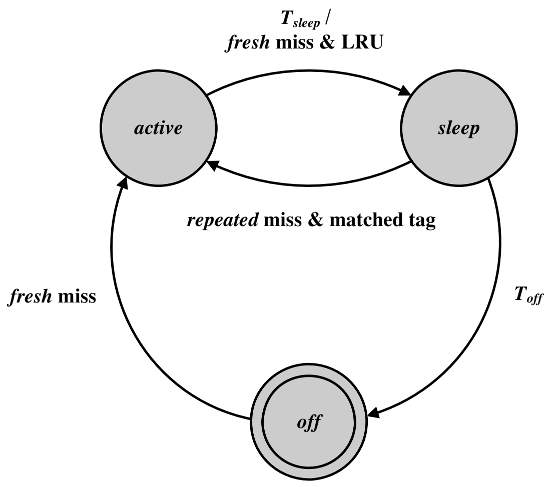


Fig. 2. States of SSA cache blocks.

Consider a four-way set-associative cache with only two 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 Fig. 1a. 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 Fig. 1b. 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 rereferenced, 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* 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 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* to the *active* state. An *active* block will be put to sleep when one of two conditions occurs: (1) when a *fresh* miss happens

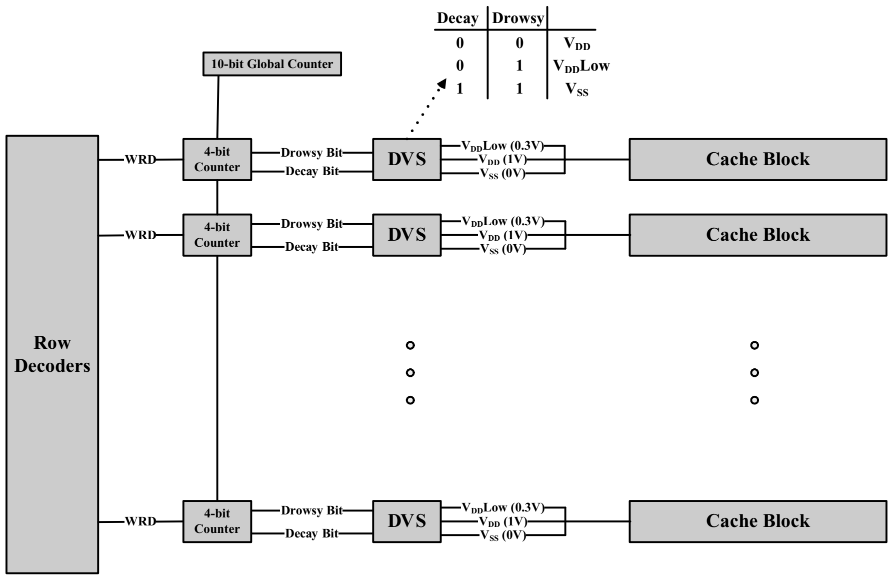


Fig. 3. SSA organization.

and this block is the LRU block in the set, or (2) when the preset time slot T_{sleep} expires. Finally, a block in the *sleep* state will be turned off when it is idle over a preset time window T_{off}.

These three states can be implemented by combining the techniques in cache decay [Kaxiras et al. 2001] and drowsy cache [Flautner et al. 2002], 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 one-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 [Li et al. 2004]. 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 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 [Dean et al. 1997; Zagha et al. 1996]. 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 the leakage power of active blocks, the 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 on 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 on 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 field 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 dynamic energy overhead $E_{L2access}$:

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

where R_{L2} is, in fact, the ratio $L2Access{:}leak$ (range: 5 to 100) used in cache decay, which relates the dynamic energy because of an additional miss (or write-back) to static leakage energy of a single clock cycle in the L1 cache [Kaxiras et al. 2001]. This ratio models the amount of dynamic power dissipated in the L2 cache and beyond because of an L1 cache miss. Multiplying this ratio to E_{block} and the number of additional L1 misses $ExtraMisses$ gives the dynamic energy overhead induced by extra L1 misses.

Performance of SSA caches will be evaluated by normalized leakage power *Leakage*, which uses the static power consumption of the original cache organization E_{origin} 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.

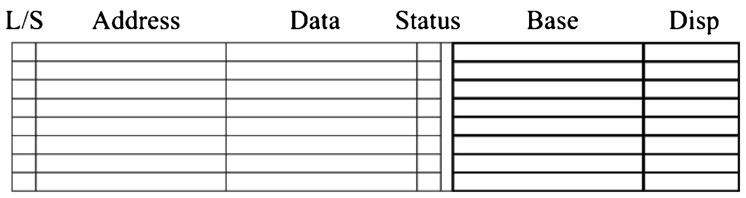


Fig. 4. Base-offset load/store queue.

3. BASE-OFFSET LOAD/STORE QUEUE

3.1 BO–LSQ

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 [Baugh and Zille 2004; Park et al. 2003]. Recently, a couple of techniques have been proposed to reduce power consumption or improve cache load time by buffering data in the LSQ [Nicolaescu et al. 2003; Peng et al. 2004]. This section presents a base-offset load/store queue (BO–LSQ) design, 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 exist in many programs [Peng et al. 2004]. The dependences can be correlated directly if the same base and offset are identified in any entry of the BO–LSQ; then a couple of pipeline stages of load instructions can be bypassed to achieve speedup.

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 [Nicolaescu et al. 2003].

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 [Kessler 1999]:

$$\begin{array}{ll} \text{load} & Ra, disp\,(Rb) \\ \text{store} & Ra, disp\,(Rb) \end{array}$$

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 [Austin et al. 2002]. 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

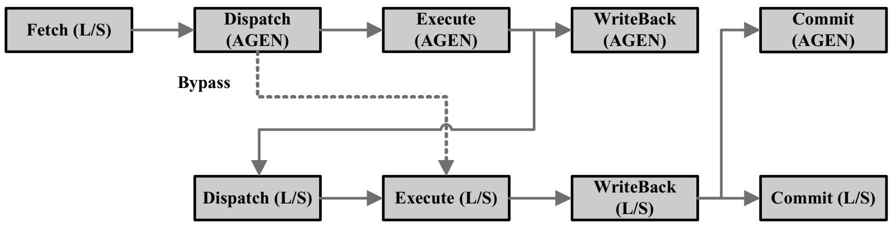


Fig. 5. Pipeline stages and bypassing.

the computation when operands are ready and the result will be broadcast in the *WriteBack* stage.

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 on 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 Fig. 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.

3.2 Cached BO–LSQ

The source of performance gain seen by SSA comes from the bypassing path of load/store instructions shown in Fig. 5. It avoids unnecessary address computations when they are available in the *address* field of BO–LSQ entries. Further runtime saving can be observed if the data of load instructions are buffered in the unused *data* field of BO–LSQ entries as well, i.e., the same technique as the *cached LSQ* proposed by Nicolaescu et al. [2003]. When the offset and the content of the base register of a load instruction match both the *offset* and *base* fields of any entry of the BO–LSQ, the value can be looked up directly from the *data* field. As a result, the instruction can jump directly to the *WriteBack* stage of its *L* / *S* phase and broadcast the loaded value. Figure 6 depicts this bypassing path from the the *Dispatch* stage of the *AGEN* phase to the *WriteBack* stage of its *L* / *S* phase, which would translate to an extra saving of one pipeline stage.

Table I. Configuration of Simulated Processor

Processor core	
Instruction window	64 RUU, 32 LSQ
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 D-cache	64KB, 64B blocks, 1/2/4/8/16/32 ways, one-cycle latency
L1 I-cache	64KB, 64B blocks, 1/2/4/8/16/32 ways, one-cycle latency
L2 cache	1MB, 64B blocks, 8-way LRU, 6-cycle latency
Memory	100-cycle latency

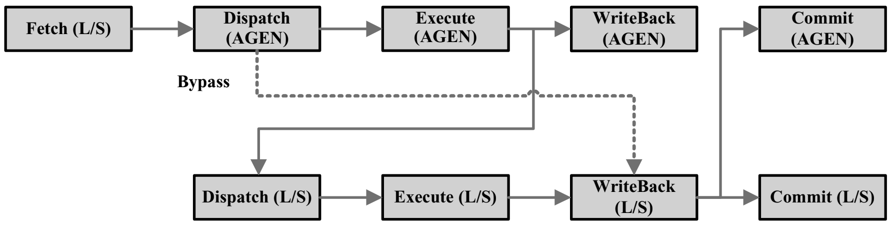


Fig. 6. Bypassing of cached BO–LSQ.

4. EXPERIMENTAL RESULTS

4.1 Setup

Experiments are conducted by executing the SPECint2000 benchmarks [Standard Performance Evaluation Corporation 2000] on SimpleScalar v3.0d [Austin et al. 2002]. The simulated processor is an Alpha 21264 [Kessler 1999] and the parameters of the processor and memory hierarchy set by SimpleScalar are listed in Table I. The values of these parameters are set according to the Alpha 21264 specification and, hence, are different from the default values set by the SimpleScalar distribution. The SPECint2000 binaries with SPEC peak settings were downloaded from the University of Michigan through a link at the SimpleScalar home page [Weaver]. Table II lists the numbers of instructions executed by the *sim-outorder* simulator using the TRAIN data set, which is smaller than the REF set and larger than the TEST set. 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 [Hennessy and Patterson 2003].

All the SPECint2000 benchmarks will be executed respectively on 1-, 2-, 4-, 8-, 16-, 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. For example, Transmeta Crusoe [Transmeta Corporation 2004] has 8-way set-associative

Table II. Instruction Counts (TRAIN Data Set)

Program	Instructions (in billions)
gzip	58.97
vpr	10.71
gcc	5.12
mcf	9.17
crafty	27.22
parser	13.43
eon	2.64
perlbmk	27.65
gap	9.52
vortex	18.81
bzip	61.13
twolf	13.20

Table III. Setups of Experiments

Name	Experiment setup
Original	Set-associative, LSQ, $T_{sleep}=\infty$
MSA	Minimum set-associative, LSQ, $T_{sleep} = 1$ K cycles, $T_{off} = 16$K cycles
SSA	Minimum set-associative, BO–LSQ, $T_{sleep} = 1$ K cycles, $T_{off} = 16$K cycles
Decay01	Cache decay [Kaxiras et al. 2001], LSQ, $T_{off} = 1$ K cycles
Decay04	Cache decay, LSQ, $T_{off} = 4$ K cycles
Decay16	Cache decay, LSQ, $T_{off} = 16$ K cycles
Drowsy01	Drowsy cache [Flautner et al. 2002], LSQ, $T_{sleep} = 1$ K cycles
Drowsy04	Drowsy cache, LSQ, $T_{sleep} = 4$ K cycles
Drowsy16	Drowsy cache, LSQ, $T_{sleep} = 16$ K cycles

caches and Intel XScale [Intel Corporation 2001] has 32-way set-associative caches. 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 III 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.

Since the SSA cache architecture can be deployed onto data and instruction caches, this section will present performance evaluations of the following three configurations: data caches only (Section 4.2), instruction caches only (Section 4.3), and both data and instruction caches (Section 4.4).

4.2 Data Caches

4.2.1 *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 7 shows on average only about 3.85% of cache blocks are active for SSA caches when the active window is set to 1 K cycles, while the average active ratios for cache decay and drowsy cache with the same active window are 5.99 and 3.88%, respectively.

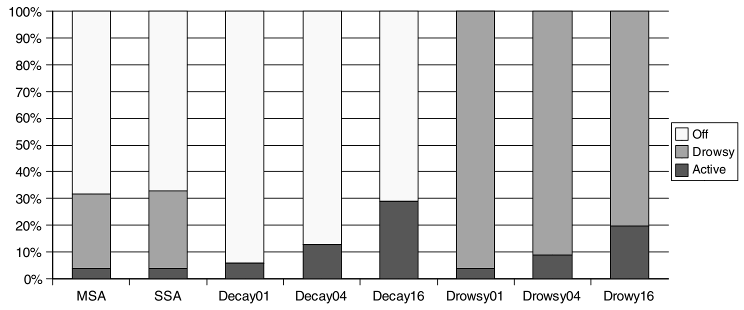


Fig. 7. Average active ratios.

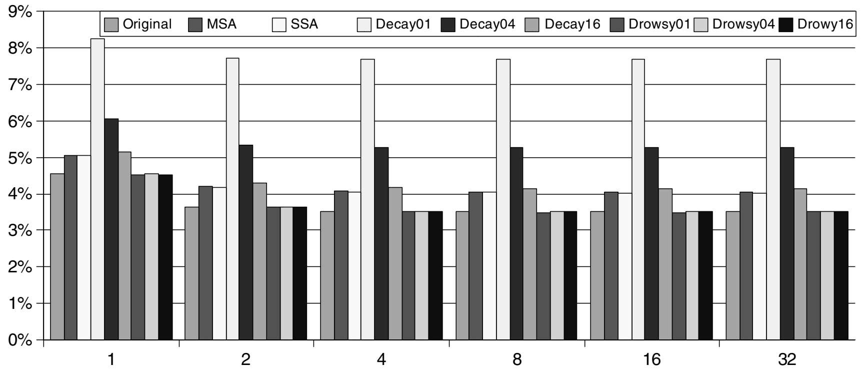


Fig. 8. Average miss rates.

4.2.2 *Performance Evaluation*

4.2.2.1 Miss rates. Turning off cache lines definitely increases miss rates. However, the extra miss rates introduced by SSA are very small. Figure 8 shows the average miss rates of SPECint2000 benchmarks observed on the configurations listed in Table III with 1-, 2-, 4-, 8-, 16-, and 32-way caches. The average miss rates of the original configuration on 1-, 2-, 4-, 8-, 16-, and 32-way caches are 4.6, 3.7, 3.5, 3.5, 3.5, and 3.5%, respectively. SSA caches with $T_{sleep} = 1\,\text{K}$ cycles and $T_{off} = 16\,\text{K}$ cycles introduce an additional miss rate of about 0.5% to each case, and the average miss rates are 5.1, 4.2, 4.0, 4.0, 4.0, and 4.0%.

Cache decay with $T_{off} = 16\,\text{K}$ raises average miss rates about 0.6%, but the average miss rates are almost doubled when T_{off} is reduced to 1 K 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.

4.2.2.2 Static energy. Figure 9 plots the normalized cache leakage energy 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 and the relative overhead resulting from an additional miss R_{L2} is 50. The bars in the graph are

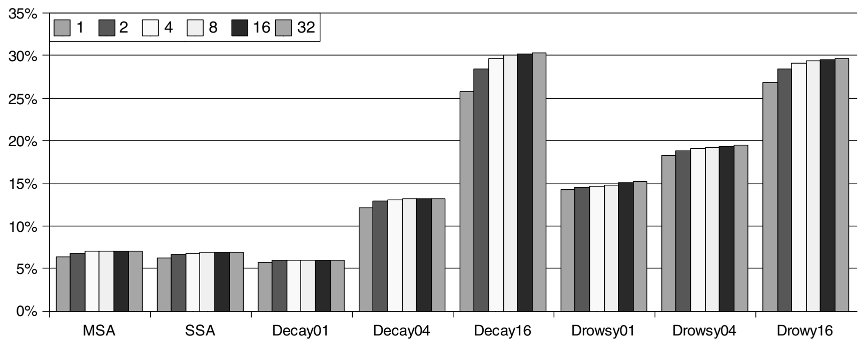


Fig. 9. Average normalized leakage power.

normalized to the static energy dissipated by the original L1 data cache with 1-, 2-, 4-, 8-, 16-, or 32-way organization.

SSA can significantly reduce the leakage power of L1 data cache, up to 93% when $T_{sleep} = 1$ Kc and $T_{off} = 16$ Kc. 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 1, 4, and 16 K cycles, the normalized leakage powers of drowsy cache configurations are about 15, 19, and 29%, 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.

4.2.2.3 Runtime 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 1 Kc time window, cache decay causes the average execution times to increase by 3% and drowsy cache about 1%, as shown in Fig. 10. Similarly, the MSA scheme of SSA alone degrades the performance by about 1% (the MSA configuration). With 1% of performance degradation, MSA reduces the leakage power by 93%, whereas cache decay ($T_{off} = 4$ K cycles) and drowsy cache ($T_{sleep} = 1$ K cycles) cut the static power dissipation by roughly 87 and 85%, respectively.

SSA deploys the BO–LSQ by modifying the default LSQ design to compensate the performance loss of the MSA scheme. Figure 10 shows that the SSA cache with the BO–LSQ design (i.e., the SSA configuration) can not only recover the 1% performance loss, but also improves the average runtime by 5%.

Figures 9 and 10 together demonstrate that SSA can outperform the original configuration by 5% while just dissipating 7% 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 6%, but average execution times will rise about 3%. On the other hand, drowsy cache can contain performance penalties within 1%. However, the normalized leakage energy might reach 15$\sim$29%.

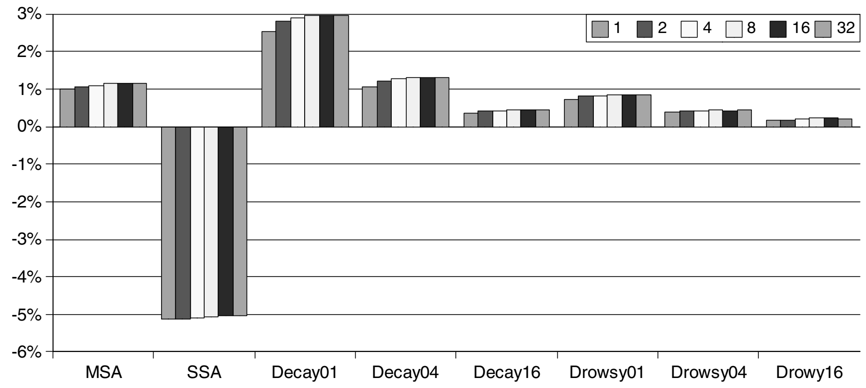


Fig. 10. Average runtime impact.

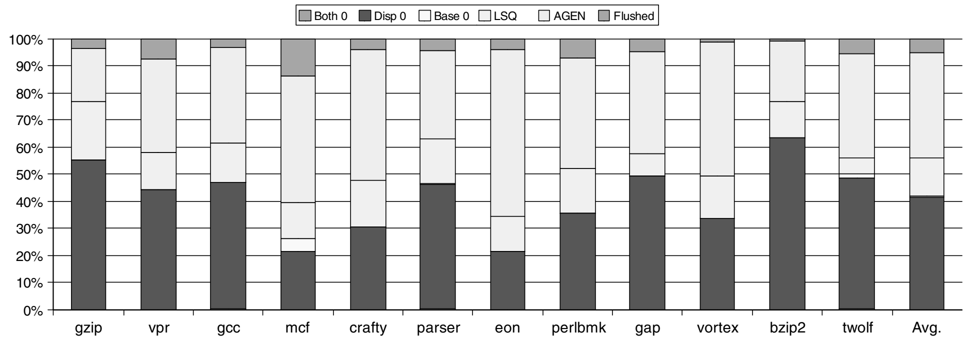


Fig. 11. Percentage breakdown of load/store instructions.

4.2.3 *BO–LSQ Bypassing*. The performance gain observed in the SSA configuration comes from the bypassing of pipeline stages by BO–LSQ. The saving comes from two sources:

- Load or store instructions with zero offsets or bases, which account for 42%.
- Load or store instructions with offsets or bases matched entries in BO–LSQ, which account for 14% in a 32-entry BO–LSQ.

Consequently, the address computation phase of 56% of load or store instructions can be bypassed in a 32-entry BO–LSQ, as shown in Fig. 11.

Figure 11 indicates that 39% of load and store instructions do not match any entries in the BO–LSQ and the address computation phase can not be bypassed. The rest (5%) is flushed by branch prediction before addresses are computed.

4.3 Instruction Caches

This section presents the efficacy of applying SSA to instruction caches only. The time window T_{sleep} for instruction caches is set to 2 K cycles, while the interval T_{off} remains 16 K cycles.

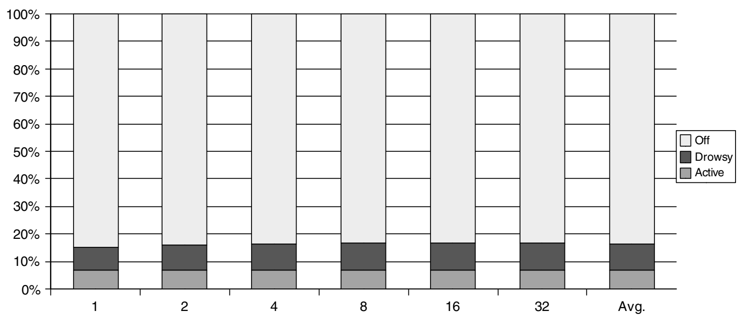


Fig. 12. Average active ratios (L1 instruction caches).

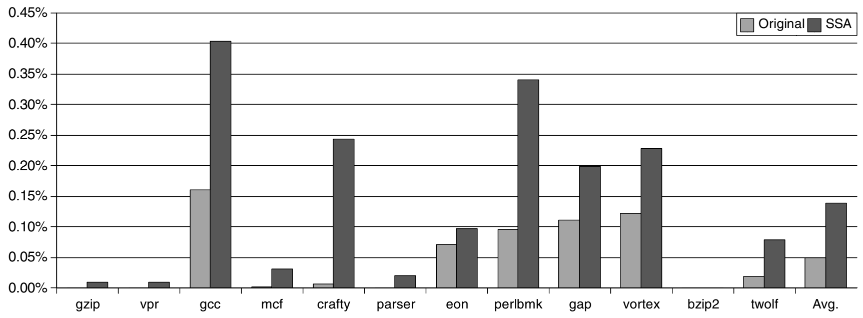


Fig. 13. Average miss rates (L1 instruction caches).

4.3.1 *Active Ratios.* Figure 12 shows the average active ratios of 64 KB 1-, 2-, 4-, 8-, 18-, and 32-way instruction caches. It depicts that the average active ratios of L1 instruction SSA caches are only 6.9% for SPECint2000 benchmark programs, which are about one half of the numbers observed by "drowsy instruction caches" [Kim et al. 2002].

4.3.2 *Performance Evaluation*

4.3.2.1 Miss rates. Figure 13 compares the average miss rates of L1 instruction SSA caches for SPECint2000 benchmarks with the original L1 instruction caches. The average miss rates of the original L1 instruction caches are very small—about 0.05%. SSA caches with $T_{sleep} = 2$ K cycles and $T_{off} = 16$ K cycles introduce an additional miss rates of about 0.09%. Consequently, the average miss rates are 0.14%.

4.3.2.2 Static energy. Figure 14 presents the normalized cache leakage of 64-KB 1-, 2-, 4-, 8-, 18-, and 32-way instruction caches for the SPECint2000 benchmarks. It shows SSA can reduce the leakage power of L1 instruction caches by more than 98% for some benchmark programs with high degrees of locality. On average, over 92% of leakage dissipation can be cut by SSA for the SPECint2000 benchmarks.

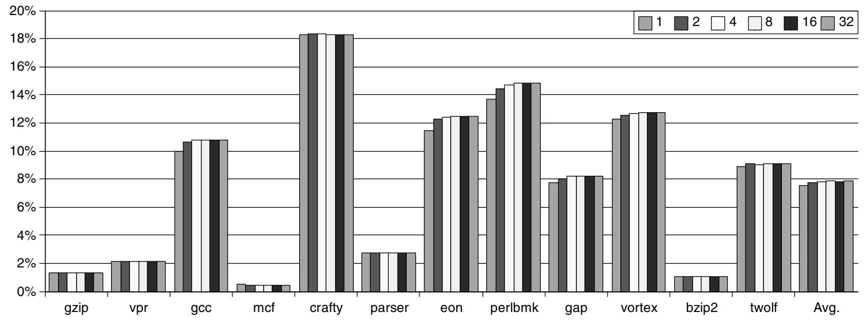


Fig. 14. Average normalized leakage power (L1 instruction caches).

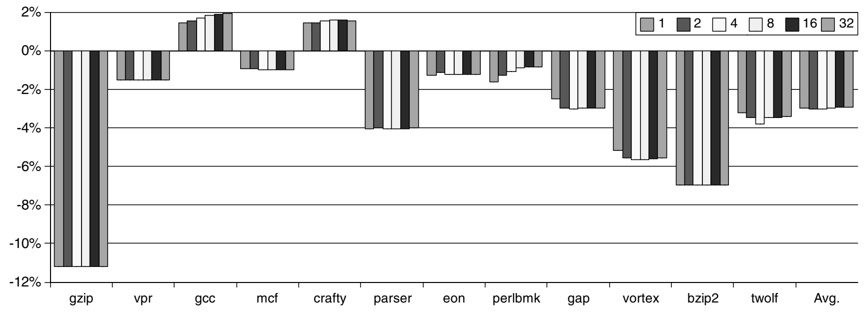


Fig. 15. Average runtime impact (L1 instruction caches).

4.3.2.3 Runtime impact. Figure 15 displays the runtime impact of L1 instruction SSA cache with 32-entry BO–LSQ, comparing to L1 instruction caches with the default 32-entry LSQ. It demonstrates that SSA can recover the performance losses caused by deactivating unused cache blocks for most of the SPECint2000 benchmarks. In fact, an average performance gain of 3% can be achieved by L1 instruction SSA cache.

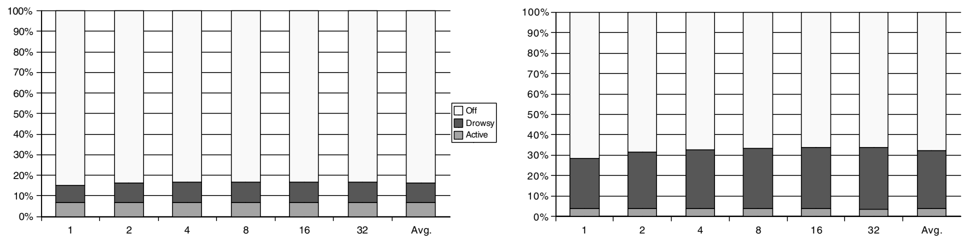
4.4 Instruction and Data Caches

This section presents the performance evaluations of simultaneously applying SSA to L1 instruction and L1 data caches. The time window T_{sleep} is set to 2 K cycles for instruction caches and 1 K cycles for data caches, while the interval T_{off} remains 16 K cycles.

4.4.1 *Active Ratios.* Figure 16 shows the average active ratios of 64 KB L1 instruction SSA caches and 64 KB L1 data SSA caches. It depicts that the average active ratios of L1 instruction SSA caches are only 6.8% for SPECint2000 benchmark programs, while the average active ratios of L1 data SSA caches are only 3.8%.

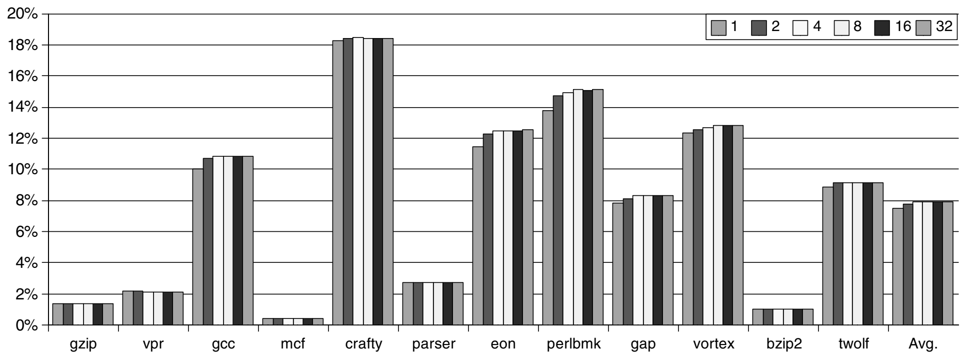
4.4.2 *Performance Evaluation*

4.4.2.1 Static energy. Figure 17 presents the normalized cache leakage of 64-KB L1 instruction SSA caches and 64-KB L1 data SSA caches for the

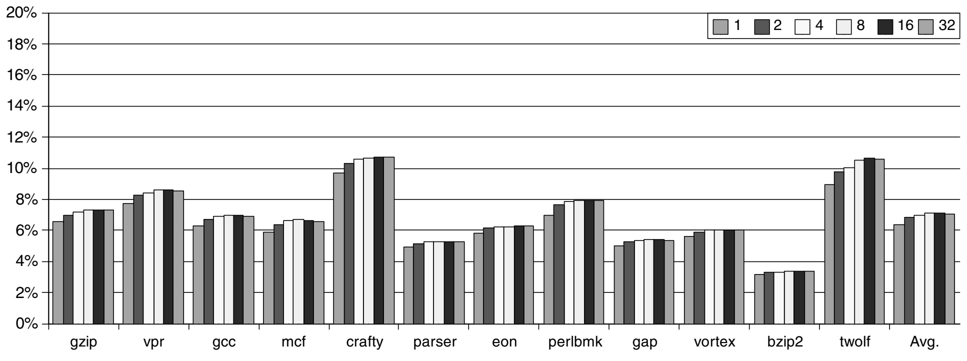


(a) Instruction Caches (b) Data Caches

Fig. 16. Average active ratios (L1 instruction and data caches).



(a) Instruction Caches



(b) Data Caches

Fig. 17. Average normalized leakage power (L1 instruction and data caches).

SPECint2000 benchmarks. While both instruction and data caches cut the leakage dissipation down to 7–8%, greater variations are observed for instruction caches, as shown in Fig. 17a.

4.4.2.2 Runtime impact. Figure 18 displays the runtime impact of L1 instruction and data SSA caches with 32-entry BO–LSQ, comparing to L1 instruction and data caches with the default 32-entry LSQ. It demonstrates that SSA can recover the performance losses caused by deactivating unused cache

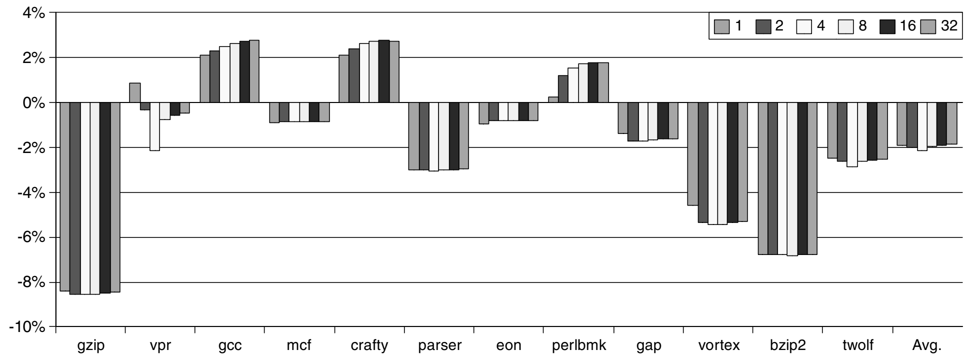


Fig. 18. Average runtime impact (L1 instruction and data caches).

blocks for most of the SPECint2000 benchmarks, except for *gcc*, *crafty*, and *perlbmk*. In fact, an average performance gain of 2% can be achieved by L1 instruction and data SSA caches.

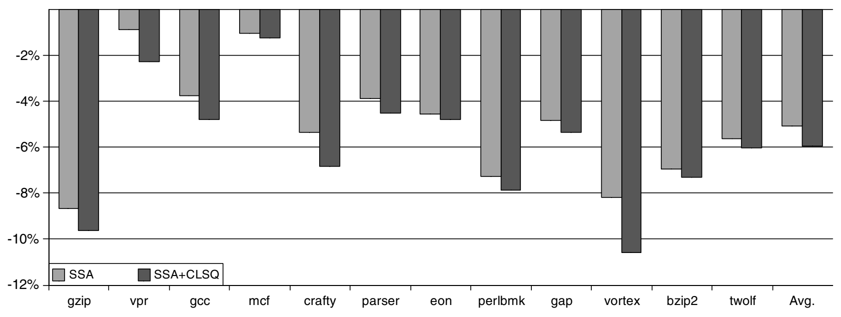
4.5 Cached BO–LSQ

Figure 19 compares the effects of cached BO–LSQ on execution times of SPECint2000 benchmarks for different SSA configurations. When the cached BO–LSQ is deployed on L1 data SSA caches, the average performance gain is extended from 5 to 6%, as shown in Fig. 19a. Similarly, runtime reduction of L1 instruction SSA caches is increased from 3 to 4% when cached BO–LSQ is applied, as depicted in Fig. 19b. Finally, Fig. 19c reveals that cached BO–LSQ can further improve performance by about 1% even when SSA is simultaneously applied to L1 instruction and L1 data caches.

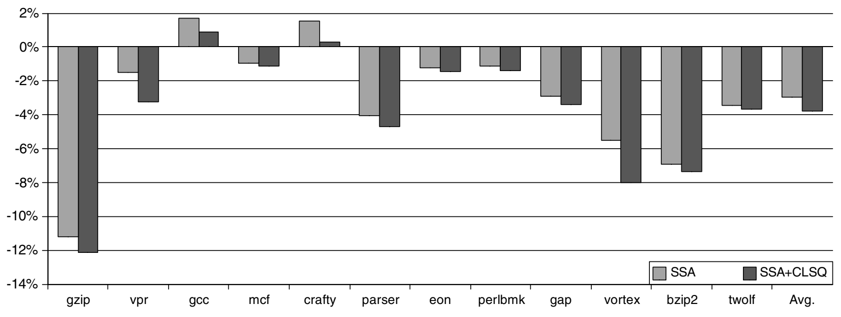
An additional advantage of the cached BO–LSQ is that it reduces dynamic power consumption as well, since it lowers the accesses to cache lines. When the value of a load instruction can be looked up directly from the buffer in the cached BO–LSQ, the access to the target cache line can be avoided. Figure 20 indicates the average access rate of L1 data caches is trimmed by 11%. As a result, the dynamic power of L1 data cache will be cut since dynamic power dissipation of caches is proportional to the access rates [Zhang et al. 2003].

5. DISCUSSION

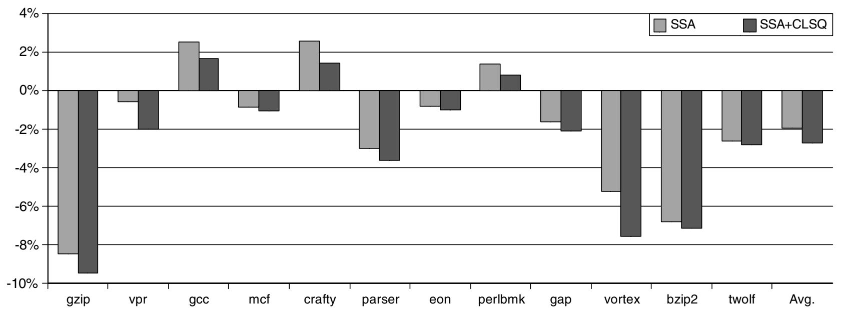
This paper uses cache decay and drowsy cache to implement SSA and, hence, the efficacy of these two techniques is used as the baseline comparison of performance evaluation presented in the previous section. However, the implementation of SSA is not limited to these two techniques. Any leakage reduction techniques that can work together to model the state transitions of SSA can be employed. Similarly, the methods proposed by this paper can be incorporated into other techniques as well. This section will explore some possibilities and assess the potential.



(a) Data Caches Only



(b) Instruction Caches Only



(c) Data and Instruction Caches

Fig. 19. Runtime impact (cached BO–LSQ).

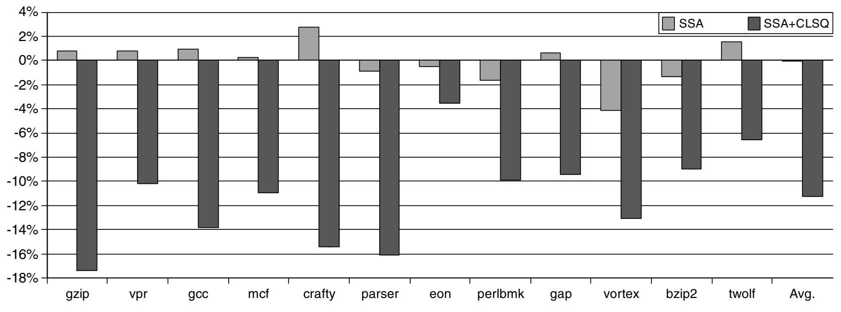
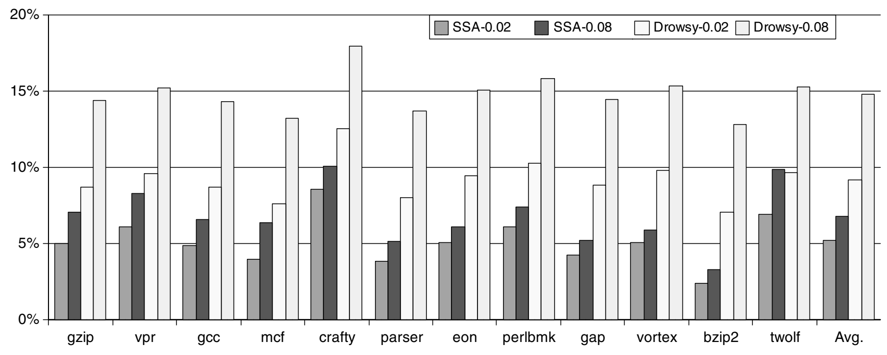
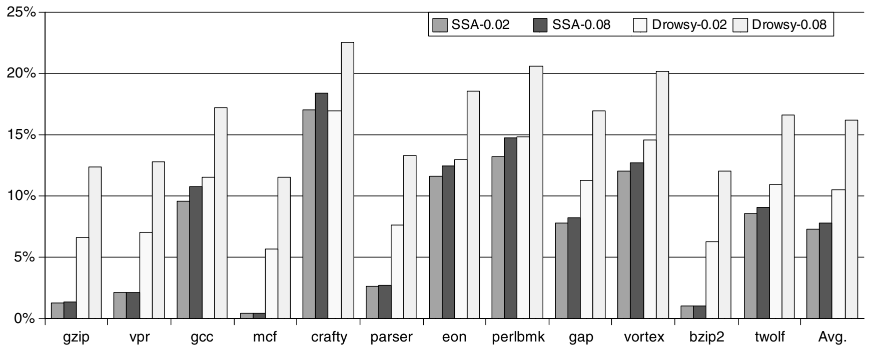


Fig. 20. Access rates of L1 data caches (cached BO–LSQ).



(a) Data Caches



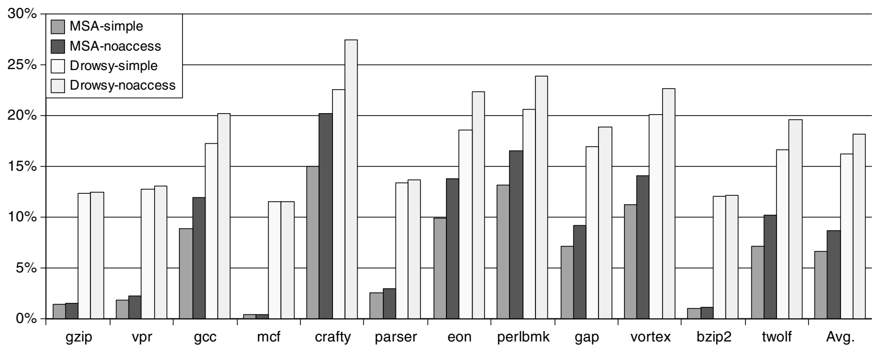
(b) Instruction Caches

Fig. 21. Normalized leakage of super drowsy caches.

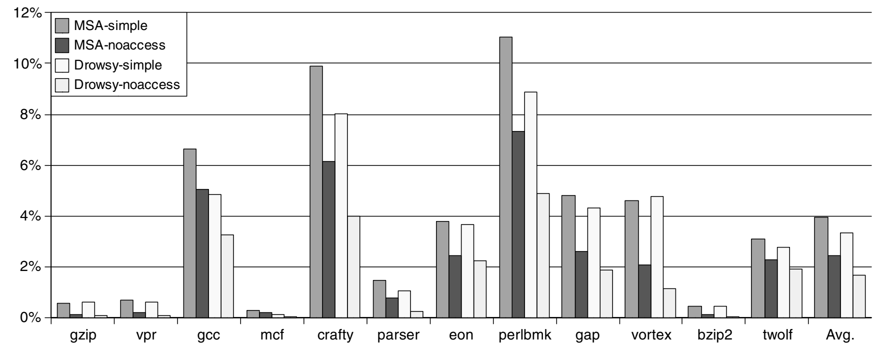
5.1 Super-Drowsy Caches

Super-drowsy cache circuit is a new state-preserving technique that can significantly reduce leakage power [Kim et al. 2004b]. When V_{DD} is lowered to 250 mV, the cell leakage power can be reduced by 98%. That is, R_{sleep} is equal to 0.02 for the super-drowsy caches, whereas R_{sleep} of the original drowsy caches is 0.08 [Flautner et al. 2002]. This circuit technique can be adapted by SSA to further reduce leakage dissipation of caches during the *sleep* state. Figure 21a shows that the normalized leakage energy of SSA data caches can be reduced from 7 to 5% by the super-drowsy circuit, and the normalized leakage of drowsy data caches can be cut from 15 to 9% through the same technique. Similarly, Fig. 21b shows that the super-drowsy circuit lowers the normalized leakage energy of SSA instruction caches from 8 to 7%, and decreases the normalized leakage of drowsy instruction caches from 16 to 11%.

The same work also compares the effects of the *noaccess* policy (only lines that have not been accessed during a fixed period are put into drowsy mode) and the *simple* policy (all lines are put into drowsy mode periodically) [Kim et al. 2004b]. These two policies can be easily implemented on SSA and other low-leakage cache architectures. Figure 22 reveals a similar outcome that has been observed, that is, deploying the *noaccess* policy on SSA



(a) Normalized Leakage Energy



(b) Runtime Impact

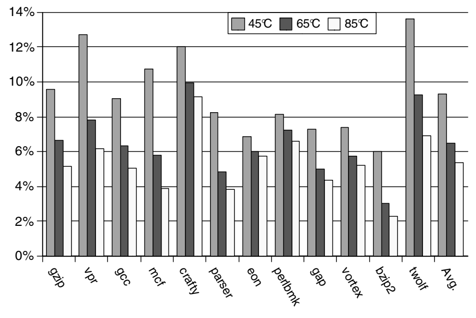
Fig. 22. Noaccess policy versus simple policy (instruction caches).

and drowsy caches results in better performance, but slightly worse leakage reduction.

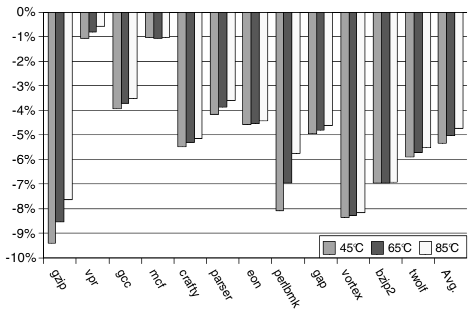
5.2 Temperature-Aware Leakage Reduction

Kaxiras et al. [2005] have developed a temperature-aware leakage reduction technique that can adapt the decay mode to temperature. They proposed a triggering mechanism based on the principle of decaying 4T thermal sensors. This triggering mechanism can be adopted by SSA as well. Figure 23 depicts the runtime impact and normalized leakage of temperature-aware SSA at 45°, 65°, and 85°C, whose decay intervals T_{off} are about 53 K, 13 K, and 6 K cycles, respectively. The average normalized leakage of SSA for SPECint2000 benchmarks at 45°, 65°, and 85°C is reduced to 9.3, 6.5, and 5.4% and the average runtime differences are −5.3, −5.0, and −4.7%, respectively.

Although the implementation of SSA is not limited to combining cache decay and drowsy cache, performance evaluation, presented in Section 4, was based on these two techniques. As a result, MSA is very similar to the hybrid *drowsy + decay* scheme deployed by them and even other researchers [Hu et al. 2003; Kaxiras et al. 2005; Meng et al. 2005]. The difference is that MSA does not activate an additional cache block right away once a cache miss occurs, as done

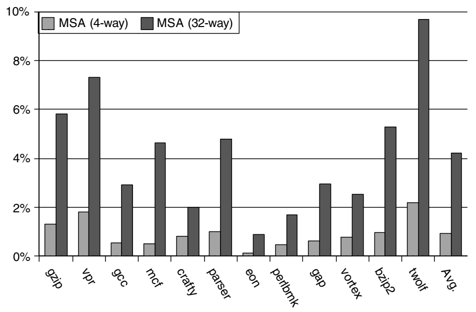


(a) Normalized Energy

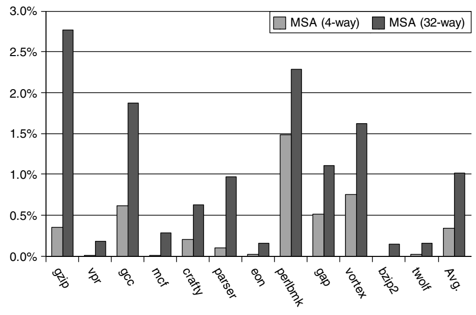


(b) Runtime Impact

Fig. 23. Temperature-aware SSA based on the delay4T sensors.



(a) Data Caches



(b) Instruction Caches

Fig. 24. Drowsy rates over the hybrid method.

by the hybrid technique. 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. Figure 24 demonstrates that MSA can put more cache lines into the drowsy mode than the hybrid scheme, especially for caches with high associativity.

5.3 Comparing with Optimal Leakage Saving

Recently Meng et al. [2005] have developed a model to estimate the optimal leakage savings by combining cache decay and drowsy cache. When oracle knowledge of future accesses is available, they estimate the upper limits of average leakage power saving that can be achieved on the six selected benchmarks are 96% for instruction caches and 99% for data caches with no performance losses. On the other hand, without the aide of oracle knowledge of future accesses, SSA instruction caches can accomplish 92% of leakage power saving for the SPECint2000 benchmarks with 3% performance improvement. Similarly, SSA data caches can reach 93% of leakage power saving for the SPECint2000 benchmarks with 5% performance improvement. Specifically, SSA can closely approach the optimal (within 4% for instruction caches and within 6% for data caches).

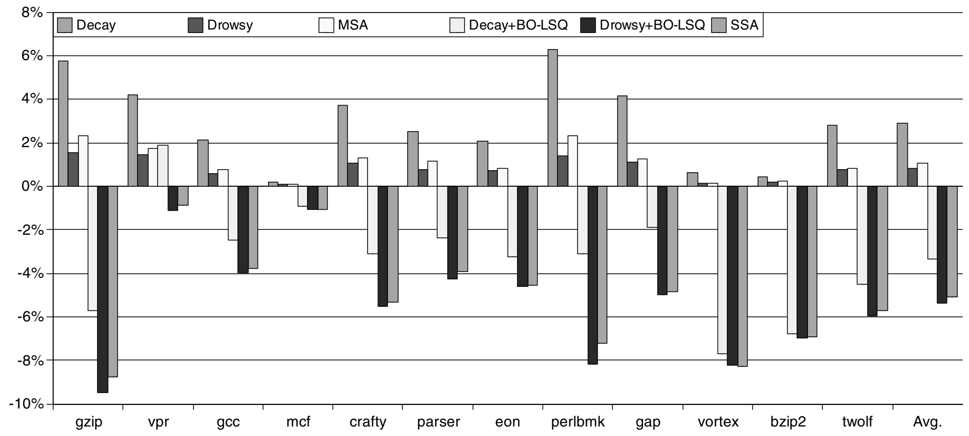


Fig. 25. Runtime impact of BO–LSQ.

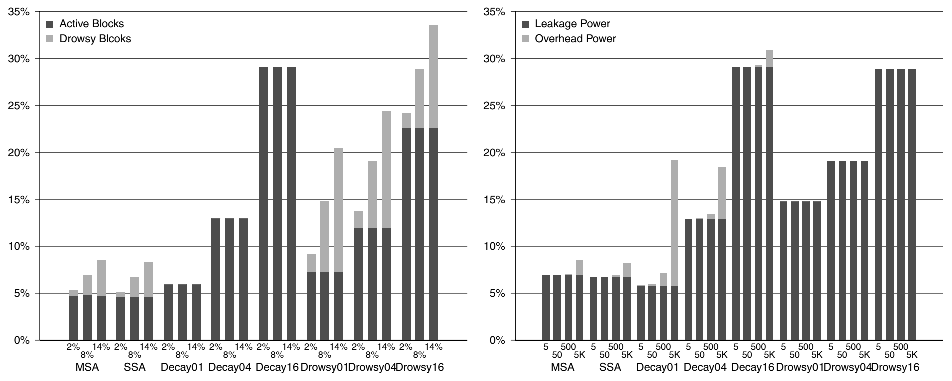
5.4 Applying BO–LSQ to Other Techniques

BO–LSQ is the technique used by this paper to compensate the performance loss incurred by the MSA scheme of SSA. However, this technique is not limited only to SSA. In fact, BO–LSQ can be adopted by cache decay and drowsy cache to offset the performance penalties as well. Figure 25 shows that BO–LSQ can recover the 3 and 1% performance degradations of cache decay and drowsy cache, respectively, and achieve performance improvement of 3 and 5%, respectively, for the SPECint2000 benchmarks, on average.

5.5 Sensitivity to Parameters

There are two parameters used to compute normalized leakage energy of L1 caches: R_{sleep} is the ratio of static energy of a drowsy block relative to an active block and R_{L2} relates the dynamic energy due to an additional miss (or write-back) to static leakage energy of a single clock cycle in the L1 cache. Figure 26 shows how changes in these two ratios affect the leakage savings by the various leakage reduction techniques listed in Table III. It demonstrates that SSA is relatively insensitive to these parameters than cache decay and drowsy cache.

Figure 26a depicts the average normalized leakage energy of L1 data caches for the SPECint2000 benchmarks when R_{sleep} is set to 2, 8, and 14%, respectively. It reveals that SSA produces the best leakage reduction with R_{sleep} = 2%, and dissipates comparable static energy with cache decay when R_{sleep} is equal to 8%. Furthermore, SSA outperforms drowsy cache even with the worst R_{sleep} ratio. Another way to view Fig. 26a is to perform a breakeven analysis by taking horizontal slices of the figure [Prybylski et al. 1988]. The configurations in the same horizontal slice achieve the same level of leakage reduction. For instance, the slice of 5–10% indicates that SSA with any of these three R_{sleep} ratios, Decay01, and Drowsy01 with R_{sleep} = 2% are the best among all possible configurations, while the slice of 30–35% exposes that Drowsy16 with R_{sleep} = 14% is the worst arrangement.



(a) R_{sleep} (b) R_{L2}

Fig. 26. Impact of parameters on leakage.

Figure 26b analyzes the dynamic energy overheads of extra L1 misses incurred by different leakage reduction techniques with various R_{L2} ratios ranging from 5 to 5000. Kaxiras et al. [2001] observed that the R_{L2} ratio could be as low as 8.9 on Alpha 21164, and predicted this ratio would be lower because of higher leakage in future techniques. All the experiments presented in Section 4 were conducted with $R_{L2} = 50$. $R_{L2} = 500$ and 5000 are two extreme cases that might occur when L2 caches are not on-chip or even no L2 caches are present. This figure shows that extra L1 misses introduced by SSA incur negligible dynamic energy overheads. Dynamic energy overheads of SSA are visible only at the extreme case of $R_{L2} = 5000$, but still take only relatively small percentages. Furthermore, performing a breakeven analysis on this figure also reveals that SSA will be among the configurations in the horizontal slice of the best leakage performance regardless of the R_{L2} values.

6. RELATED WORK

Selective cache ways proposed the first method of turning off unneeded ways to reduce cache energy [Albonesi 1999]. It is a coarse-grain approach that can turn off entire cache ways of set-associative caches for different programs. Recently, several fine-grain techniques have been developed that turn off or deactivate individual cache lines to reduce power consumption [Flautner et al. 2002; Hanson et al. 2001; Kaxiras et al. 2001; Kim et al. 2002; Zhou et al. 2003]. 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 [Peng et al. 2004]. 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 [Nicolaescu et al. 2003]. This technique can be integrated in BO–LSQ, as well, to reduce accesses to cache blocks. The difference is that the cached LSQ still has to perform address computation while cached BO–LSQ can skip this process if the effect addresses can be looked up from BO–LSQ entries.

There are also other cache systems designed by other researchers to reduce static power dissipation of instruction caches [Allu and Zhang 2004; Kim et al. 2002; Hu et al. 2003; Yang and Lee 2004]. They all have deployed techniques similar to those used in reducing leakage power of data caches. In addition, reducing dynamic power consumption is a very important issue as well. Researchers have implemented techniques to trim the frequencies of accessing cache lines to save power [Powell et al. 2001; Zhang et al. 2004].

Recently Meng et al. [2005] 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. The work is closely related to this paper, but the model requires oracle knowledge of future accesses. On the contrary, the SSA of this paper can improve performance without such oracle information while significantly cutting leakage dissipation of data and instruction caches.

7. CONCLUSIONS

This paper presents a cache architecture, the SSA cache, that can reduce leakage power without incurring performance penalties. The SSA cache implements the MSA scheme, which only activates the minimal numbers of ways in every cache set; and It cuts static power consumption of instruction and data caches by over 92%, on average, for SPECint2000 benchmarks. In addition, the performance losses caused by the MSA scheme are compensated by the BO–LSQ design and the average execution times of SPECint2000 benchmarks are reduced by 2–5%. In addition, cached BO–LSQ can further extend the performance improvement by an additional 1%. Meanwhile, since average access rate of data caches is trimmed by 11%, the dynamic power of data caches should be cut by a similar proportion as well.

REFERENCES

Albonesi, D. H. 1999. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, Washington, D.C. 248–259.

Allu, B. and Zhang, W. 2004. Static next sub-bank prediction for drowsy instruction cache. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM Press, New York. 124–131.

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

Baugh, L. and Zille, C. 2004. 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)*.

Borkar, S. 1999. Design challenges of technology scaling. *IEEE Micro 19,* 4, 23–29.

Burger, D. C., Goodman, J. R., and Kägi, A. 1995. The declining effectiveness of dynamic caching for general-purpose microprocessors. Technical report 1261, University of Wisconsin-Madison Computer Science Department.

Butts, J. A. and Sohi, G. S. 2000. A static power model for architects. In *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*. ACM Press, New York. 191–201.

Dean, J., Hicks, J. E., Waldspurger, C. A., Weihl, W. E., and Chrysos, G. 1997. Profileme: hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, Washington, D.C. 292–302.

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

Flautner, K., Kim, N. S., Martin, S., Blaauw, D., and Mudge, T. 2002. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. IEEE Computer Society, Washington, D.C. 148–157.

Hanson, H., Hrishikesh, M., Agarwal, V., Keckler, S., and Burger, D. 2001. Static energy reduction techniques for microprocessor caches. In *Proceedings of 2001 International Conference on Computer Design*. IEEE Computer Society, Washington, D.C. 276–283.

Hennessy, J. L. and Patterson, D. A. 2003. *Computer Architecture: A Quantative Approach*, 3rd ed. Morgan Kaufmann, San Mates, CA.

Hu, J., Nadgir, A., Vijaykrishnan, N., Irwin, M., and Kandemir, M. 2003. Exploiting program hotspots and code sequentiality for instruction cache leakage management. In *Proceedings of 2003 International Symposium on Low Power Electronics and Design*. 402–407.

Intel Corporation. 2001. *Intel XScale Microarchitecture*. Intel Corporation, Santa Clara, CA.

Kaxiras, S., Hu, Z., and Martonosi, M. 2001. Cache decay: exploiting generational behavior to reduce cache leakage. In *Proceedings of the 28th annual International Symposium on Computer Architecture*. ACM Press, New York. 240–251.

Kaxiras, S., Xekalakis, P., and Keramidas, G. 2005. A simple mechanism to adapt leakage-control policies to temperature. In *Proceedings of 2005 International Symposium on Low Power Electronics and Design*. 54–59.

Kessler, R. 1999. The Alpha 21264 microprocessor. *IEEE Micro 19,* 2, 24–36.

Kim, N. S., Flautner, K., Blaauw, D., and Mudge, T. 2002. 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*. IEEE Computer Society Press, Washington, D.C. 219–230.

Kim, N. S., Flautner, K., Blaauw, D., and Mudge, T. 2004a. Circuit and microarchitectural techniques for reducing cache leakage power. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 12,* 2 (Feb.), 167–184.

Kim, N. S., Flautner, K., Blaauw, D., and Mudge, T. 2004b. Single-V_{DD} and single-V_t super-drowsy techniques for low-leakage high-performance instruction caches. In *Proceedings of 2004 International Symposium on Low Power Electronics and Design*. 54–57.

Li, L., Degalahal, V., Vijaykrishnan, N., Kandemir, M., and Irwin, M. J. 2004. Soft error and energy consumption interactions: a data cache perspective. In *Proceedings of the 2004 international symposium on Low power electronics and design*. ACM Press, New York. 132–137.

Meng, Y., Sherwood, T., and Kastner, R. 2005. On the limits of leakage power reduction in caches. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*.

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

PARK, I., OOI, C., AND VIJAYKUMAR, T. 2003. Reducing design complexity of the load/store queue. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, Washington, D.C. 411–422.

PENG, L., PEIR, J.-K., AND LAI, K. 2004. Signature buffer: Bridging performance gap between registers and caches. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*. 164–175.

POWELL, M. D., AGARWAL, A., VIJAYKUMAR, T. N., FALSAFI, B., AND ROY, K. 2001. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, Washington, D.C. 54–65.

PRYBYLSKI, S., HOROWITZ, M., AND HENNESSY, J. 1988. Performance tradeoffs in cache design. In *Proceedings of the 15th Annual International Symposium on Computer architecture*. IEEE Computer Society Press, Washington, D.C. 290–298.

STANDARD PERFORMANCE EVALUATION CORPORATION. 2000. SPEC CPU2000 v1.1.

TANENBAUM, A. S. 1999. *Structured Computer Organization*, 4th ed. Prentice-Hall, Englewood Cliffs, NJ.

TRANSMETA CORPORATION. 2004. *Crusoe Processor Model TM5700/TM5900 Data Book*.

WEAVER, C. SPEC 2000 binaries. http://www.eecs.umich.edu/~chriswea/benchmarks/spec2000.html.

YANG, C.-L. AND LEE, C.-H. 2004. Hotspot cache: Joint temporal and spatial locality exploitation for i-cache energy reduction. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*. ACM Press, New York. 114–119.

ZAGHA, M., LARSON, B., TURNER, S., AND ITZKOWITZ, M. 1996. Performance analysis using the mips r10000 performance counters. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing*. ACM Press, New York.

ZHANG, C., VAHID, F., AND NAJJAR, W. 2003. A highly configurable cache architecture for embedded systems. In *Proceedings of the 30th annual international symposium on Computer architecture*. ACM Press, New York. 136–146.

ZHANG, C., VAHID, F., YANG, J., AND NAJJAR, W. 2004. A way-halting cache for low-energy high-performance systems. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*. ACM Press, New York. 126–131.

ZHOU, H., TOBUREN, M. C., ROTENBERG, E., AND CONTE, T. M. 2003. Adaptive mode control: A static-power-efficient cache design. *ACM Transactions on Embedded Computing Systems (TECS) 2,* 3, 347–372.

Received August 2005; revised January 2006; revised June 2006; accepted July 2006