Cache Memory Architecture for Leakage Energy Reduction

Kiyofumi Tanaka
School of Information Science, Japan Advanced Institute of Science and Technology
kiyofumi@jaist.ac.jp

Abstract

Recently, energy dissipation by microprocessors is getting larger, which leads to a serious problem in terms of allowable temperature and performance improvement for future microprocessors. Cache memory is effective in bridging a growing speed gap between a processor and relatively slow external main memory, and has increased in its size. Almost all of today's commercial processors, not only high-performance microprocessors but embedded ones, have on-chip cache memories. However, energy dissipation in the cache memory will approach or exceed 50% of the increasing total energy dissipation by processors. An important point to note is that, in the near future, static (leakage) energy will dominate the total energy consumption in deep sub-micron processes. This paper describes cache memory architecture, especially for on-chip multiprocessors, that achieves efficient reduction of leakage energy in cache memories by exploiting gated-Vdd control, software self-invalidation for L1 cache, and dynamic data compression for L2 cache. The simulation results show that our techniques can reduce a substantial amount of leakage energy without large performance degradation.

1. Introduction

In recent years, energy consumption of a microprocessor is getting larger due to increasing transistor counts according to Moore's Law and improvement of operation clock frequency. The high energy consumption makes a lifetime of increasingly common battery-powered devices short. In addition, the increase of energy dissipation raises the temperature of LSIs and consequently violates an operational condition or becomes an obstacle to progress of microprocessor's running clock frequency. Therefore, reduction of energy consumption is indispensable to performance improvement of future microprocessors.

Several commercial processors have facilities for dynamic voltage and frequency scaling (DVS, DVFS) [21] which is an effective technique for energy reduction by dynamically lowering the voltage and clock frequency, since energy consumption of CMOS devices depends on the driving voltage and clock frequency. However, DVS mainly reduces dynamic energy and has little effect on static or leakage energy. For future sub-micron processes, some techniques for static energy reduction are required.

On the other hand, large-scale and sophisticated software is spreading and working set size in applications is getting larger. Therefore, high performance processing requires a large amount of cache memory in order to bridge a speed gap between a processor and external memory. Consequently, energy dissipation in the cache memory exceeds 50% of the total consumption by a processor [2]. The energy reduction in cache memories is essential and some solution to the problem must be provided for future microprocessor architecture, especially in terms of leakage energy that would be more serious in the future sub-micron processes.

On-chip multiprocessors are becoming popular since they have the advantage of high performance while they consume relatively low energy compared to uniprocessors with a higher clock frequency. In this paper, we describe a low-energy cache memory hierarchy for on-chip multiprocessors, that exploits gated-Vdd transistors [14] and explicit gated-Vdd control. The primary cache memory performs explicit control of gated-Vdd transistors, which is caused by execution of some special load and store instructions. We call this technique "software self-invalidation". On the other hand, a data compression technique is applied to the secondary cache and vacant areas are turned off by the gated-Vdd, which does not increase additional cache misses that generates long penalty due to external memory accesses.

Section 2 describes several related works on leakage energy reduction in cache memories. In Section 3, we give a cache memory hierarchy for on-chip multiprocessors. In Section 4, the software self-invalidation technique for primary caches is shown. Section 5 describes the leakage energy reduction scheme for secondary cache memories with the hardware data compression. Section 6 shows the simulation results and the effectiveness of the energy reduction

1527-1366/07 $25.00 © 2007 IEEE
DOI 10.1109/IWIA.2007.12

schemes and Section 7 concludes this paper.

2. Related Work

There are several architectural techniques proposed for leakage energy reduction in cache memories. Dynamically ResIzable instruction cache (DRI i-cache) reduces energy dissipation by dynamically downsizing effective caching areas [17]. Whether downsizing or upsizing is performed depends on the number of cache misses that occurred in an execution interval. When the miss count is smaller than a bound given in advance, the cache is downsized, and vice versa. The area that is not to be accessed is turned off by controlling gated-Vdd transistors and does not consume static energy after that [14]. This method focuses only on an instruction cache and the whole cache is divided into only two parts, active and sleeping areas.

There are other methods that are based on fine-grain gated-Vdd control. Cache decay is an energy-reduction scheme that controls gated-Vdd transistors per cache block [18]. A block is in a dead-time state when it is in the interval between the last access to the block and replacement. Blocks in the dead-time state are turned off by gated-Vdd control and then any static energy is not wasted for the blocks. However, it is impossible for a hardware mechanism to exactly decide whether a block is in dead-time or not. In their hardware organization, the decision depends on a counter value for each block. The counter counts cycles or ticks during which the block is not accessed. When the counter gets saturated, the corresponding block is regarded as having entered dead-time. This mechanism requires extra hardware for the counters and cannot eliminate misjudgment completely due to various access patterns in applications that include both short and long access intervals.

Cache blocks that are turned off cannot preserve data values in the methods mentioned above. Therefore, reaccessing such blocks causes a cache miss and involves a miss penalty and additional dynamic energy to access lower-level memories. On the other hand, there are state-preserving techniques, ABC-MT-CMOS (Auto-Backgate-Controlled Multi-Threshold CMOS) [9] and drowsy cache [8]. ABC-MT-CMOS is a technique where threshold voltages are dynamically manipulated and leakage energy is reduced. Memory cells can retain values even in a sleep mode. However, reaccessing the sleep cells requires waiting for the cells to wake up. The MT-CMOS requires complex circuitry and therefore tends to increase the hardware size.

Drowsy cache prepares two different supply-voltage modes, high-power and low-power modes, instead of turning off. Cache blocks in the low-power mode cannot be read or written. Although the amount of energy reduction is smaller than the gated-Vdd control, blocks even in the low-power mode can preserve data values. Each block periodically falls into the low-power mode, and is woken up to the high-power mode when the block is reaccessed. The penalty for waking up a low-powered block is much smaller than that in the gated-Vdd controlled caches. This mechanism expects the characteristics in programs that there are a limited number of memory blocks that are frequently accessed in some short period, and effectively reduces leakage energy.

The defects of the gated-Vdd control are an additional cache miss penalty caused by data disappearance (mis-shutdown) and increase of dynamic energy consumption for accessing the next level memory hierarchy on the misses. On the other hand, those of state preserving ones such as drowsy caches are relatively large additional hardware and lower efficiency of leakage energy reduction since any memory cells always keep some voltage.

Our methods shown in this paper aims at eliminating extra cache misses caused by mis-shutdown while achieving as much energy reduction as gated-Vdd control, by using software self-invalidation for primary caches and dynamic data compression for a secondary cache.

3. Cache Memory Hierarchy

As depicted in Figure 1, the memory hierarchy of our system consists of multiple processor cores each of which has primary (L1) instruction and data caches based on a writeback scheme, a write buffer, a secondary (L2) unified writeback cache on chip, and an external main memory. All processor cores and the write buffer are connected by a shared bus. Data in L1 caches are kept coherent by following a snoop-based invalidation protocol.

The leakage energy reduction method for the L2 cache exploits data compression and decompression, and therefore the compressor and decompressor hardware components are provided in the memory hierarchy system. This organization is similar to that in the literature [3] which has compressor and decompressor, except that they assumes an exclusion property between L1 and L2. On the other hand, our L1 and L2 have an inclusion property that is more natural for today's microprocessor architecture. In addition, the objective in the literature [3] is virtual increase of associativity of the cache where the cache can behave as any associativity from 4-way to 8-way depending on compression results. On the other hand, we aim at reducing leakage energy consumption in the L2 cache without a change of the associativity.

On writeback from L1 to L2 cache in replacement or on filling from external main memory to L2 cache, block data pass through the pipelined compressor. When missing in L1 cache and hitting in L2 cache, or when a block in the L2 is replaced, the compressed block data are decompressed by

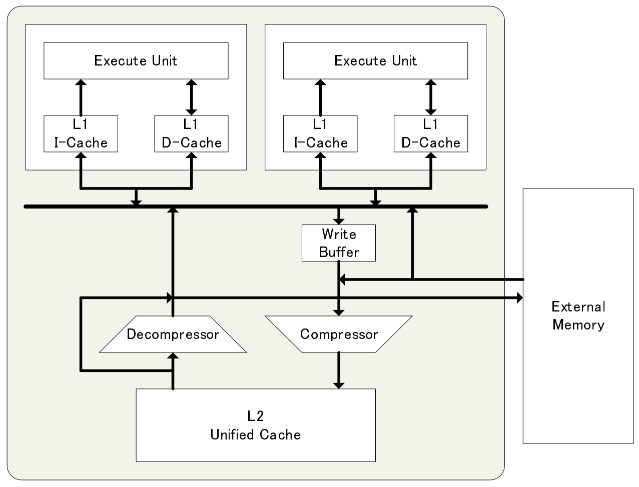


Figure 1. Cache memory hierarchy.

the pipelined decompressor. When the block is the original (uncompressed) one, the decompressor is bypassed and the data are directly sent to the L1 cache and the execution pipeline, or to external main memory, in order to eliminate the decompression overheads.

4. Leakage Energy Reduction by Software Self-Invalidation

In multiprocessor systems with snooping caches, cache blocks can get invalid when receiving invalidation requests. It is an effective way to turn off those invalid blocks by controlling gated-Vdds. In addition, self-invalidation can increase the number of blocks that can be turned off.

Self-invalidation was originally a technique for mitigating overheads of cache coherence management in distributed shared memory [5, 1]. We apply the concept of self-invalidation to energy reduction in cache memory. The self-invalidation methods proposed in the literatures [5, 1] were mechanisms that were controlled fully by hardware. Therefore, they are not appropriate for energy-reduction since they require special hardware, version number directory or signature tables, that would consume dynamic energy by itself. Then we introduce a software self-invalidation technique in our system.

4.1. Last-touch memory access instructions

For efficiency of software self-invalidation, we introduce the instructions, last-touch load/store, execution of which not only functions as conventional load/store but also invalidates cache blocks after accessing them. There are two types of condition for invalidation as follows.

- A cache block is invalidated at the same time as it is accessed.
- A word is marked when it is accessed. The cache block is invalidated when all words in it get marked.

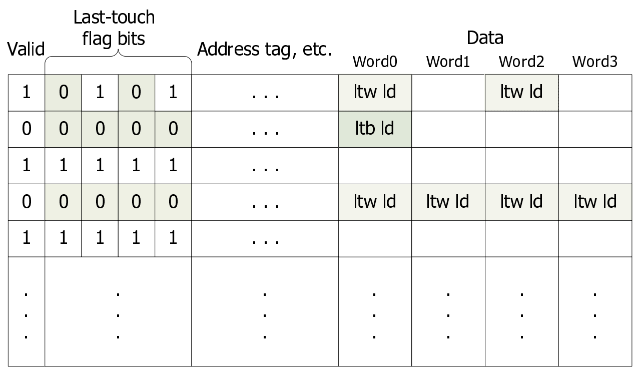
We call the former type of instructions "last-touch-block load/store (ltb ld/st)", and the latter "last-touch-word load/store (ltw ld/st)". When write-back policy is employed and a block that is designated to be invalidated is of a modified state, the invalidation is performed after write-back operation or insertion into a write buffer.

For example, load/store instructions that access each address only once before it is invalidated by other processor caches can be replaced by the last-touch load/store instructions. Similarly, ones that access each address only once in a generation (between block filling and replacement) can be replaced by the last-touch load/store instructions.

4.2. Hardware Mechanisms

It is necessary to give a small modification to conventional L1 cache memory structure to reduce energy dissipation by using the last-touch instructions.

Last-touch flag bits are a part of L1 cache tag information and indicate which word in the cache block has been accessed by the last-touch load or store instruction. Figure 2 shows the cache memory structure including the last-touch flag bits, when the block size is 16 bytes and a word size is 4 bytes. A single last-touch flag bit corresponds to a word in the block. When a last-touch-word load/store instruction is executed, the corresponding flag bit is cleared. On the other hand, when a last-touch-block load/store instruction is executed, all flag bits are cleared (as depicted in the second row in the figure). Then, a block is invalidated when all the flag bits are cleared.



Valid	Last-touch flag bits				Address tag, etc.	Data			
						Word0	Word1	Word2	Word3
1	0	1	0	1	. . .	ltw ld		ltw ld	
0	0	0	0	0	. . .	ltb ld			
1	1	1	1	1	. . .				
0	0	0	0	0	. . .	ltw ld	ltw ld	ltw ld	ltw ld
1	1	1	1	1	. . .				
⋮	⋮				⋮	⋮	⋮	⋮	⋮

Figure 2. L1 cache memory structure.

A valid bit of the whole cache block can be generated by a logical disjunction (OR) of the last-touch flag bits. In other words, last-touch flag bits are regarded as a valid bit of each word. The last-touch flag bits are additional hardware to conventional cache tag information. We choose flag bits per word, not per byte, considering that the additional hardware amounts should be small and that applications often process data on a word basis.

We assume that the gated-Vdd (or gated-Vss) is implemented by following the technique proposed by Yang, et al. [17]. This is a wide NMOS dual-Vt gated-Vdd with a charge pump and has about 5% of area overheads. The gated-Vdd transistor is inserted between ground and SRAM cells (virtual ground). When the gated-Vdd transistor is turned off, the leakage energy is virtually eliminated. Figure 3 is a conceptual diagram of an L1 cache block. The address tag and data parts of the block are connected with one or more gated-Vdd transistors. The gated-Vdd transistors are controlled by the valid bit. When the valid bit is one, the gated-Vdd is turned on, otherwise, turned off and leakage energy in the address tag and data areas is eliminated. (When the valid bit is prepared separately from last-touch flag bits, the last-touch flag bits can be turned off as well.)

After a block is turned off, it takes a certain delay to wake the block up again. This wakeup latency depends on the LSI process used and the number of bits that a single gated-Vdd transistor takes charge of. Short wakeup latency is desirable from a performance point of view [16]. Kaxiras, et al. were optimistic about the wakeup latency, since they estimated that the latency is hidden by an L1 cache miss penalty [18]. Several researches adopted relatively short time, a few cycles, as the wakeup latency [22][23]. We follow the same (optimistic) assumption in this paper, since the latency can be hidden by the effect of the write buffer or by access latency for the next level in the hierarchy.

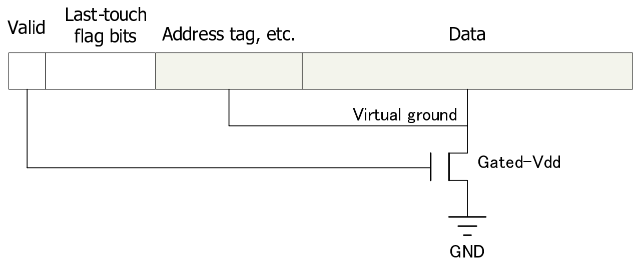


Figure 3. L1 gated-Vdd control.

5. Leakage Energy Reduction by Data Compression

The size and energy dissipation of L2 cache memories would grow in future processors, and therefore we try to efficiently reduce leakage energy by gated-Vdd, without increasing cache misses. We proposed the technique that used gated-Vdd control per block and data compression [10, 11]. In our approach, only cache areas that had no valid data were turned-off, which means that the turned-off areas lead to no additional cache misses and the drawbacks of gated-Vdd control that data might disappear can be eliminated. This section describes the summary of the technique we proposed, including a compression algorithm that we apply newly in this paper.

5.1. Data compression schemes in secondary cache

In our strategy, data in a secondary cache memory are compressed and the areas vacated by the compression are turned off by controlling gated-Vdd transistors, which leads to effective reduction of leakage energy. We use compression thresholds of 1/4, 1/2 and 3/4. For example, when a block could be compressed into smaller than a fourth, the compressed block data are stored in L2 cache and the other three fourths area is turned off. When a block could not be compressed into smaller than three fourths, the original block is stored as it is. Although compression and decompression overheads exist when accessing the secondary cache, they are not significant since a frequency in accessing the secondary cache is not high.

There are several hardware algorithms proposed for data compression. Although the compression ability in the sense that how small data can be compressed into is an important point, the hardware size of the compressor/decompressor and the compression/decompression latency are more important from the point of view of a tradeoff between the cost/performance and the amount of energy-reduction.

In our previous study, we applied several hardware compression/decompression algorithms, frequent pattern compression (FPC) [4], frequent value compression (FVC) [7], X-match algorithm and X-RL algorithm [13]. In the FPC, pattern matching rules are applied to cache block data on a word-by-word basis. The rules consist of zero run, 4-, 8-, or 16-bit data that can be decompressed to the original 32-bit data by sign-extension, repeated bytes, and so on. The FVC performs compression by referring a dictionary that contains frequent values in a running program. The X-match temporarily constructs a dictionary for a given memory block and uses it for compression of the block. X-RL is the same as the X-match, except it includes a zero run rule. In the evaluation, X-RL was the best among all the above algorithms in terms of compression sizes of cached blocks

[11].

The evaluation in [11] was targeted to programs that dealt mainly with integer data. The X-RL algorithm might not fit applications with many floating point data. Therefore, we investigate another compression algorithm that may be appropriate for floating point data, "floating-point compression algorithm (FPCA)" [12]. FPCA performs compression per 64-bit data that is the size of a double-precision floating-point number, and exploits value prediction techniques which are originally for speculative execution. It reads a datum in a block and predicts the next value. Then, the difference between the predicted value and the actual value is used as a compressed datum. The prediction techniques are FCM [24] and DFCM [6] that are context-based predictors.

5.2. Hardware mechanisms

Figure 4 depicts the tag information for the L2 cache. The "C1" and "C0" are the compression bits which indicate how the corresponding block has been compressed. That is, the value of "11" in the bits means the block has been compressed into smaller than a fourth, "10" is a half, "01" is three fourths, and "00" means the block couldn't be compressed. This information is directly used for controlling the gated-Vdd transistors where at least three gated-Vdd transistors are required for a cache block to support the above compression grain. This compression-bit field should be held by some flip flops separated from the tag memory. (Actually, this field does not need to be visible to any software.)

The tag includes an "M" bit field that indicates the block has been modified, since the L2 cache is managed by write-back policy. The L1 caches also follow writeback policy to alleviate the compression overheads in the L2 on every store operation. For cache coherence management in case of a multiprocessor configuration, the modified bit information in the L1 caches should be immediately propagated to the L2 tag on every update in the L1 cache.

Figure 5 is a conceptual diagram of an L2 cache block. The rightmost part of the block is directly connected with a ground, that is, the SRAM cells are always active with the value of either one or zero. The other three parts are connected with one or more gated-Vdd transistors. Each gated-Vdd transistor is controlled by the compression bits. When the compression bits are "00", all the gated-Vdd transistors are turned on. When "01", the leftmost gated-Vdd is turned

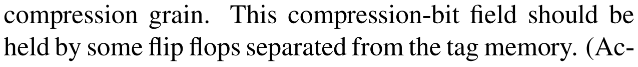


Figure 4. Tag in L2 cache.

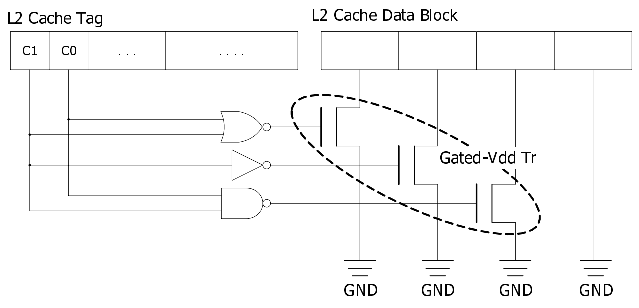


Figure 5. L2 gated-Vdd control.

off, and so on.

6. Evaluation

6.1. Simulation environment

We developed a scalar processor simulator that executes the SPARC version 9 instruction set [19]. Binary executable files generated by GNU C compiler were input to the simulator. The simulator executes an instruction per cycle (several instructions such as multiplication and division take three or more cycles) and outputs the total execution cycles and other informations; the number of cache misses, write buffer stall cycles, consumed leakage energy, etc. The simulator has two target processors for multiprocessor configuration. Each processor has L1 instruction/data split caches that follow write-back policy. The L1 caches are connected by a shared bus and managed based on a write invalidate protocol. The system includes a write buffer and an L2 unified cache that is shared by all processors. The block size, associativity, total cache size, and access latency can be set up to any values.

In the simulation, the simulator calculated leakage energy in the L1 and L2 caches by using the following formula.

$$\begin{aligned}Leak\ energy &= Active\ cells \times Active\ leakage \times Active\ cycles\\ &+ Standby\ cells \times Standby\ leakage \times Standby\ cycles\end{aligned}$$

The active leakage is for a turned-on cell and the standby leakage is for a turned-off cell. The parameter values that were shown in the literature [14] were applied to the above formula (1740 nJ/s for an active cell and 53 nJ/s for a standby cell, under 0.18μm).

6.2. Effects of self-invalidation scheme

For evaluation of software self-invalidation, we simulated five kernel programs in the SPLASH-2 suite [15] in multiprocessor configuration. The input data sizes and input file in the five programs are shown in Table 1.

The programs were simulated in advance to generate traces of memory accesses. After that, we updated the program codes by manually replacing load/store instructions with last-touch ones, referring to the traces. For the last-touch instructions, we exploited load/store instructions from/to an alternate space that are implementation-dependent instructions in the SPARC architecture. These instructions can specify an address space identifier (ASI). We used a discrete ASI value for each of last-touch-block and last-touch-word instructions. We set up the L1 instruction and data caches as 32KB, 4-way set associative, 16B block, and 1-cycle latency on hit, and the L2 cache as infinite size, 16B block, and 10-cycle on hit, and the write buffer as infinite size and 1-cycle for insertion.

Table 2 shows the results of the five programs on three different execution schemes; "base" is the execution without gated-Vdd control, "inv.off" is the execution with gated-Vdd control of invalid blocks, and "last-touch" is the execution with gated-Vdd control of invalid blocks supported by last-touch load/store instructions. In the table, "Exec. time" is the relative execution time normalized to the base. Similarly, "Leakage energy" is the relative leakage energy normalized to the base.

The "inv.off" execution reduced 15.5% of leakage energy in LU-contig, 33.0% in LU-noncontig, and 2.6% in RADIX. For FFT and CHOLESKY, the execution could not reduce leakage energy. (0.6% and 0.08%, respectively.)

For all of the five programs, our method ("last-touch") reduced more leakage energy than the simple "inf.off" execution; 2.5% of leakage energy in FFT, 20.6% in LU-contig, 46.3% in LU-noncontig, 46.5% in RADIX, and 1.0% in CHOLESKY. Table 3 shows the number of self-invalidation by last-touch-block (ltb) and that by last-touch-word (ltw) instructions, and the number of last-touch-word instructions executed. Roughly, one self-invalidation operation is performed every four executions of the last-touch-word instructions. The table 2 and table 3 show that a large amount of leakage energy was reduced by "last-touch" for LU-noncontig and RADIX, which included many self-invalidations.

For all the programs, the "last-touch" execution decreased the execution cycles, although the difference was small. This is because the number of cache misses was decreased. The caches basically employed LRU replacement policy where a block that was least recently used was replaced. On replacement, an invalid block entry, if it existed, was selected for an entry that was filled with a missing block. Therefore, the self-invalidation facilitates optimal replacement decision by invalidating blocks that are already in dead-time. On the other hand, without self-invalidation, the simple LRU might replace blocks that are still in live-time and lead to cache misses later. This is the reason why the execution time of "last-touch" was shorter than those of "base" and "inv.off".

Table 1. Input data sizes and input file for SPLASH-2 five programs.

Program	Input data size / input file
FFT	65,536 complex
LU contig	256x256 matrix
LU non-contig	256x256 matrix
RADIX	262,144 keys
CHOLESKY	wr10.O

Table 2. Results of last-touch load/store scheme in L1 cache.

Program	Exec. scheme	Exec. time	Leakage energy
FFT	base	1.0000	1.0000
	inv.off	1.0000	0.9938
	last-touch	0.9999	0.9752
LU-contig	base	1.0000	1.0000
	inv.off	1.0000	0.8447
	last-touch	0.9999	0.7943
LU-noncontig	base	1.0000	1.0000
	inv.off	1.0000	0.6697
	last-touch	0.9968	0.5369
RADIX	base	1.0000	1.0000
	inv.off	1.0000	0.9741
	last-touch	0.9989	0.5353
CHOLESKY	base	1.0000	1.0000
	inv.off	1.0000	0.9992
	last-touch	0.9997	0.9895

Table 3. The number of self-invalidation.

Program	# of self-invalidation		# of ltw
	ltb	ltw	instructions
FFT	36	66,044	264,190
LU contig	18	82,150	396,571
LU non-contig	18	157,156	908,555
RADIX	25	272,420	1,179,675
CHOLESKY	9	14,839	71,630

6.3. Effects of data compression scheme

We show the results of the data compression scheme for the L2 leakage reduction. We used seven programs in the SPEC CINT95 suite [20]; 099.go, 124.m88ksim, 129.compress, 130.li, 132.ijpeg, 134.perl, 147.vortex, and 130.li. Table 4 shows the seven programs. (Working sets in SPLASH-2 programs were not large enough to utilize the L2 cache. Therefore we did not use them for this evaluation.) For each program, the first one billion of instructions for initialization were skipped and the next one billion of instructions were traced. We set up the L1 instruction and data caches as 64KB, 2-way set-associative and 32B block, and the L2 cache as 1MB, 2-way and 32B block. The write buffer had 8 entries. The latency of compression and decompression was assumed 10 cycles. For the frequent value compression (FVC) algorithm, we used 16 frequent values that were obtained in the execution of the first 100 million instructions. For FPCA, we used the level three, which means a $2^3 = 8$-entry table was used to store data contexts.

Table 5 shows the results of the seven programs on five different compression schemes. The values in the table are the relative leakage energy normalized to the base execution without gated-Vdd control or compression. Of the five compression algorithms, X-RL cut leakage energy most for all programs. The execution of 099.go, 124.m88ksim, 130.li, 134.perl, and 147.vortex could reduce more than 30% of leakage energy by X-RL. However, the execution of 129.compress could not reduce enough leakage energy. The execution with FPCA could not match that with X-RL for these integer applications. We will investigate the ability of FPCA by using SPEC CFP programs in future works.

We confirmed that influence of the compression and decompression overheads was insignificant. As for compression, the write buffer could absorb the overheads since blocks that were written back to the L2 cache were inserted into the buffer while the execution pipeline could continue to run. After leaving the buffer, the blocks are sent to the compressor. On the other hand, decompression latency cannot be hidden. However, the frequency of accesses to compressed blocks in the L2 was not high. Therefore, additional execution time was relatively short. At worst, 3.7% degradation in execution time was observed in 130.li with X-RL. On average, the degradation was small, 1.82%, with X-RL.

Table 4. Benchmark programs.

Program	Outline	Input
099.go	An internationally ranked go-playing program	ref
124.m88ksim	A chip simulator for Motorola 88100 microprocessor	ref
129.compress	An in-memory version of the common UNIX utility	ref
130.li	Xlisp interpreter	ref
132.ijpeg	Image compress/decompress on in-memory images	ref
134.perl	An interpreter for the Perl	ref
147.vortex	An object oriented database	ref

7. Conclusion

Increasing energy consumption of microprocessors not only raises the temperature of the LSIs but is an obstacle to improvement of running speed. Therefore, reduction of energy dissipation is essential to performance improvement of future microprocessors. In this paper, we focused on cache memories the size and energy consumption of which are growing, and showed leakage energy reduction methods that utilize gated-Vdd control, the software self-invalidation scheme and dynamic data compression scheme.

In the evaluation, the software self-invalidation technique could reduce leakage energy consumption in the L1 cache without performance degradation compared to the execution without the self-invalidation. The evaluation used codes that were generated by manual translation based on execution traces, which brought optimal application of last-touch instructions, but would not be practical in actual software execution. We will explore other methods of automatic code generation. For example, the literature [25] showed that cache misses could be decreased by compiler optimization that made load and store instructions have information about temporal and spatial locality between instructions. We have prospects of applying similar compiling techniques for leakage energy reduction.

The data compression technique reduced leakage energy consumption in the L2 cache with less than 4% performance degradation. Our evaluation did not include neither dynamic energy consumption nor energy by the tag memory. In the future, we will try to evaluate total (static and dynamic) energy including that of the compressor and decompressor hardware. For the purpose, we will design the compressor and decompressor hardware, which will reveal exact overheads and the hardware size.

In addition, for more practical usage, we will evaluate the techniques, software self-invalidation for L1 and dynamic compression for L2 simultaneously, with larger block sizes appropriate for the secondary caches, by using a simulator of a processor whose execution is based on instruction-level parallelism, superscalar and out-of-order execution, where a rate of memory references would be higher. Furthermore, we will try to evaluate our methods for embedded processors which suffer energy limitation more seriously and have smaller caches that cause more cache misses, which might impact the methods.

Table 5. Results of data compression scheme in L2 cache.

Program	Compression algorithm				
	fpc	fvc	x-match	x-rl	fpca
099.go	0.7227	0.9671	0.7647	0.6484	0.8673
124.m88ksim	0.8066	0.8793	0.6496	0.6201	0.9920
129.compress	0.9955	0.9981	0.9842	0.9824	0.9845
130.li	0.7561	0.7995	0.7522	0.6789	0.9346
132.ijpeg	0.8630	0.8898	0.9105	0.8601	0.8907
134.perl	0.7055	0.9821	0.7164	0.6201	0.8056
147.vortex	0.5624	0.7097	0.6913	0.5550	0.7579

References

[1] A.C.Lai and B.Falsafi. Selective, Accurate and Timely Self-Invalidation Using Last-Touch Prediction. In *Proc. of ISCA*, pages 139–148, June 2000.

[2] A.Malik, W.Moyer, and D.Cermak. A Low Power Unified Cache Architecture Providing Power and Performance Flexibility. In *ISLPED*, pages 241–243, Aug. 2000.

[3] A.R.Alameldeen and D.A.Wood. Adaptive Cache Compression for High-Performance Processors. In *Proc. of 31st ISCA*, pages 212–223, June 2004.

[4] A.R.Alameldeen and D.A.Wood. Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches. In *Technical Report 1500, Computer Sciences Department, Univ. of Wisconsin-Madison*, Apr. 2004.

[5] A.R.Lebeck and D.A.Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proc. of ISCA*, pages 48–59, June 1995.

[6] B.Goeman, H.Vandierendonck, and K.Bosschere. Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency. In *Proc. of International Symposium on High-Performance Computer Architecture*, pages 207–216, Jan. 2001.

[7] J.Yang, Y.Zhang, and R.Gupta. Frequent Value Compression in Data Caches. In *Proc. of 33rd Micro*, pages 258–265, Dec. 2000.

[8] K.Flautner, N.S.Kim, S.Martin, D.Blaauw, and T.N.Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *Proc. of 29th ISCA*, pages 148–157, May 2002.

[9] K.Nii, H.Makino, Y.Tujihashi, C.Morishima, Y.Hayakawa, H.Nunogami, T.Arakawa, and H.Hamano. A Low Power SRAM using Auto-Backgate-Controlled MT-CMOS. In *Proc. of ISLPED*, pages 293–298, Aug. 1998.

[10] K.Tanaka and A.Matsuda. Static Energy Reduction in Cache Memories Using Data Compression. In *Proc. of TENCON*, pages CD–ROM, Nov. 2006.

[11] K.Tanaka and T.Kawahara. Leakage Energy Reduction in Cache Memories by Data Compression. In *Proc. of International Workshop on Advanced Low Power Systems (ALPS)*, pages 23–30, June 2007.

[12] M.Burtscher and P.Ratanaworabhan. High Throughput Compression of Double-Precision Floating-Point Data. In *Proc. of Data Compression Conference (DCC)*, pages 293–302, Mar. 2007.

[13] M.Kjelso, M.Gooch, and S.Jones. Main Memory Hardware Data Compression. In *Proc. of 22nd Euromicro*, pages 423–430, Sept. 1996.

[14] M.Powell, S.H.Yang, B.Falsafi, K.Roy, and T.N.Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In *Proc. of ISLPED*, pages 90–95, Aug. 2000.

[15] S.C.Woo, M.Ohara, E.Torrie, J.P.Singh, and A.Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of ISCA*, pages 24–36, June 1995.

[16] S.Heo, K.Barr, M.Hampton, and K.Asanovic. Dynamic Fine-Grain Leakage Reduction Using Leakage-Biased Bitlines. In *Proc. of 29th ISCA*, pages 137–147, May 2002.

[17] S.H.Yang, M.D.Powell, B.Falsafi, K.Roy, and T.N.Vijaykumar. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches. In *Proc. of 7th HPCA*, pages 147–158, Jan. 2001.

[18] S.Kaxiras, Z.Hu, and M.Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *Proc. of 28th ISCA*, pages 240–251, June 2001.

[19] SPARC International,Inc. *The SPARC Architecture Manual Version 9*, 1994.

[20] Standard Performance Evaluation Corp. *http://www.spec.org/cpu95/CINT95/*.

[21] T.Pering, T.Burd, and R.Brodersen. Dynamic Voltage Scaling and the Design of a Low-Power Microprocessor System. In *Proc. of Power Driven Microarchitecture Workshop*, pages 107–112, June 1998.

[22] Y.Li, D.Parikh, Y.Zhang, K.Sankaranarayanan, M.Stan, and K.Skadron. State-Preserving vs. Non-State-Preserving Leakage Control in Caches. In *Proc. of DATE'04*, pages Vol1, 22–27, Feb. 2004.

[23] Y.Meng, T.Sherwood, and R.Kastner. On the Limits of Leakage Power Reduction in Caches. In *Proc. of 11th HPCA*, pages 154–165, Feb. 2005.

[24] Y.Sazeides and J.E.Smith. The Predictability of Data Values. In *Proc. of International Symposium on Microarchitecture*, pages 248–258, Dec. 1997.

[25] Z.Wang, K.S.McKinley, and A.L.Rosenberg. Improving Replacement Decisions in Set-Associative Caches. In *Technical Report, University of Massachusetts, UM-CS-2001-002*, Mar. 2001.