Variation-Aware Software Techniques for Cache Leakage Reduction using Value-Dependence of SRAM Leakage due to Within-Die Process Variation

Goudarzi, Maziar
Kyushu University

Ishihara, Tohru
Kyushu University

Noori, Hamid
Kyushu University

出版情報：Lecture Notes in Computer Science. 4917, pp.224-239, 2008-01-18. Springer
バージョン：

Variation-Aware Software Techniques for Cache Leakage Reduction using Value-Dependence of SRAM Leakage due to Within-Die Process Variation

Maziar Goudarzi, Tohru Ishihara, Hamid Noori

Kyushu Univesity, Fukuoka, Japan
{goudarzi, ishihara}@slrc.kyushu-u.ac.jp, noori@c.csce.kyushu-u.ac.jp

Abstract. We observe that the same SRAM cell leaks differently, under within-die process variations, when storing 0 and 1; this difference can be up to 3 orders of magnitude (averaging 57%) at 60mv variation of threshold voltage (V_{th}). Thus, leakage can be reduced if most often the values with less leakage are stored in the cache SRAM cells. We show applicability of this proposal by presenting three binary-optimization and software-level techniques for reducing instruction cache leakage: we *(i)* reorder instructions within basic-blocks so as to match up the instructions with the less-leaky state of their corresponding cache cells, *(ii)* statically apply register-renaming with the same aim, and *(iii)* at boot time, initialize unused cache-lines to their corresponding less-leaky values. Experimental results show up to 54%, averaging 37%, leakage energy reduction at 60mv variation in V_{th}, and show that with technology scaling, this saving can reach up to 84% at 100mv V_{th} variation. Since our techniques are one-off and do not affect instruction cache hit ratio, this reduction is provided with only a negligible penalty, in rare cases, in the data cache.

Keywords: Leakage power, power reduction, cache memory, process variation.

1 Introduction

Cache memories, as the largest component of today's processor-based chips (e.g. 70% of StrongARM [1]) are among the main sources of power dissipation in such chips. In nanometer SRAM cells, most of the power is dissipated as leakage [2] due to lower threshold-voltage (V_{th}) of transistors and higher V_{th} variation caused by random dopant fluctuations (RDF) [3] when approaching atomic sizes. This inherent variation impacts stability, power and speed of the SRAM cells. Several techniques exist that reduce cache leakage power at various levels [4]-[11], but none of them takes advantage of a new opportunity offered by this increasing variation itself: *the subthreshold leakage current (I_{off}) of a SRAM cell depends on the value stored in it and this difference in leakage increases with technology scaling*. When transistor channel length approaches atomic sizes, process variation due to random placement of dopant atoms increases the variation in V_{th} of same-sized transistors even within the same die [13]. This is an unavoidable physical effect which is even more pronounced

in SRAM cells as area-constrained devices that are typically designed with minimum transistor sizes. Higher V_{th}-variation translates to much higher I_{off}-variation ($I_{off} \propto exp(-v_{th}/(s/\ln(10)))$ where s is the subthreshold swing [13]) even in the transistors of a single SRAM cell. Since some of these transistors leak when storing a 1 and others when storing a 0, cell leakage differs in the two states. Thus cache leakage can be reduced if the values stored in it can be better matched with the characteristics of their corresponding cache cells; i.e., if most of the time a 0 is stored in a cache cell that leaks less when storing a 0, and vice versa. To the best of our knowledge, no previous work has observed this saving opportunity. Monte Carlo simulations in Section 3 show that theoretically 70% leakage saving (comparing full match to the full mismatch) would be available in a technology node with 60mv standard deviation of within-die V_{th} variation.

In this paper, we *(i)* reschedule instructions inside each basic-block (BB) of a given application to let them better match their corresponding cache cells, *(ii)* at the same time, we use register-renaming to further improve the match between the instructions and their cache cells, and *(iii)* the least-leaky values are stored in the cache-lines that won't be used by the embedded application. In total, these techniques result in up to 54.18% leakage reduction (36.96% on average) on our set of benchmarks, with only a negligible penalty in the data-cache caused by the instruction-reordering since techniques *(i)* and *(ii)* are applied offline and *(iii)* is only applied once at the processor boot time. Furthermore, it is important to note that this technique reduces leakage in the active- as well as standby-mode of system operation (even when the memory cells are being accessed) and that it is orthogonal to current circuit/device-level techniques.

2 Related Works

Leakage in CMOS circuits can be reduced by power gating [4], source-biasing [2], reverse- and forward-body-biasing [5][6] and multiple or dynamic V_{th} control [7]. For cache memories, selective turn-off [8][9] and dual-supply drowsy caches [10] disable or put into low-power drowsy mode those parts of the cache that are not likely to be accessed again. All these techniques, however, need circuit/device-level modification of the SRAM design while our proposal is a software technique and uses the cache as is. Moreover, none of the above techniques specifically addresses the leakage variation issue (neither variation from cell to cell, nor the difference between storing 0 and 1) caused by within-die process variation. We do that and we work at system-level such that our technique is orthogonal to them. Furthermore, all previous works focus on leakage power reduction when the SRAM cell is not likely to be in use, but our above *(i)* and *(ii)* techniques save power even when the cell is actively in use.

The leakage-variation among various cache-ways in a set-associative cache is used in [11] to reduce cache leakage by disabling the most-leaky cache ways. Our techniques, in contrast, do not disable any part of the cache and use it at its full capacity, and hence, do not incur any performance penalty due to reduced cache size. Moreover, our techniques are applicable to direct-map caches as well.

In logic circuits, value-dependence of leakage power has been identified and used in [12] to set the input vector to its leakage-minimizing value when entering standby

mode. We show this value-dependence exists, with increasing significance, in nano-scale SRAM cells and can benefit power saving even out of standby time.

Register-renaming is a well-known technique that is often used in high-performance computing to eliminate false dependence among instructions that otherwise could not have been executed in parallel. It is usually applied dynamically at runtime, but we apply it statically to avoid runtime overhead. To the best of our knowledge, register-renaming has not been used in the past for power reduction.

Cache-initialization, normally done at processor reset, is traditionally limited to resetting all *valid*-bites to indicate emptiness of the entire cache. We extend this initialization to store less-leaky values in all those cache-lines that won't be used by the embedded application. This is similar to cache-decay [9] in addressing leakage power dissipated by unused cache-lines, but our technique does not require circuit-level modification of the cache design that has prevented cache-decay from widespread adoption.

3 Motivation and Our Approach

Leakage is increasing in nanometer-scale technologies, especially in cache memories which comprise the largest part of processor-based embedded systems. Fig. 1 shows the breakdown of energy consumption of the 8KB instruction-cache of M32R embedded processor [13] running MPEG2 application. The figure clearly shows that although dynamic energy decreases with every technology node, the static (leakage) energy increases such that, unlike in micrometer technologies, total energy of the cache increases with the shrinking feature sizes. Thus it is increasingly more important to address leakage reduction in cache memories in nanometer technologies.

We focus on I_{off} as the primary contributor to leakage in nanometer caches [13]. Fig. 2 shows a 6-transistor SRAM cell storing a 1 logic value. Clearly, only M5, M2, and M1 transistors can leak in this state while the other three may leak only when the cell stores a 0 (note that bit-lines are precharged to supply voltage, V_{DD}). Process variation, especially in such minimum-geometry devices, causes each transistor to

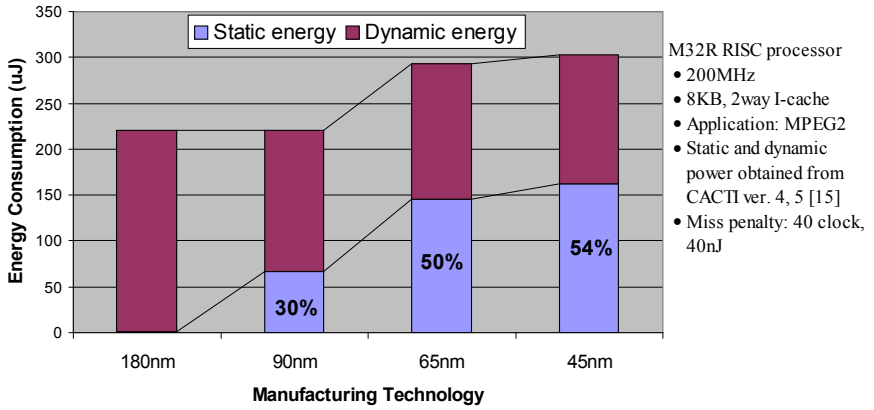

Fig. 1. Cache energy consumption in various technology nodes.

have a different V_{th} and consequently different I_{off} value, finally resulting in different subthreshold leakage currents when storing 1 and 0. Since the target V_{th} is in general reduced in finer technologies, in order to keep the circuit performance when scaling dimensions and V_{DD}, the I_{off} value is exponentially increased, and consequently, the above leakage difference is no longer negligible.

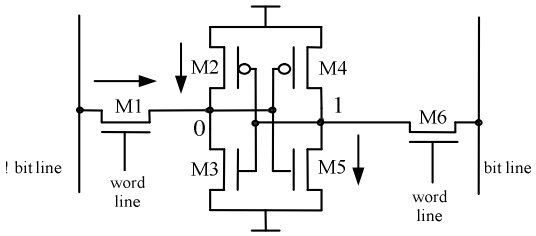

Fig. 2. A 6-transistor SRAM cell storing a logic 1. Arrows show leakage paths.

To quantify this effect, we used Monte Carlo simulation to model several similar caches and for each one computed maximum leakage difference once in each cell and once more in the entire cache. Notations and formulas are:

- **leak0 (leak1):** leakage power of the cell when storing 0 (1).
- **low** = min(leak0, leak1)
- **high** = max(leak0, leak1)

$$per-cell \quad saving = |leak0 - leak1| / high \tag{1}$$

$$Upper \quad bound \quad of \quad per-cache \quad saving = (\sum_{all \ \ cells} high - \sum_{all \ \ cells} low) \Big/ \sum_{all \ \ cells} high \tag{2}$$

Eq. 1 gives leakage difference between less-leaky and more-leaky states of a single cell, while Eq. 2 gives, in the entire cache, the difference between the worst case (all cells storing more-leaky values) and the best case (all cells storing less-leaky values).

Variation in transistors V_{th} results from die-to-die (inter-die) as well as within-die (intra-die) variation. We considered both in these experiments. Inter-die variation, which results in varying average V_{th} among different chips, is generally modeled by Gaussian distribution [16] while for intra-die variation, which results in different V_{th} values for different transistors even within the same chip and the same SRAM cell, independent Gaussian variables are used to define V_{th} of each transistor of the SRAM cell [17][18]. We used the same techniques to simulate manufacturing of 1000 16KB caches (direct-map, 512-set, 32-byte lines, 23 bits per tag) and obtained the maximum theoretical per-cell and per-chip savings given in Fig. 3 for $\sigma_{Vth\text{-}intra}$ (i.e. standard-deviation of intra-die V_{th} variations) varying from 10 to 100mv. We assumed each cache is within a separate die and used a single $\sigma_{Vth\text{-}inter}$=20mv for all dies. The mean value of V_{th} was set to 320mv but our experiments with other values showed that the diagrams are independent of the V_{th} mean value; i.e., although the absolute value of the saving does certainly change with different V_{th} averages (and indeed increases with lower V_{th} in finer technologies), but the maximum *saving ratio* (Eq. 1 and 2) remains invariant for a given $\sigma_{Vth\text{-}intra}$, but the *absolute value* of the saved power increases with decreasing V_{th}. This makes sense since this saving opportunity is enabled by the V_{th} *variation*, not the V_{th} *average value*.

Since $\sigma_{Vth-intra} \propto 1/\sqrt{L \times W}$ [3], where L and W are effective channel length and width respectively, the V_{th} variation is only to increase with technology scaling, and as Fig. 3 shows, this increases the significance of value-to-cell matching. In 0.13µm process, empirical study [19] reports $\sigma_{Vth\text{-}intra}$=22.1mv for W/L=4 which by extrapolation gives $\sigma_{vth\text{-}intra}$>60mv in 90nm for minimum-geometry transistors; ITRS roadmap also shows similar prospects [20]. (We found no public empirical report on 90nm and 65nm processes, apparently due to sensitiveness and confidentiality.) Thus we present results at various $\sigma_{vth\text{-}intra}$ values, but consider 60mv as a typical case. Note that even if the extrapolation is not accurate for 90nm process, $\sigma_{vth\text{-}intra}$=60 finally happens at a finer technology node due to $\sigma_{Vth-intra} \propto 1/\sqrt{L \times W}$. Fig. 3 shows that maximum theoretical saving using this phenomenon at 60mv variation can be as high as 70%.

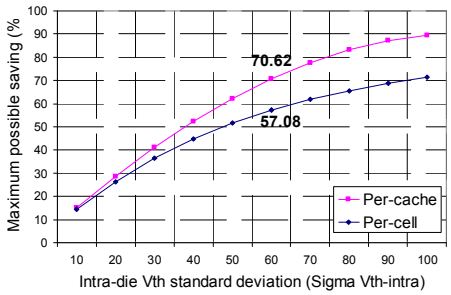

Fig. 3. Leakage saving opportunity increases with V_{th}-variation.

3.1 Our Approach

We propose three techniques applicable to instruction-caches: rescheduling instructions within basic-blocks, static register-renaming, and initializing unused cache-lines. We first illustrate them by examples before formal formulation.

Illustrative Example 1: Intra-BB Instructions Rescheduling. Fig. 4 illustrates our approach applied to a small basic block (shown at left in Fig. 4) consisting of three 8-bit instructions against a 512-set direct-mapped cache with 8-bit line size. The arrow at the right of instruction-memory box represents dependence of instruction 2 to instruction 1. For simplicity, we assume *(i)* all the 3 instructions spend the same amount of time in the cache, and *(ii)* the leakage-saving (i.e., |*leak0-leak1*|) is the same for all bits of the 3 cache lines. An SRAM cell is called *1-friendly* (*0-friendly*) or equivalently *prefers 1* (*prefers 0*), if it leaks less power when storing a 1 (a 0). This *leakage-preference* of the cache lines are given in gray in the middle of Fig. 4; for example, the leftmost bit of cache line number 490 prefers 0 (is 0-friendly) while its rightmost bit prefers 1 (is 1-friendly). The *Matching table* in Fig. 4 shows the number of matched bits for each *(instruction, cache-line)* pair. Due to instruction dependencies, only three schedules are valid in this example: 1-2-3 (i.e., the original one), 1-3-2, and 3-1-2 with respectively 3+1+3, 3+3+7, and 1+7+7 number of matched bits (see the *Matching table* in Fig. 4). We propose to reschedule basic-blocks, subject to dependencies among the instructions, so as to match up the

instructions with the leakage-preference of cache lines. Thus, the best schedule, shown at right in Fig. 4, is 3-1-2 which improves leakage of this basic-block by 47% (from 24-7 mismatches to 24-15 ones).

Obviously, the two simplifying assumptions in the above example do not hold in general. Potential leakage-saving differs from cell to cell, and also the amount of time spent in the cache differs from instruction to instruction even in the same BB. We consider and analyze these factors in our formulation and experiments.

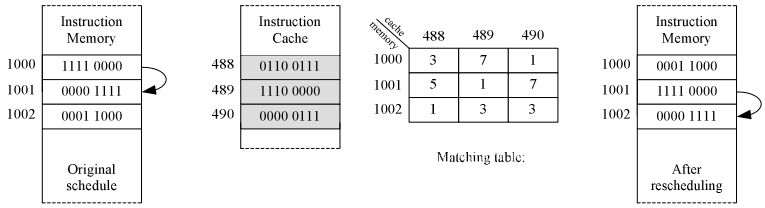

Fig. 4. An example illustrating instruction-rescheduling

Illustrative Example 2: Register-Renaming. Assume that the two right-most bits of each instruction in Fig. 5 represent a source register and the two left-most bits give the other source which is also the destination register. Fig. 5 depicts a simple example of register-renaming on the cache in the middle of the figure; for presentational purposes, we ignore instruction rescheduling here and merely apply register-renaming although our algorithm applies both at the same time. When applying merely register-renaming to these instructions, R0 can be renamed to R3 in the first two instructions (note that this implies similar renaming in all predecessor, and successor, instructions that in various control-flow scenarios produce, or consume, the value in R0; this is not shown in the figure). Similarly, original R3 in the same two instructions can be equally-well renamed to either R1 or R0; it is renamed to R1 in Fig. 5. For the third instruction, there is no better choice since source and destination registers are the same while their corresponding cache cells have opposite preferences (renaming to R1, which results in only the same leakage-preference-matching, is inappropriate since the instruction would then conflict with the now-renamed first instruction).

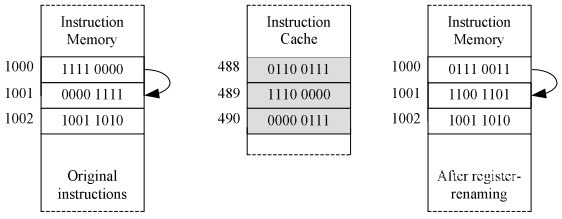

Fig. 5. An example illustrating register-renaming

Illustrative Example 3: Initializing Unused Cache-Lines. Depending on the cache size and the application, some parts of the instruction cache may never be used during application execution. Fig. 6 shows the histogram of *cache-fill* operations in the 8KB instruction cache of M32R processor [13] (a 32-bit RISC processor) when executing FFT application. 69 out of the 512 16-byte cache-lines are never used in this case. We propose to initialize such unused cache-lines with values that best match

the leakage-preference of their SRAM cells. Many processors today are equipped with cache-management instructions (e.g. ARM10 family [21] and NEC V830R processor [22]) that can load arbitrary values to every cache location. Using these instructions, the unused cache-lines can be initialized at boot time to effectively reduce their leakage-power during the entire application execution. For instance, if in Fig. 5 cache-line number 490 were not to be used at all by the application, it would be initialized to 00000111 to fully match its leakage-preference. A minimum power-ON duration is required to break even the dynamic energy for cache initialization and the leakage energy saved. We consider this in our problem formulation and experiments.

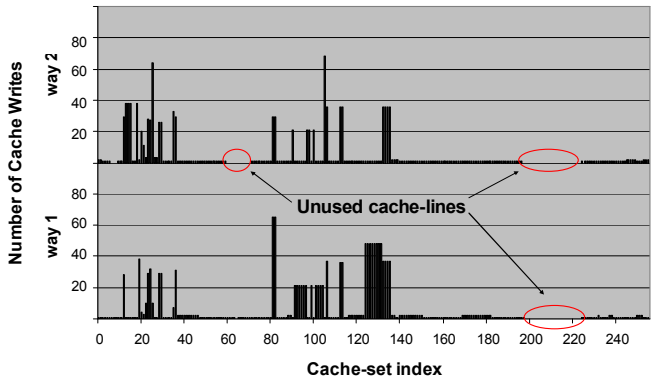

Fig. 6. Unused cache-lines for FFT application (8KB 2-way cache with 16-byte cache-lines).

Leakage-Preference Detection. This can be incorporated in the manufacturing test procedure that is applied to each chip after fabrication. Usually walking-1 and walking-0 test sequences are applied to memory devices [23] to test them for stuck-at and bridging faults. Leakage current can be measured at each step of this test procedure (similar to delta-IDDQ testing [24]) to determine the leakage-preference of cells. This can even be done in-house since commodity ammeters can easily measure down to 0.1fA [25] while the nominal leakage of a minimum geometry transistor is 345pA in 90nm process available to us. For some cells, this difference may be negligible, but one can detect more important cells that cause larger leakage differences. Test time for an 8KB cache, assuming 1MHz current measurements, would be 128ms (measuring *leak0* and *leak1* for each SRAM cell).

4 Problem Formulation

We formulate the problem using the following notation:

- N_s, N_w: The number of sets and ways of the cache.
- N_{BB}: The number of basic-blocks in the given application.
- $N_i(bb)$: The number of instructions in basic-block no. *bb*.
- $L(i, bb, w)$: Leakage power dissipated by the corresponding word of the cache line at way *w* of the cache when instruction number *i* of basic-block number *bb* is stored

there. Note that the cache set corresponding to the instruction is fixed, but the cache way may differ over time.

- ***T(i, bb, w)*** or *cache-residence time*: The amount of time that instruction number *i* of basic-block number *bb* remains in way *w* of the corresponding cache set.
- E_{BB}**:** Total leakage energy of instruction cache due to basic-block instructions:

$$E_{BB} = \sum_{bb=1}^{N_{BB}} \sum_{i=1}^{N_i(bb)} \sum_{w=1}^{N_w} L(i,bb,w) \times T(i,bb,w) \quad (3)$$

  Each term in this summation gives the leakage energy dissipated by instruction *i* of basic-block *bb* at way *w* of cache.
- T_{viable}**:** The minimum amount of time that the embedded system should remain ON so that the cache-initialization technique would be *viable* (i.e., would save energy).

The problem is formally defined as "*For a given application and cache organization (i.e. for given N_s, N_w, N_{BB}, and $N_i(bb)$ vector), (i) minimize E_{BB}, and (ii) find T_{viable}.*"

Algorithms. We use a list-scheduling algorithm for problem *(i)* above to achieve high efficiency; register-renaming is performed at each iteration of the algorithm:

```
Algorithm 1: ListScheduling(G)
Inputs: (G: control-data-flow Graph of application)
Output: (S: obtained Schedule for instructions of the application)
1  S = empty-list;
2  foreach basic-block do
3    BA = Base-Address of the basic-block;
4    L  = Length of the basic-block;
5    for addr=BA to BA + L do
6      lowestLeakage = +INFINITY;     bestChoice = 0
7      for each i in ready-list(G, BA) do
8        (ni, src, dst, flag) = applyRegRenaming(i, addr);
9        leak = get_instruction_leakage(ni, addr)
10       if leak < lowestLeakage then
11         lowestLeakage = leak;    bestChoice = ni;
12         bestRegs = (src, dst, flag);
13       endif
14     endfor
15     propagateRegRenamings( G, bestRegs );
16     S = S + {bestChoice};
17     Mark {bestChoice} as scheduled in G to update ready-list(G, BA);
18   endfor
19 endfor
20 return S
```

The algorithm sequentially processes each basic-block in the application binary and stores the new schedule with the new register-names in *S* as output. It needs the control-data-flow graph of the application for register-renaming so as to figure out live registers and the instructions that produce and consume them. For each basic-block, all *ready* instructions (i.e. those with all their predecessors already scheduled), represented by `ready-list(G, BA)` in line 7, are tried and the one with the least leakage is chosen (lines 9-13) and appended to the schedule (lines 16, 17); line 9 computes the leakage corresponding to the instruction by giving the innermost summation of Eq. 3. Register-renaming is also applied to each *ready*-instruction (line 8) and if chosen as the best, its corresponding new register-names are propagated to all predecessor and successor instructions (line 15); these procedures are given below:

```
Procedure: applyRegRenaming(i, addr)
Inputs: (i: the instruction binary to manipulate),
       (addr: the address of i in memory)
Outputs:(new_i: instruction after register-renaming),
       (src, dst: new source and destination regs),
       (flag: shows which regs were finally renamed)
1  src = first-source-register of i;
3  dst = destination-register of i;
3  flag = 0;
4  if src not affixed
5     src = get_best_src1_choice(i, addr); flag+=1;
6  if dst not affixed
7     dst = get_best_dest_choice(i, addr); flag+=2;
8  new_i = i with src, and dst;
9  return new_i, src, dst, flag;
```

This procedure checks the two source and destination registers (in M32R, the destination register and the second source register are the same) and if each of them is not *affixed,* tries to rename it to the best available choice. A source or destination register is *affixed* if due to an already-applied register-renaming it is previously determined and should be kept unchanged; the below procedure pseudo-code shows this. In some cases, it may be beneficial to reconsider renaming since the leakage reduction by the new register-renaming may outweigh the loss in previously renamed instructions; we did not consider this for simplicity and efficiency.

```
Procedure: propagateRegRenamings(G, i, src, dst, flag)
Inputs: (G: control data flow Graph of application),
       (i: instruction before register-renaming),
       (src, dst: new source and destination regs)
       (flag: shows which regs are renamed)
1 org_src = first-source-register of i;
2 org_dst = destination-register of i;
3 if (flag & 1)
4   rename org_src to src, and mark it affixed, in all predecessors and
    successors of i in G
5 if (flag & 2)
6   rename org_dst to dst, and mark it affixed, in all predecessors and
    successors of i in G
```

The algorithm has a time complexity of $O(m.n^2)$ and memory usage of $O(m.n)$ where m and n respectively represents the number of basic-blocks in the application and the number of instructions in the basic-block. Note that the algorithm correctly handles set-associative caches since the innermost summation in Eq. 3 considers individual leakages of each cache-way. The algorithm does not necessarily give the absolute best schedule neither the best register-names, but comparing its experimental results to that of exhaustive search in the feasible cases, which is still prohibitively time-consuming, shows the results are no more than 12% less optimal than the absolute best schedule.

5 Experimental Results

We used benchmarks from MiBench, MediaBench, and also Linux `compress` (Table 1) in our experiments. Monte Carlo simulation was used to model within-die

process variation; independent Gaussian random values for V_{th} of each transistor of the cache were generated with 320mv as the mean and 60mv as the standard deviation. To consider the randomness of process variations, we simulated 1000 chips and ran our algorithm on all of them. Die-to-die variations do not change the saving percentage (see Section 3) and were not modeled in these experiments. Benchmarks were compiled with no compiler optimization option and were simulated using M32R instruction-set simulator to obtain cache-residence and cache-line usage statistics for 1 million instructions (FIR ran up to completion).

Table 1. Benchmarks specifications

Benchmark	No of basic-blocks	Basic-block size (#instr.) Average	Largest
MPEG2 encoder ver. 1.2	16000	5.36	596
FFT	12858	4.83	75
JPEG encoder ver. 6b	11720	5.68	248
Compress ver. 4.1	9586	5.11	718
FIR	450	7.59	57
DCT	508	4.96	64

Fig. 7 shows the average leakage powers (corresponding to an industrial 90nm process) before and after applying our leakage-saving techniques, obtained over 1000 8KB direct-mapped caches with 16-byte cache-line size. Each bar is composed of two parts: the leakage power dissipated by the cache-lines that were used during application execution, and those that were never used. Our rescheduling algorithm reduces the former, while the cache-initialization technique suppresses the latter.

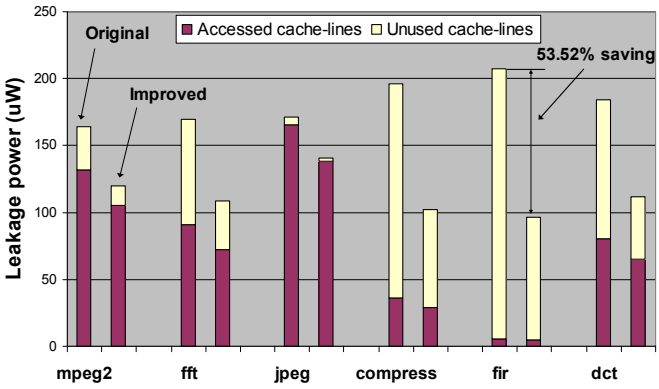

Fig. 7. Average leakage power on 1000 8KB direct-map caches.

Table 2 gives the individual average and maximum savings obtained by each technique over the above 1000 chips; note that the values in *rescheduling* and *initializing* columns respectively correspond to the leakage savings *only in used* and *only in unused* cache-lines. The rescheduling and register-renaming technique saves up to 31.31% of power for FIR while savings by the cache-initialization technique reaches 58.36% for JPEG benchmark. Average saving obtained by cache-initialization

is 54.51% for all benchmarks since we assumed that before initialization, SRAM cells in the unused cache-lines randomly contain 0 or 1 values.

Table 2. Average and maximum leakage savings by our techniques.

Benchmark	Average saving (%)			Maximum saving (%)		
	rescheduling	initializing	Together	rescheduling	initializing	Together
MPEG2	20.10	54.51	26.78	21.67	56.16	28.25
FFT	20.50	54.51	36.28	22.43	55.7	37.36
JPEG	16.70	54.51	17.96	17.91	58.36	19.26
Compress	19.74	54.51	48.15	23.95	55.32	48.92
FIR	20.04	54.51	53.52	31.31	55.19	54.18
DCT	19.31	54.51	39.09	21.49	55.61	40.13

Different cache-sizes result in different number of unused cache-lines, and hence, affect saving results. Fig. 8 depicts the savings for 16KB, 8KB, and 4KB direct-map caches with 16-byte line-size. As the figure shows, in general, the leakage saving reduces in smaller caches proportional to the reduction in the number of unused cache-lines. This, however, does not affect the results of the rescheduling and register-renaming techniques, and hence, increases their share in total leakage-reduction (see Fig. 8). Consequently, when finally all cache-lines are used by the application in a small cache, the leakage reduction reaches its minimum (as in MPEG2 and JPEG cases in Fig. 8), which is equal to the saving achieved by the rescheduling and register-renaming technique alone (compare MPEG2 and JPEG in Fig. 8 to their corresponding rows in Table 2 under *rescheduling* column).

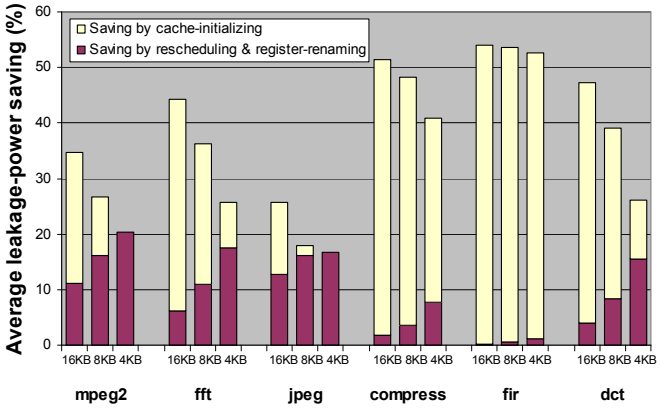

Fig. 8. Effect of cache-size on average leakage-saving results.

Set-associative caches take better advantage of the available cache-lines and reduce the number of unused ones. Fig. 9 shows the leakage savings in an 8KB cache when the number of ways changes from 1 (direct-map) to 8. The leakage-saving by cache-initialization reduces in caches with higher associativity, and finally total saving reduces to that obtained by the rescheduling and register-renaming technique as is again the case for MPEG2 and JPEG in Fig. 9.

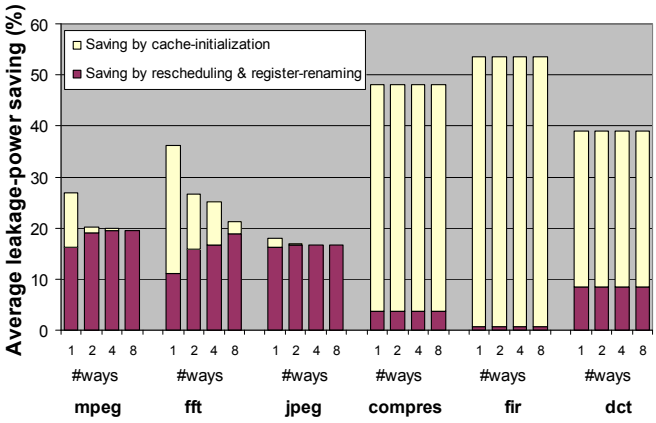

Fig. 9. Effect of set-associative caches on total leakage saving.

Furthermore, in set-associative caches, the location of each instruction in the cache cannot be precisely determined since there are multiple cache-lines in the cache-set that corresponds to the address of the instruction. This uncertainty is expected to decrease the saving results of the rescheduling algorithm, however, our cache simulator gives separate per-way residence-times for each instruction so as to direct the matching process toward the cache-ways with higher probability of hosting the instruction. Saving results of Algorithm 1 are given in Fig. 10; as in Fig. 9, cache size and line-size are respectively fixed at 8KB and 16-bytes while the number of cache-ways varies from 1 to 8. The figure confirms that the number of cache-ways only slightly affects the results due to the above-mentioned technique for directing the algorithm towards matching the instruction against the more likely used cache-way. Some marginal increases are seen in Fig. 10 for MPEG2, Compress, and FIR at higher cache associativity; these are random effects that happen since the algorithm does not give the absolute optimal schedule and also the cache-lines that correspond to each instruction changes when changing the number of cache-ways.

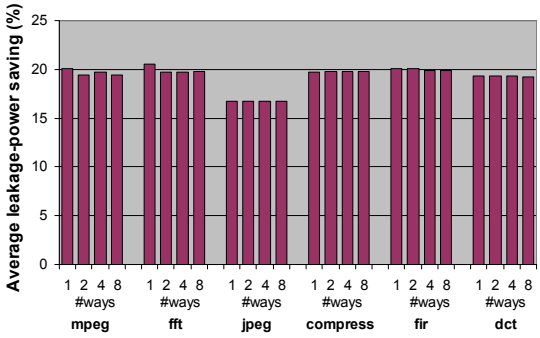

Fig. 10. Effect of set-associative caches on rescheduling algorithm.

Execution-times of the rescheduling algorithm for the above caches are given in Table 3; values are measured on a Xeon 3.8GHz processor with 3.5GB memory. The

execution time increases with the number of cache-ways, since more calculations are necessary, but it remains reasonably low to be practical.

Table 3. Algorithm execution-time (in seconds).

Benchmark	Cache configuration (sets×ways×line_size)			
	512×1×16	256×2×16	128×4×16	64×8×16
MPEG2	0.15	0.33	0.55	1.04
FFT	0.08	0.19	0.31	0.60
JPEG	0.18	0.40	0.70	1.35
Compress	0.05	0.10	0.15	0.26
FIR	0.01	0.01	0.02	0.04
DCT	0.03	0.06	0.12	0.23
Average	0.08	0.18	0.31	0.59

Fig. 3 suggests that the achievable energy saving rises with the increase in V_{th} variation caused by technology scaling. We repeated the experiments for 8KB, 512-set direct-map cache with $\sigma_{Vth\text{-}intra}$ varying from 20 to 100mv (with mean-V_{th}=320mv in all cases). Fig. 11 shows the trend in saving results which confirm the increasing significance of the approach in future technologies where random within-die V_{th} variation is expected to increase [20] due to random dopant fluctuation which is rising when further approaching atomic sizes in nanometer processes.

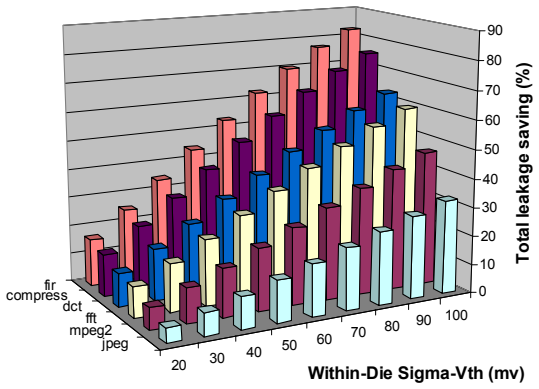

Fig. 11. Saving improvement with technology scaling.

Costs of Intra-BB Rescheduling and Register-Renaming. Register-renaming imposes absolutely no penalty. Instruction-rescheduling has no impact on instruction-cache but may in rare cases marginally affect data-cache: since the order and address of basic-blocks do not change, instruction cache performance is kept intact. In data cache, however, reordering of instructions may change the sequence of accesses to data elements, and hence, may change cache behavior. If a miss-causing instruction is moved, the hit-ratio is kept, but residence-times (and hence leakage power) of the evicted and fetched data items change negligibly. In addition, if two instructions that access cache-conflicting data elements change their relative order, the cache hit-ratio changes if the originally-first one was to be a hit. This case may also change the data

that finally remains in the cache after basic-block execution, and hence, potentially affects leakage power of the data cache. It is, however, very unlikely to happen when noting that due to locality of reference, two conflicting data accesses are unlikely to follow closely in time (and in a single BB). In our experiments data cache power and performance varied no more than 1%.

Cost of Cache Initialization. As explained in Section 3, the cache-initialization technique consumes some dynamic power to execute the cache-management instructions before it can save leakage power. Our implementation of M32R processor with two separate 8KB instruction and data caches on a 0.18μ process technology consumes 200mW at 50MHz clock frequency. This gives, on average, 4nJ per clock cycle or pessimistically 20nJ per instruction in the 5-stage pipelined M32R processor. Assuming all 512 cache-lines of the instruction cache are to be initialized, 10.24μJ is consumed for cache-initialization. T_{viable} can now be calculated using the power-saving values obtained by cache-initialization (Fig. 7). Results are given in Table 4 which confirm that most often a small fraction of a second is enough to make the initialization technique viable. Even for the worst benchmark, JPEG, a few seconds is enough. Assumptions in the estimations were pessimistic to not overestimate benefits: *(i)* processor implementation in a finer technology (e.g. 90nm) would consume less dynamic power, *(ii)* more than one instruction is often in the processor pipeline so average power per instruction would be less than 20nJ, *(iii)* not all cache-lines need to be initialized (e.g. for JPEG, only 14 cache-lines remain unused and should be initialized). Thus, values in Table 4 should be considered as upper bounds for T_{viable}.

Table 4. Estimated T_{viable} upper bounds for different applications.

	MPEG2	FFT	JPEG	Compress	FIR	DCT
T_{viable} (s)	0.590	0.238	3.281	0.117	0.093	0.182

6 Conclusion

Our contributions here are *(i)* observing and analyzing a new opportunity for reducing cache leakage in nanometer technologies enabled by the reducing V_{th} and the increasing V_{th}-variation in such processes, and *(ii)* presenting first techniques that take advantage of this opportunity and reduce leakage up to 54.18% (36.96% on average) with negligible impact on system performance. It is important to note that our techniques (*i)* become more effective with technology scaling, *(ii)* reduce leakage also in the normal mode of system operation (in addition to standby mode) even when the cache-lines are actively in use, and *(iii)* are orthogonal to other techniques for leakage reduction such as body- and source-biasing. As future work, we are investigating techniques similar to garbage-collection so as to invalidate the cache-lines that won't soon have a hit and to store the less-leaky values in them.

Acknowledgments. This work is supported by VDEC, The University of Tokyo with collaboration of STARC, Panasonic, NEC Electronics, Renesas Technology, and Toshiba. This work is also supported by CREST project of Japan Science and Technology Corporation (JST). We are grateful for their support.

References

1. Moshnyaga, V.G., Inoue, K., Low-Power Cache Design. In: Piguet, C. (eds.) Low-Power Electronics Design. CRC Press (2005)
2. Roy, K., et al., Leakage Current Mechanisms and Leakage Reduction Techniques in Deep-Submicron CMOS Circuits. In Proc. IEEE (2003)
3. Taur, Y., Ning, T.H., Fundamentals of Modern VLSI Devices. Cambridge University Press (1998)
4. Kao, J.T., Chandrakasan, A.P., Dual-Threshold Voltage Techniques for Low-Power Digital Circuits. IEEE J. of Solid State Circuits 35, 1009-1018 (2000)
5. Fallah, F., Pedram, M., Circuit and System Level Power Management. In Pedram, M., Rabaey, J., (eds.) Power Aware Design Methodologies. Kluwer, pp. 373-412 (2002)
6. De, V., Borkar, S., Low Power and High Performance Design Challenge in Future Technologies. In: Great Lake Symposium on VLSI (2000)
7. Kuroda, T., Fujita, T., Hatori, F., Sakurai, T., Variable Threshold-Voltage CMOS Technology. IEICE Trans. on Fund. of Elec., Comm. and Comp. Sci. E83-C (2000)
8. Powell, M.D., et al., Gated-Vdd: a Circuit Technique to Reduce Leakage in Cache Memories," In: Int'l Symp. Low Power Electronics and Design (2000)
9. Kaxiras, S., Hu, Z., Martonosi, M., Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In: Int'l Symp. on Computer Architecture, 240-251 (2001)
10. Flautner, K., et al., Drowsy Caches: Simple Techniques for Reducing Leakage Power. In: Int'l Symp. on Computer Architecture (2002)
11. Meng, K., Joseph, R., Process Variation Aware Cache Leakage Management. In: Int'l Symp. on Low Power Electronics and Design (2006)
12. Abdollahi, A., Fallah, F., Pedram, M., Leakage Current Reduction in CMOS VLSI Circuits by Input Vector Control. IEEE Trans. VLSI 12(2), 140-154 (2004)
13. Clark, L., De, V., Techniques for Power and Process Variation Minimization. In: Piguet, C. (eds.) Low-Power Electronics Design. CRC Press (2005)
14. M32R Family 32-bit RISC Microcomputers, http://www.renesas.com
15. CACTI Integrated Cache Access Time, Cycle Time, Area, Leakage, and Dynamic Power Model, HP Labs., http://www.hpl.hp.com/personal/Norman_Jouppi/cacti4.html
16. Agarwal, A., Paul, B.C., Mahmoodi, H., Datta, A., Roy, K., A Process-Tolerant Cache Architecture for Improved Yield in Nanoscale Technologies. IEEE Trans. VLSI 13(1) (2005)
17. Luo, J., Sinha, S., Su, Q., Kawa, J., Chiang, C., An IC Manufacturing Yield Model Considering Intra-Die Variations. In: Design Automation Conference, pp. 749-754 (2006)
18. Agarwal, K., Nassif, S., Statistical Analysis of SRAM Cell Stability. In: Design Automation Conference (2006)
19. Toyoda, E., DFM: Device & Circuit Design Challenges. In: Int'l Forum on Semiconductor Technology (2004)
20. International Technology Roadmap for Semiconductors—Design, 2006 Update, http://www.itrs.net/Links/2006Update/2006UpdateFinal.htm
21. Hill, S., The ARM 10 Family of Embedded Advanced Microprocessor Cores. In: HOT-Chips (2001)
22. Suzuki, K., Arai, T., Kouhei, N., Kuroda, I., V830R/AV: Embedded Multimedia Superscalar RISC Processor. IEEE Micro 18(2), 36-47 (1998)
23. Hamdioui, S., Testing Static Random Access Memories: Defects, Fault Models and Test Patterns, Kluwer (2004)
24. Thibeault, C., On the Comparison of Delta IDDQ and IDDQ Testing. In: VLSI Test Symp., pp. 143-150 (1999)
25. DSM-8104 Ammeter, http://www.nihonkaikeisoku.co.jp/densi/toadkk_zetuenteikou_dsm8104.htm