The CSI Journal on Computer Science & Engineering

The CSI Journal on
Computer Science and Engineering
Vol. 5, No. 2&4 (b), 2007
Pages 16-24
Regular Paper

Process-Variation-Aware Instruction Rescheduling to Reduce Leakage in Nanometer Instruction Caches

Maziar Goudarzi[1,2] Tohru Ishihara[2]

[1]Department of Computer Engineering, Sharif University of Technology, Tehran, Iran
[2]System LSI Research Center, Kyushu University, Fukuoka, Japan

Abstract

Process variation, despite introducing challenges in stability and power of SRAM memories, provides a new opportunity for leakage-power reduction: statistical simulation shows that the same SRAM cell leaks differently when storing 0 and 1; this difference is as high as 57% at 60mv variation of threshold voltage (V_{th}). Thus, leakage can be reduced if values with less leakage can be stored in the cells. We show applicability of this proposal by presenting a first technique for reducing instruction cache leakage: we reorder instructions within basic-blocks so as to match up the instructions with the less-leaky state of their corresponding cache cells. Experimental results show up to 12.51%, averaging 10.73%, leakage energy reduction at 60mv variation in V_{th}, and that this saving increases with technology scaling. Since intra-basic-block rescheduling does not affect instruction cache hit ratio, this reduction is provided with only a negligible penalty, in rare cases, in the data cache.

Keywords: Process Variation, Leakage Power, Instruction Cache, Instruction Rescheduling, Low-Power Design.

1. Introduction

Cache memories, as the largest component of today processor-based chips (e.g. 70% of StrongARM [1]) are among 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. This inherent variation impacts stability, power and speed of the cells. Several techniques exist that reduce cache leakage power at various levels [3-10], but none of them takes advantage of a new opportunity offered by this increasing variation itself: *the subthreshold leakage current (I_{off}) of an 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 [11]. 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}-variatoin ($I_{off} \propto exp(-v_{th}/(s/\ln(10)))$ where s is the subthreshold swing [11]) 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 2.1 show that theoretically 70% leakage saving (comparing the best to the worst case) can be achieved in a technology node with 60mv standard deviation of V_{th}. In this paper, we reschedule instructions inside each basic-block (BB) of a

given application to make them better match their corresponding cache cells. This results in 10.73% leakage reduction on average on a set of benchmarks and direct-map caches, with only a negligible penalty in data-cache, since intra-BB rescheduling does not change instruction cache hit ratio. Furthermore, it is important to note that this technique reduces leakage in the active-mode (when the processor is not idle and is doing useful work) as well as standby-mode (when the processor is idle and processor components can be put in a low-power sleep mode) of system operation and that it is orthogonal to circuit/device-level techniques for leakage reduction.

This paper is an extension of our previous work 0. Here we have added the following major points:

- The technique is significantly extended to address leakage power even during normal mode of system operation where the cache lines are actively being used.
- An OS-based leakage-reduction flow is introduced in order to remove the need to store a different binary object code per chip.

In the rest of this paper, Section 2 reviews related previous work and compares them to our proposal outlined there. Section 3 formally formulates the problem and presents the algorithms. Experimental results are presented and analyzed in Section 4, and finally, Section 5 concludes the paper.

2. Previous Work and Our Approach

Leakage in CMOS circuits can be reduced by power gating [3], source-biasing [2], reverse- and forward-body-biasing [4, 5] and multiple or dynamic V_{th} control [6]. For cache memories, selective turn-off [7, 8] and dual-supply drowsy caches [9] disable or put into low-power drowsy mode those parts of the cache that are not likely to be accessed again. All of these techniques, however, require circuit/device-level modification of the SRAM design while our proposal is a system-level 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 process variation at nano scales. We do that and we work at system-level such that our technique is orthogonal to them. Furthermore, all previous work focus on leakage power when the SRAM cell is not likely to be reused, but our technique saves power even when the cell is actively being used.

In logic circuits, value-dependence of leakage power has been identified and used in [10] to set the input vector to its leakage-minimizing values when entering standby mode. We show that this value-dependence exists, with increasing significance, in nanometer SRAM cells and can benefit power saving even during active mode of system operation.

2.1. Observation

We focus on I_{off} as the primary contributor to leakage in nanometer caches [11]. Figure 1 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 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 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.

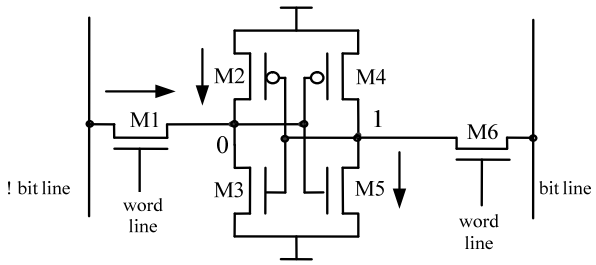

Figure 1. A 6-transistor SRAM cell storing a logic 1

To quantify this effect, we used Monte Carlo simulation to model several similar caches and for each one computed maximum possible leakage saving in each cell as well as in the entire cache. The following notations are used to formally define these savings:

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

The formulas are:

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

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

Eq. 1 gives the 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 of them in these experiments. Inter-die variation, which results in varying average V_{th} among different chips, is generally modeled by Gaussian distribution [13] 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 [14, 15]. 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 possible per-cell and per-chip savings given in Figure 2 for $\sigma_{Vth\text{-}intra}$ 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 400mv 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 $\sigma_{Vth\text{-}intra}$. This makes sense since this saving opportunity is enabled by the V_{th} variation, not the V_{th} average value, but its absolute value increases with decreasing V_{th}.

Note that with the shrinking feature sizes, the V_{th} variation is only to increase and as Figure 2 shows, this increases the significance of value-to-cell matching. Empirical measurement [16] reports $\sigma_{Vth\text{-}intra}$=22.1mv for random-logic (W/L=4) in 0.13µm which by extrapolation (noting $\sigma_{Vth-intra} \propto 1/\sqrt{L \times W}$ where L and W are effective channel length and width respectively) gives $\sigma_{vth\text{-}intra}$>60mv in 90nm process for minimum-size transistors (W/L=1); The International Technology Roadmap for Semiconductors (ITRS roadmap) also shows similar prospects. Thus, maximum possible saving using this phenomenon in 90nm process can be as high as 70% (see Figure 2).

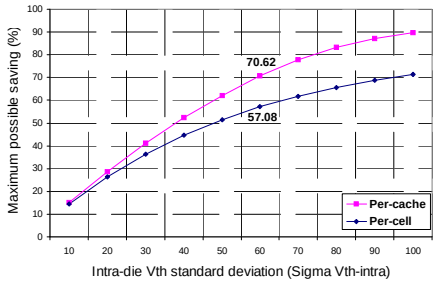

Figure 2. Increasing leakage saving opportunity with V_{th}-variation, reported per SRAM cell and per 16KB cache

2.2. Our Approach

Several techniques can be applied to take advantage of the above observation. In this paper, we focus on instruction cache and on rescheduling instructions within basic-blocks. This does not affect hit-ratio of instruction cache, only marginally impacts data cache, and furthermore, it can be easily applied to binary object code of applications without recompilation. We first illustrate it by an example, and then, formulate the problem and present the algorithms.

Illustrative example. Figure 3 illustrates our approach applied to a small basic block (shown at left in Figure 3) 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 in Figure 3 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 Figure 3; 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 table at the bottom of Figure 3 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 Figure 3). 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 Figure 3, is 3-1-2 which improves leakage of this basic-block by 47% (from 24-7 mismatches to 24-15 ones).

Figure 3. Illustrative example

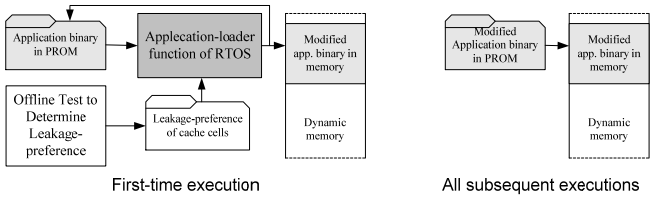

Figure 4. RTOS-based alternative for leakage-reduction flow

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.

Leakage reduction flow. Leakage-aware rescheduling must be done before application execution on the target processor and cache. In order to keep the same binary on all chip instances, we propose to let the Real-time Operating System (RTOS) do this while loading the application binary to the system memory. An RTOS is common in many modern embedded systems and one of its typical tasks is to load applications into dynamic memory and initialize them for execution.

Figure 4 shows the RTOS-based leakage reduction flow. At very first execution, an offline test procedure (the lower left box in Figure 4—described below) detects the leakage-

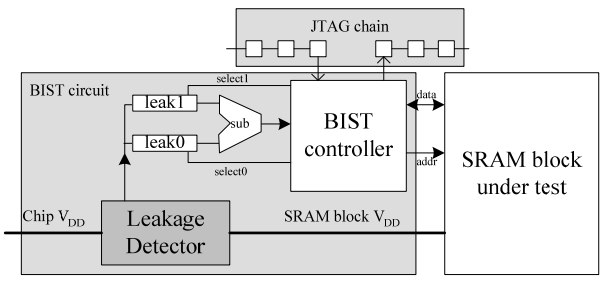

Figure 5. BIST circuitry for leakage-preference detection

preference of all cache lines. This information is used by the rescheduling algorithm, in the dark grey box, to reorder the instructions of the application binary before putting them in the dynamic memory and executing. The modified binary is also written back to the PROM so that subsequent executions of the application directly load the program to memory without modification (right-hand side of Figure 4). A practical implementation of this flow requires reasonable memory usage and execution time. In next section we formulate the optimization problem and provide a low-complexity algorithm whose efficiency and good accuracy are experimentally proved in Section 4.

Testing for leakage-preference. Our technique needs to know the leakage-preference of cache lines. We are interested in (leak1-leak0) value which represents both type and magnitude of the leakage-preference. A Built-in Self Test (BIST) circuitry shown in Figure 5 can fulfill this purpose. It consists of a JTAG scan chain to communicate with other parts of the chip, a BIST circuit, and a block of SRAM to be tested. An external controller initially puts the cache in test mode and finally reads out leakage-preferences using JTAG scan chain. For each cache bit, the testing procedure is as follows: the BIST controller writes once a 0 and once a 1 to the bit, the on-chip leakage detector (the dark gray box) measures the leakage in each case and stores it in leak0 or leak1 registers selected by the BIST controller in Figure 5, and finally leakage-preference information (i.e., leak1-leak0 value) is shifted out using the JTAG chain. The leakage detector is a conventional A/D converter that digitizes the voltage drop across a small resistor that only at the test time is put in the V_{DD} line of the block-under-test. This A/D can be an array of simple voltage comparators similar to the one in [17] whose reference voltages can be set at design time by sizing their corresponding transistors. The authors in [17] report only 7% area overhead in a 64kb SRAM containing 512 1-bit voltage comparators (the 7% overhead also covers a 512-bit scan chain and 512 fuses). Noting that recent research on memory testing in nanometer era reveals the necessity of on-chip facilities for test-and-repair [18] and self-calibration for stability [19], embedding a leakage-detection circuitry is not unordinary.

Further elaborating this leakage testing circuit and its associated tradeoffs (e.g. the size of the SRAM block per BIST circuit and the resolution of the leakage detector) is out of scope of this paper especially since there are well-known circuit techniques for such current detectors. This leakage-preference detection can even be done during manufacturing test or even in-house using external equipments since commodity ammeters can easily measure down to 0.1fA (femto Ampere) 0 while the nominal leakage of a minimum geometry transistor is 345pA in a 90nm process available to us.

In the experiments reported in Section 4, we assume 4-bytes per cache cell for representing leak1-leak0 value. The corresponding cost is discussed at the end of that section.

3. Problem Definition

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 by 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 the instruction cache when storing basic-block instructions:

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

Each term in this summation gives the leakage energy dissipated by instruction i of basic-block bb at way w of cache.

The problem is formally defined as "Minimize E_{BB} for a given application and cache organization (i.e. for given N_s, N_w, N_{BB}, and N_i(bb) vector)."

It is noteworthy that we do not include branch instructions in the basic-blocks (as they cannot be rescheduled), and hence, E_{BB} is marginally less than total leakage energy of the application. This is confirmed by experimental results in Section 4.

3.1. Algorithms

We provide two algorithms each applied to all basic-blocks of the given application. The first one finds the absolute best schedule at the cost of longer execution time while the second one is orders of magnitude faster but loses some potential saving.

The first algorithm, an adaptation of [21], is a depth-first enumeration of all possible schedules. Here, G represents the graph of dependencies among instructions to be scheduled, BA is the base-address of them, and ready-list(G) gives all G instructions with no predecessors.

Algorithm 1: ExhaustiveSearch (G, BA)

```
Inputs:  (G: instruction-dependencies Graph),
         (BA: Base-Address for instructions in G)
Output:  (BS: the Best and complete Schedule)
```

```
1   if G is empty return empty
2   if inHash(G) return getHash(G)
3   lowestLeakage = +INFINITY; BS=empty-list;
4   for each i in ready-list(G) do
5     NS = ExhaustiveSearch(G-{i}, BA+1)
6     leak = get_BB_leakage( {i}+NS, BA)
7     if leak<lowestLeakage
8      lowestLeakage=leak; BS={i}+NS
9    endfor
10   addHash(G, BS)
11   return BS
```

This exhaustively checks all possible schedules of the instructions in G by selecting each of the ready instructions

at each step (line 4), and then calling the function recursively to schedule the remaining ones (line 5). Then line 6 calculates the leakage energy (the two inner-most summations of Eq. 3) of the new schedule (i.e., {i}+NS) at address BA, and then the best one is selected and returned (lines 3, 7, 8, 11). Two heuristics reduce execution time:

Heuristic 1: Whenever the best partial-schedule is obtained, it is saved in a hash table (line 10) which is consulted in later calls to the function (line 2) to see if the solution is already known.

Heuristic 2: A limit is put on basic-block length so that longer ones are split into several pieces smaller than that limit for rescheduling. Consequently, if such long basic-blocks exist in the code, the algorithm may not give the absolute optimum schedule but instead execution-time is kept reasonable.

Example 1: See the illustrative example in Section 2.2.

Algorithm 1 (potentially) finds the optimum schedule, but its execution time may not always be acceptable. Specifically, if an RTOS is to do this when loading an application into memory, a faster algorithm is required. For such cases, we propose the second algorithm which is far faster but loses some potential leakage saving:

```
Algorithm 2: ListScheduling(G, BA)

Inputs:   (G: instruction-dependencies Graph),
          (BA: Base Address of the basic-block)
Output:   (S: obtained Schedule)

1  S=empty-list;
2  for addr=BA to (the end of basic-block) do
3    lowestLeakage=+INFINITY; bestChoice=0
4    for each i in ready-list(G) do
5      leak = get_instruction_leakage(i, addr)
6      if leak < lowestLeakage
7         { lowestLeakage=leak;   bestChoice=i; }
8     endfor
8     S=S+{bestChoice};   G=G-{bestChoice}
9   endfor
10  return S
```

This is a list-scheduling algorithm that uses leakage-preference of cache cells as the function to choose the best instruction among the candidates at each iteration. Line 5 gives the innermost summation of Eq. 3 for instruction i.

Example 2: Applying Algorithm 2 to the same problem in Figure 3, in the first iteration, instructions 1 and 3 comprise the ready-list and the algorithm chooses 1 after the inner loop since it better matches cache line 488 (the cache location corresponding to address 1000). In the second iteration, the ready-list contains instructions 2 and 3, and the loop chooses 3 as the best choice for address 1001. Finally, the third iteration puts the only remained instruction at address 1002. This example illustrates one case where Algorithm 2 does not give the optimum schedule, while Algorithm 1 does. Note that the final iteration is not required since there is no choice left. Our implementation considers this but for simpler presentation this is not shown in the pseudo-code.

The two algorithms respectively have time complexity of O(n!) and $O(n^2)$ and memory usage of O(n!) and O(n) where n represents the number of instructions in the basic-block. Note that both algorithms correctly handle set-associative caches since the innermost summation in Eq. 3 considers individual leakage-preferences for each way.

3.2. Performance and Power Penalties

We divide this into two parts. The first part involves merely the modified application binary and the implications of running it instead of the original binary. The second part discusses overhead of our instruction rescheduling technique.

Concerning the modified application binary. Instruction-rescheduling has no impact on instruction-cache but may in rare cases marginally affect data-cache power or performance: By definition basic-blocks do not contain any branches (except as their last instruction of course), and hence, it is easy to see that the sequence of addresses of executed instructions does not change when the instructions are reordered within a basic-block, and hence, the access pattern to instruction cache, and thus the hit ratio, remains invariant. Thus, our technique has inherently no impact on instruction cache performance. Regarding 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 basic block). Our experimental results confirm this.

Concerning the instruction-rescheduling. Note that since instruction-rescheduling is applied only once for each embedded system (Figure 4), it does not impact performance or power consumption during normal system operation.

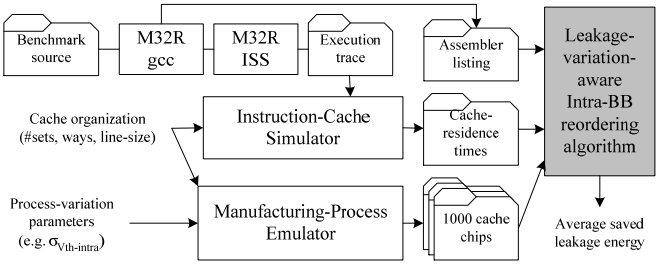

Figure 6. Simulation setup of the experiments

4. Experimental Results

We setup a simulation environment to assess our variation-aware algorithms when running various benchmarks on a processor equipped with a cache affected by process variation. Figure 6 shows our simulation setup. Benchmarks are compiled by GNU C compiler for M32R [22] (a 32-bit

RISC processor) with "-O3" option, and are run on its instruction-set simulator (ISS) to obtain an execution trace of one million instructions which is given to an instruction-cache simulator to produce per-way cache-residence times of each instruction (i.e., T(i, bb, w) values defined in Section 3). The assembler listing of the compiled benchmark is also given to the algorithm to find basic-block boundaries and the dependencies among instructions in each of them. The variation-affected cache is modeled by the manufacturing process emulator box in Figure 6 which uses Monte Carlo simulation, with mean V_{th}=400mv and $\sigma_{Vth\text{-}intra}$=60mv, to generate Gaussian-distributed V_{th} values for cache transistors and compute leak1 and leak0 values for each cache SRAM cell. To consider the randomness of process variation, we generated 1000 chips (using the same mean V_{th} and $\sigma_{Vth\text{-}intra}$) for each cache configuration and ran the algorithm on all of them. Die-to-die variation is not modeled in these experiments.

We used benchmarks from MiBench [23], MediaBench [24], and also Linux compress program (summarized in Table 1) in our experiments. For each cache configuration and benchmark, the saving in BB leakage energy (E_{BB}, given by Eq. 3) is reported. Since we eliminated branch instructions from basic-blocks, their corresponding leakage energy is not in E_{BB}. We computed their energy in the experiments and report the saving in total energy (E_{total}) as well.

Table 1. Benchmark 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

It is noteworthy that, without loss of generality, we used a RISC processor in our experiments so that all instructions have the same size. In such case, note that given an application execution trace, the cache-residence time of a basic-block instruction only depends on its address, and hence, if another instruction is put at this address as a result of rescheduling, it obtains the cache-residence time of this address. This reduces rescheduling complexity by making T(i, bb, w) values invariant during rescheduling, and hence, they need not be recalculated for each different schedule.

Experiments and results. Table 2 gives the improvements in the total energy of basic-blocks (E_{BB}) and total energy of the entire application (E_{total}) obtained by each algorithm under various cache configurations for the same-sized 2KB cache, along with their corresponding algorithm execution times on a Xeon 3.8GHz processor with 3.5GB memory. Cache line size is fixed at 4-bytes (one M32R instruction) and the number of cache ways is changed from 1 (direct-map) to 2, 4, and 8 to study the effect of associativity on the achievements. For each cache configuration, 1000 cache chips were generated. Each entry in Table 2 gives the average obtained for the corresponding benchmark over all 1000 chips. Reports for the exhaustive search algorithm correspond to splitting basic-blocks to pieces of no longer than 10 instructions. Although exhaustive search has been proposed in previous work (such as [21]) to find the near-optimal schedule, we use it primarily as a reference to evaluate the quality of the results obtained by the list-scheduling algorithm.

Expectedly, the best improvements are obtained on direct-map cache and the achievements reduce when the number of cache ways increases. This is a result of uncertainty in the location of the instruction in the cache in multi-way caches; in a direct-map cache each instruction corresponds to exactly one location in the cache, but in a w-way set-associative cache, the same instruction may go to any of the w possible locations in the same set of the cache during application execution, and hence, the algorithms have to match the instruction with multiple locations which results in reduced matching. Our cache simulator gives separate per-way

Table 2. Obtained improvements and the algorithms execution-time

Algorithm	Benchmark	E_{BB} Improvement (%)				E_{total} Improvement (%)				Execution Time (sec)			
		Cache configs (sets×ways×line_size)				Cache configs (sets×ways×line_size)				Cache configs. (sets×ways×line_size)			
		512×1×4	256×2×4	128×4×4	64×8×4	512×1×4	256×2×4	128×4×4	64×8×4	512×1×4	256×2×4	128×4×4	64×8×4
Exhaustive search	MPEG2	13.36	10.55	7.83	6.20	10.85	7.68	5.48	3.92	220.10	366.43	672.09	1086.61
	FFT	13.40	10.43	8.25	6.86	10.74	7.39	5.44	4.26	107.46	129.75	249.39	290.74
	JPEG	9.99	8.88	8.74	8.47	8.59	5.69	2.81	1.12	150.09	169.89	133.43	132.46
	Compress	13.29	13.04	12.73	11.14	10.96	6.84	6.01	5.27	91.98	94.84	93.51	187.86
	FIR	13.43	13.38	12.96	12.97	12.51	12.98	12.57	12.55	9.00	17.66	33.88	67.80
	Average	12.69	11.26	10.10	9.12	10.73	8.12	6.46	5.42	115.73	155.71	272.20	394.40
List scheduling	MPEG2	11.15	8.72	6.59	5.46	9.05	6.35	4.61	3.45	0.06	0.06	0.07	0.09
	FFT	10.98	8.48	6.91	5.82	8.80	6.01	4.56	3.61	0.03	0.03	0.04	0.05
	JPEG	7.37	6.55	6.54	5.93	6.34	4.20	2.10	0.78	0.06	0.05	0.04	0.04
	Compress	12.28	11.24	13.00	13.94	10.21	5.94	3.05	2.65	0.03	0.03	0.02	0.02
	FIR	12.30	12.68	12.23	12.14	11.90	12.30	11.86	11.75	0.00	0.00	0.00	0.00
	Average	10.82	9.53	9.05	8.66	9.26	6.96	5.24	4.45	0.03	0.03	0.04	0.04

residence-times so as to direct the matching process toward the ways with higher probability of hosting the instruction.

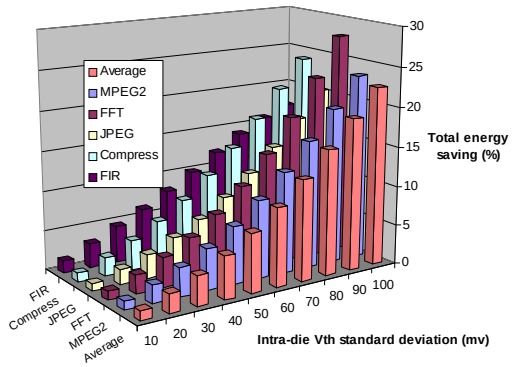

Figure 7. Saving improvement with technology scaling

Despite the above trend confirmed by the average obtained savings, the savings achieved by the list-scheduling algorithm do not always decrease with higher associativity; saving in E_{BB} increases (although marginally) from 2-way to 8-way cache for `compress` and from direct-map to 2-way for `FIR`. Examining the corresponding cache simulation results shows that higher associativity removes most cache conflicts in `compress` (99.99% hit ratio in the 8-way cache), and hence, increasingly more instructions remain forever in the same cache way once fetched, and thus they are better matched to the cache cells. This does not happen for other benchmarks since 2KB cache is not big enough for them. It is also interesting to see that list-scheduling results outperform exhaustive-search for `compress` due to several big basic-blocks where exhaustive-search fails to find the optimal schedule because of the mandatory splitting. A very tiny increase is also observed in exhaustive-search results for `FIR`; such a tiny change is due to randomness of V_{th} (and I_{off}) in different parts of the cache.

The difference between E_{BB} and E_{total} is on average 3.08%, but in some cases, such as `compress`, it even exceeds 10% and highly impacts total energy saving despite good saving in E_{BB}. This is again due to the big cache size for the benchmark: most jump and branch instructions (which cannot be rescheduled to match the cache cells) remain in the cache for the entire application run and dissipate leakage.

Algorithm 1 gives near-optimal figures for the improvements; thus, Algorithm 2 whose results are on average only 12.58% less optimal, but is 4 orders of magnitude faster is certainly a reasonable choice. Especially note that Algorithm 2 does not really need the T(i,bb,w) values since it fills basic-block instruction-locations one by one, and hence, the T value at each iteration is the same irrespective of the instruction put there. Thus, it better suits implementation in an RTOS since it needs less memory (no application trace information).

Figure 2 suggests that the achieved energy saving rises with the increase in V_{th} variation caused by technology scaling. We repeated all the above experiments (using only Algorithm 1) with σ_{Vth} varying from 10 to 100mv (with mean-V_{th}=400mv and BB-splitting threshold=10 on a 512×1×4 cache in all cases). Figure 7 shows the trend in saving results which confirm the increasing significance of the approach in future technologies.

To assess impact on data cache, we used our custom data cache simulator for M32R reporting hit-ratio as well as cache-residence times so as to compute leakage power similar to Eq. 3. For a 4KB direct-map 1024-set data cache, the hit-ratio reduction averaged only 0.29% (maximum 1.41% in FIR with 91.59% original hit-ratio) and the leakage power remained almost the same (±0.1% for different benchmarks). Although the above impacts certainly change for different benchmarks and various cache configurations, the experiments confirm that the causing cases are very rare due to locality of reference (see Section 3.2).

It is noteworthy that although our experiments involved a single-issue in-order execution processor, many processor architecture techniques such as out-of-order execution and pipelining are orthogonal to our technique since they neither affect instructions placement in the instruction memory nor they change the order of fetching instructions from the instruction cache. Also in case of multithreading or multitasking, our technique is still applicable without any modification if either (i) the placement of tasks or threads in instruction memory is fixed in all executions, or (ii) the instruction cache is virtually-addressed (i.e. uses virtual address of instructions for mapping them to cache lines) even if tasks or threads are loaded to different addresses at different times. Otherwise, when tasks or threads are dynamically loaded to different places in the instruction memory and the instruction cache is physically-addressed (i.e., uses physical addresses for mapping), the exact place of instructions in the cache cannot be statically known, and hence, our technique can only be applied at run-time which incurs some dynamic power penalty.

Memory requirements. We used Eq. 3 to formulate the problem and to present the algorithms. The same formula was implemented in the above experiments to report the results. Eq. 3, however, requires availability of both leak0 and leak1 values (and hence twice as memory) while only (leak1-leak0) value suffices for finding the best schedule. To demonstrate this, we defined another metric, called leakage-saving, showing how much the i-th instruction of the bb-th basic-block saves power if residing in w-th way of instruction cache:

$$S(i,bb,w) = \sum_{matched}\sum_{bits}|leak1 - leak0| - \sum_{mismatched}\sum_{bits}|leak1 - leak0| \quad (4)$$

where the first summation gives the leakage energy saved by a match between an instruction bit and the leakage preference of its corresponding cache cell, and the second summation gives the extra energy dissipated due to mismatches. Then, we changed the algorithms to maximize Eq. 5 below instead of minimizing Eq. 3.

$$EnergySaving = \sum_{bb=1}^{N_{BB}}\sum_{i=1}^{N_i(bb)}\sum_{w=1}^{N_w} S(i,bb,w)\times T(i,bb,w) \quad (5)$$

The experiments were repeated with this new target function which resulted in exactly the same improvements as in Table 2. As the experiments confirmed, this is another formulation for the same problem as before, but requires half memory by eliminating the need to both leak0 and leak1values.

As explained in Section 2.2, we used 4 bytes to store (leak1-leak0) value to designate leakage-preference of each SRAM cell in the cache, resulting in only 64KB of memory for a 16Kb cache. Cache residence-times are not required in list-scheduling algorithm (see above) and were only used in our experiments to assess the technique effectiveness and to compare results of the two algorithms.

5. Conclusions

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, (ii) presenting a first technique that takes advantage of this opportunity and reduces leakage by 10.73% (12.69% for E_{BB}) on average with negligible impact on system performance, and (iii) showing that the list scheduling algorithm gives near optimal results with much less memory and orders of magnitude higher speed and is suitable for usage in RTOS. It is important to note that our technique (i) saves more in future finer technologies, (ii) improves leakage not only in the standby mode but also in active mode of system operation, and (iii) is orthogonal to other techniques for leakage reduction such as body- and source-biasing.

As future directions for further research, we are extending this technique to globally move basic-blocks in the processor address space in addition to intra-BB rescheduling. Also, since the bit-width of (leak1-leak0) values affects both the test circuit overhead as well as the quality of obtained savings, it can be another area for research and optimization.

Acknowledgments

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

Referneces

[1] V. G. Moshnyaga, and K. Inoue, "Low-Power Cache Design," in *Low-Power Electronics Design*, C. Piguet Eds., CRC Press, 2005.

[2] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand, "Leakage current mechanisms and leakage reduction techniques in deep-submicron CMOS circuits, " *Proceedings of IEEE*, Vol. 91, No. 2, pp. 305-327, Feb. 2003.

[3] J. T. Kao, and A. P. Chandrakasan, "Dual-threshold voltage techniques for low-power digital circuits," *IEEE Journal of Solid State Circuits*, Vol. 35, No. 7, pp. 1009-1018, July 2000.

[4] F. Fallah, and M. Pedram, "Circuit and system level power management," in *Power Aware Design Methodologies*, M. Pedram and J. Rabaey Eds., Kluwer Academic Pub., pp. 373-412, 2002.

[5] V. De, and S. Borkar, "Low power and high performance design challenge in future technologies," *Proc. Great Lakes Symp. on VLSI*, pp. 1-6, 2000.

[6] T. Kuroda, T. Fujita, F. Hatori, and T. Sakurai, "Variable threshold-voltage CMOS technology," *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. E83-C, 2000.

[7] M. D. Powell, Y. Se-Hyun, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Gated-Vdd: a circuit technique to reduce leakage in cache memories," *Proc. Int'l Symp. Low Power Electronics and Design (ISLPED)*, 2000.

[8] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: exploiting generational behavior to reduce cache leakage power," *Proc. Int'l Symp. on Computer Architecture (ISCA)*, pp. 240-251, 2001.

[9] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power," *Proc. Int'l Symp. on Computer Architecture (ISCA)*, pp. 148-157, 2002.

[10] A. Abdollahi, F. Fallah, and M. Pedram, "Runtime mechanisms for leakage current reduction in CMOS VLSI circuits," *Proc. Int'l Symp. on Low Power Electronics and Design (ISLPED)*, pp. 213-218, August 2002.

[11] L. Clark, and V. De, "Techniques for Power and Process Variation Minimization," in *Low-Power Electronics Design*, C.Piguet Eds.,CRC Press, 2005.

[12] M. Goudarzi, T. Ishihara, and H. Yasuura, "Ultra-Leaky SRAM Cells Caused by Process Variation: Detection and Leakage Suppression at System-Level," *Proc. Computer Society of Iran Computer Conference (CSICC)*, 2007.

[13] A. Agarwal, B. C. Paul, H. Mahmoodi, A. Datta, and K. Roy, "A process-tolerant cache architecture for improved yield in nanoscale technologies," *IEEE Trans. VLSI*, Vol. 13, No. 1, pp. 27-38, 2005.

[14] J. Luo, S. Sinha, Q. Su, J. Kawa, and C. Chiang, "An IC manufacturing yield model considering intra-die variations," *Proc. of Design Automation Conference (DAC)*, pp. 749-754, 2006.

[15] K. Agarwal, and S. Nassif, "Statistical analysis of SRAM cell stability," *Proc. Design Automation Conference (DAC)*, pp. 57-62, 2006.

[16] E. Toyoda, "DFM: Device & Circuit Design Challenges," *Int'l Forum on Semiconductor Technology*, 2004.

[17] K. Kanda, N. Duc Minh, H. Kawaguchi, and T. Sakurai, "Abnormal leakage suppression (ALS) scheme for low standby current SRAMs," *Proc. IEEE Int'l Solid-State Circuits Conference*, 2001.

[18] E. J. Marinissen, B. Prince, D. Keitel-Schulz, and Y. Zorian, "Challenges in embedded memory design and test," *Proc. Design Automation and Test in Europe (DATE)*, Vol. 2, pp. 722-727, 2005.

[19] S. Ghosh, S. Mukhopadhyay, K. Kim, K. Roy, "Self-calibration technique for reduction of hold failures in low-power nano-scaled SRAM," *Proc. Design Automation Conference. (DAC)*, pp. 971-976, 2006.

[20] DSM-8104 ammeter, http://www.nihonkaikeisoku.co.jp/densi/toadkk_zetuenteikou_dsm8104.htm, August 2008.

[21] H. Tomiyama, T. Ishihara, A. Inoue, and H. Yasuura, "Instruction scheduling for power reduction in processor-based system design," *Proc. Design Automation & Test in Europe (DATE)*, pp. 855-860, 1998.

[22] M32R family 32-bit RISC microcomputers, http://www.renesas.com/fmwk.jsp?cnt=m32r_family_landing.jsp&fp=/products/mpumcu/m32r_family/, August 2008.

[23] MiBench (version 1.0), http://www.eecs.umich.edu/mibench/, August 2008.

[24] MediaBench, http://cares.icsl.ucla.edu/MediaBench, August 2008.

Maziar Goudarzi received his B.S., M.S., and Ph.D in Computer Engineering from Sharif University of Technology, Tehran, Iran, in 1996, 1998, and 2005 respectively. He joined System LSI Research Center of Kyushu University in 2006, where he is currently a Research Associate Professor. His research interests include low power design of processor-based systems and hardware-software co-design.
E-mail: goudarzi@sharif.edu

Tohru Ishihara received his B.S., M.S., and Ph.D degrees in computer science from Kyushu University in 1995, 1997 and 2000 respectively. From 1997 to 2000, he was a Research Fellow of the Japan Society for the Promotion of Science. For the next three years he worked as a research associate in VLSI Design and Education Center, the University of Tokyo. From 2003 to 2005, he stayed at Fujitsu Laboratories of America as a research staff of an advanced CAD technology group. In 2005, he returned to Kyushu University as an associate professor. His research interests include low power SoC design and hardware/software co-design. He is a member of IEEE, ACM, IPSJ and IEICE.
E-mail: ishihara @slrc.kyushu-u.ac.jp

Paper Handling Data:

Submitted: 01.08.2007
Accepted: 13.08.2008
Corresponding author: Dr. Maziar Goudarzi,
Department of Computer Engineering, Sharif University of Technology, Tehran, Iran.