A Software Technique to Improve Yield of Processor Chips in Presence of Ultra-Leaky SRAM Cells Caused by Process Variation

Maziar Goudarzi, Tohru Ishihara, Hiroto Yasuura

System LSI Research Center
Kyushu University
Fukuoka, Japan
Tel: +81-92-847-5193
Fax: +81-92-847-5190
e-mail: {goudarzi, ishihara, yasuura}@slrc.kyushu-u.ac.jp

Abstract - Exceptionally leaky transistors are increasingly more frequent in nano-scale technologies due to lower threshold voltage and its increased variation. Such leaky transistors may even change position with changes in the operating voltage and temperature, and hence, redundancy at circuit-level is not sufficient to tolerate such threats to yield. We show that in SRAM cells this leakage depends on the cell value and propose a first software-based runtime technique that suppresses such abnormal leakages by storing safe values in the corresponding cache lines before going to standby mode. Analysis shows the performance penalty is, in the worst case, linearly dependent to the number of so-cured cache lines while the energy saving linearly increases by the time spent in standby mode. Analysis and experimental results on commercial processors confirm that the technique is viable if the standby duration is more than a small fraction of a second.

I Introduction

Supply voltages (V_{DD}) as well as transistor threshold voltages (V_{th}) have been scaled down, although at different rates, along with the manufacturing technology scaling. This reduced V_{th}, however, exponentially increases subthreshold current (I_{off}) [1]. Furthermore, when approaching atomic scales, dopant fluctuation causes higher statistical intra-die as well as inter-die V_{th} variations such that increasingly more transistors have very low V_{th} and correspondingly very high I_{off} which in turn means power consumptions much higher than normal in standby mode. This makes increasingly more chips unsuitable for low-power long-standby embedded applications [2], and hence decreases yield, while also results in a small number of transistors being responsible for most of the standby power consumed in a chip. SRAM cells are even more susceptible to this effect since they are typically designed with minimum transistor sizes (V_{th} variation is inversely proportional to transistor channel length [1]) and they take up most of the area of today processors (e.g. 70% of StrongARM110 [3]). Thus yield of microprocessors with on-chip caches can be improved for low-power long-standby embedded applications if caches containing ultra leaky SRAM cells can be tolerated without noticeable performance degradation. Such ultra leaky cells can be simply viewed as faulty cells and several fault-tolerant techniques for memories can be applied to replace [2, 4] or tolerate them with negligible performance degradation [5]. However, since V_{th} is also a function of V_{DD} and temperature, the failure map of such chips changes if the operating conditions are altered. This renders static redundancy techniques [2] inapplicable and makes dynamic ones very expensive.

In this paper, we focus on the effect of ultra leaky transistors during standby mode in long-standby applications (e.g. security cameras, PDAs, etc. that spend most of their time in standby mode) and propose a software-based runtime technique that cures the leaky cache lines during this standby mode. We show in Section II that normally the value stored in an SRAM cell determines its leakage power. Our approach is to store the *leakage-safe* value in the cell when the system is going to enter standby mode. To the best of our knowledge this is the first software-based technique that addresses leakage power issue.

The rest of the paper is organized as follows. In Section II, we summarize previous work and our observation and approach. The definition of the problem, our algorithm for detecting leaky cache lines, and the procedure to handle them in standby mode is given in Section III. Analysis and experimental results are presented in Section IV and finally, the paper is concluded in Section V.

II. Related Works and Our Approach

A. Related Works

Power gating [6] removes the power of a circuit or block to reduce leakage, but this cannot be applied to cache memories since valuable information will be lost and the processor needs a cold start from an empty cache when waking up, resulting in high performance loss. Turning off only parts of the cache [7], [8] or putting them in a low-energy "drowsy" mode using two different supply voltages [9] involve this issue, but they require circuit-level techniques, and moreover, cannot handle ultra leaky transistors caused by process variation.

Reverse body biasing [10], forward body biasing [11] and dynamic V_{th} control [12] can also be used to reduce leakage power. These techniques, however, require device-level modification of the system and use sophisticated techniques to control body bias while we use the chip as is and propose a software-level remedy for ultra leaky cells. Moreover, although body biasing techniques can effectively address die-to-die V_{th} variations and tune the V_{th} of all chips to near ideal value, to handle within-die variation (which is the focus of this work) different bias voltages need to be applied at several places in the same chip, which makes it very expensive and impractical to implement.

Abdollahi et al. [13] use the fact that the standby power consumption of a circuit is a function of its inputs and formulate the problem of finding the *minimum leakage vector* (MLV) using a series of Boolean Satisfiability problems. While their technique is aimed at logic circuits but not cache memories, we use the same fact of value-dependence of leakage power so as to find a leakage-safe value (see next section) for leaky cache lines.

The need for low-standby-current SRAMs in presence of defects caused by process fluctuation and/or dust has motivated design of cache architectures that can selectively cut off the leakage path of abnormally leaky cells and replace corresponding rows and columns with spare ones [2]. We achieve the same goal, without modifying the cache architecture and without the associated overhead, by storing leakage-safe values in the abnormally leaky cells during standby mode. We actually replace the static fuse-blowing operation in [2] with a software-level dynamic leakage suppression technique that we show in Section IV to impose negligible performance overhead. This dynamic nature of our technique helps to adapt to changing leakage patterns while static techniques inherently cannot.

B. *Observation*

Two main observations motivate our approach: firstly, an increasingly bigger portion of leakage power consumption is dissipated by only a few transistors in sub-90 nm technologies; and secondly, this leakage power in SRAM cells can be cancelled by carefully storing values there.

Process variation in sub-90 nm technologies is a well-known phenomenon that causes variations in the V_{th} of transistors. Since subthreshold current is exponentially dependent to V_{th}, the transistors with very low V_{th} will dissipate very high leakage power. The V_{th} value is believed to follow Gaussian distribution [14]. We ran a set of Monte Carlo simulations using extrapolated V_{th} standard deviation (σ_{Vth}) values to assess the effects of this variation on I_{off}. Table I summarizes the input data and results. As the basis for the extrapolation, we used the values reported by Toyoda et al. [15] from physical implementation of several nMOS transistors in 130 nm technology. Physical gate lengths and average V_{th} values (second and third row of the table) are ITRS estimations [16] for low-power processes. The fourth row contains extrapolated σ_{Vth} values considering that Vth variation is proportional to $1/\sqrt{L \times W}$, where L and W are respectively effective channel length and width. ITRS roadmap also shows similar prospects [16].

Assuming that a transistor with V_{th} below 100mv is considered ultra leaky (this is debatable; see Section IV-B), last two rows of Table I show that a very small fraction of transistors dissipate a high portion of the leakage power. Moreover, not only the number of such leaky transistors is increasing in finer technologies, but also their share in total leakage power is increasingly more significant. Consequently,

TABLE I
Share of leaky transistors in total leakage in sub-90 nm processes.

Technology Node	130 nm	90 nm	80 nm	70 nm
Physical gate length (nm)	100*	37**	32**	28**
Average V_{th} (mv)	308.3*	320**	330**	340**
V_{th} std. dev. (σ_{Vth}, mv)	22.1*	59.7	69.1	78.9
Leaky transistors (%)	0	0.01	0.04	0.12
Share in total leakage (%)	0.0	3.0	10.8	27.7

* Values taken from physical implementation in [15]
** Values from ITRS forecast [16]

it is necessary to address such higly leaky transistors in current and future technologies. A more detailed analysis in 90nm is presented in Section IV.

As the second basis for our work, we take advantage of this observation that actual occurrence of abnormal leakage current in a SRAM cell depends on the value stored in it. Fig. 1 shows a 6-trnasistor SRAM cell storing a logic 1 value. Now, assuming one leaky transistor per cell, even if M4 is actually leaky (it has an abnormally low V_{th}), the cell shall not be leaky since V_{ds} for M4 is zero. On the contrary, if the cell contains a 0 logic value, M4 will be in the *off* state with V_{ds}=V_{DD} and the cell shall be leaky accordingly. The same holds for all other 5 transistors in the cell. Abnormally low V_{th} in either M1, M2, or M5 results in a cell that is leaky if storing a 1 (called a *1-leaky* cell), and such V_{th} in M3, M4, or M6 causes high leakage in the cell only if it is storing a 0 (called a *0-leaky* cell). (To justify M1 and M6 leakage cases, note that bit lines are precharged to V_{DD} in SRAM cells [1].) Consequently, assuming that at most one of the 6 transistors in a cell are leaky, the cell will be either 0-leaky or 1-leaky depending on which transistor leaks. Thus there is a *leakage-safe* value for each leaky cell: 0 for 1-leaky cells and 1 for 0-leaky cells.

A cache line containing one or more leaky cells is called a *leaky cache line*. The leakage-safe value for a cache line is simply any value with a leakage-safe bit value at leaky bit positions. For example, if an 8-bit cache line has a 1-leaky cell at its least significant bit, any even value (i.e., xxxxxxx0) will be a leakage-safe value for that line.

C. *Our Approach*

Fig. 2 gives a big picture of our technique. The gray parts are those introduced by our technique. A testing procedure (see Section III-B) determines the leaky cache lines along

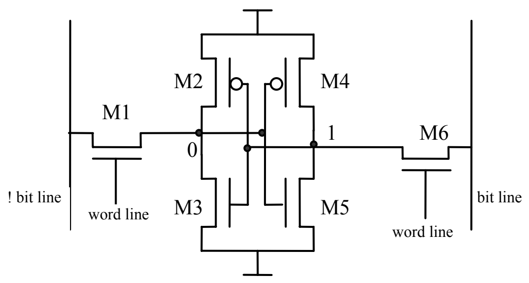

Fig. 1. A 6-transistor SRAM cell storing a logic 1 value.

with their leakage-safe values and stores this information for later use when entering standby mode. At such time, all leaky cache lines are filled with their corresponding leakage safe values so that they do not consume abnormally high leakage power while in standby mode (the same technique can be employed when the ultra leaky cell resides in the tag area of the cache). When the processor wakes up, it continues as usual. Consequently, the cache capacity is not reduced by our technique, and hence, the processor performance is not affected apart from a few mandatory cache misses caused by invalidated contents of leaky cache lines.

Obviously, the amount of energy saving directly depends on the time spent in standby mode. The upper gray box in Fig. 2 is an offline operation and does not impact system power consumption. The energy leaked through leaky transistors is saved in return for some energy to fill cache lines with leakage-safe values, and some other energy to later fetch valid data from next-level memory hierarchy to invalidated (leaky) cache lines. This introduces a *viability threshold* for our technique which is defined as “the minimum standby time beyond which this technique can actually save energy”.

The main advantage of our technique is its fully software-based nature which requires no change in the processor architecture or at circuit-level. Consequently, not only it can adapt to changing leakage patterns, but also it introduces no area overhead and furthermore, it does not even need a change in the object code software. Entering standby mode is usually managed by a Real-Time OS (RTOS) that monitors system status; thus, once this RTOS is modified, the application object code remains unchanged. Consequently, for a negligible performance loss (see Section IV), chips with any number of such ultra leaky cache cells can be salvaged. This software-based nature still helps when leaky cells are increased due to aging and electromigration while circuit-level techniques cannot. It also reduces production time and cost spent on fuse cutting operations.

III. Problem Definition

A. Problem formulation

Using the following notation,

N: The number of cured leaky cache lines,

$\boldsymbol{P_{leak}}$: Average leakage power in every leaky cache line,

the problem can be formally defined as follows:

“For a given processor and P_{leak}, *(i)* find the viability threshold (t_{viable}) beyond which the technique will be useful, and *(ii)* maximize N for a given maximum acceptable performance loss.”

Note that P_{leak} differs among leaky cache lines depending on the number and actual V_{th} of their leaky transistors, but an average, or even the worst case, can be used here without loss of generality.

B. Leaky-cell detection algorithm

Since ultra leaky transistors are a result of process variation, which is random in nature, such transistors are expected to be distributed allover the cache, and hence, it is very unlikely that more than one such transistors reside on the same cache line if only a few of them exist. The following algorithm, with O(n) time complexity where n represens the number of cache lines, can detect all leaky cache lines in this case:

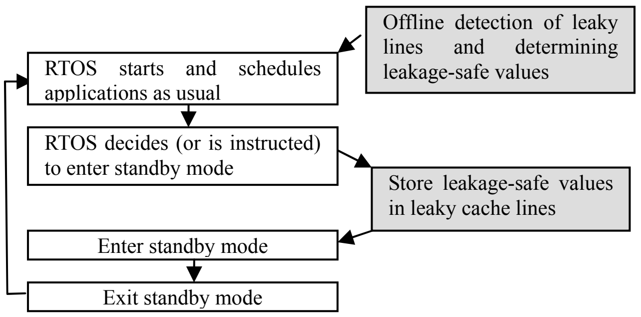
Offline detection of leaky lines and determining leakage-safe values
RTOS starts and schedules applications as usual
RTOS decides (or is instructed) to enter standby mode
Store leakage-safe values in leaky cache lines
Enter standby mode
Exit standby mode

Fig. 2. Leakage reduction flow.

```
Procedure Detect_Leaky_Cache_Lines
Outputs: List of all leaky cache lines and their leakage-safe values
  Start with arbitrary contents in the cache.
  0_leaky_lines_list = 1_leaky_lines_list = empty list;
  For each cache line k do
    Write all 0's to the line. Measure the quiescent current ($I_0$)
    Write all 1's to the line. Measure the quiescent current ($I_1$)
    If $I_1 >> I_0$ then add k to 1_leaky_lines_list
    If $I_0 >> I_1$ then add k to 0_leaky_lines_list
  Leakage-safe value = all 1's for 0_leaky_lines
  Leakage-safe value = all 0's for 1_leaky_lines
```

This approach is indeed similar to ΔI_{DDQ} testing [17]. The idea is that if there is a leaky cell inside the line under test, the measured I_{DDQ} with all 0’s in the line will be meaningfully different from that of all 1’s. Although initially proposed for single leakage case, this algorithm can actually detect multiple leaky cells in the same line provided that they are all of the same type. It, however, may be misled if multiple leaky cells of different types exist in the same cache line. For such case, we extend the algorithm as follows:

```
Procedure Detect_Multiple_Leakage_Cache_Lines
Outputs: List of all leaky cache lines and their leakage-safe values
  Start with arbitrary contents in the cache.
  leaky_lines_list = empty list
  For each cache line k do
    For each bit position j do
      Set bit j of line k to 0. Measure the quiescent current ($I_0$)
      Set bit j of line k to 1. Measure the quiescent current ($I_1$)
      If $I_1 >> I_0$ then mark the bit as 1_leaky, safe_value[j]=0
      else if $I_0 >> I_1$ then mark the bit as 0_leaky, safe_value[j]=1
      else safe_value[j]=x
    If a leaky bit is in the line, then add k and safe_value to
leaky_lines_list
```

Here, leaky_lines_list keeps a list of cache lines containing one or more leaky cells along with their corresponding leakage-safe values. An x in a bit position means *don’t care* since that bit is not leaky. The idea of the algorithm is the same as the previous one, but now applied to bit positions as well.

The amount of difference between I_1 and I_0 corresponds to the definition of *ultra leaky* transistor. Note that this difference must be detectable using current measurement equipment, which in our case can measure down to100nA. (Even on-chip current sensors, such as [1,2], can be used here, but elaborating this is out of scope of this paper.) We show in Section IV-B that "ultra leaky" can be reasonably defined as "leaking more than 500 times higher current than average". While other ratios can be applied if appropriate, we use 500 in our experiments. Noting that the average I_{off} is around 345pA in the 90 nm process available to us, the I_1 to I_0 difference would be above 172.5 nA which is well detectable by our available equipment.

C. *Leakage reduction procedure*

The above leakage-detection procedure is performed offline before the system starts normal operation. This may be just once after manufacturing, or whenever the operating conditions change (e.g. V_{DD}, temperature, or aging) such that the leakage pattern may change. The list of leaky cache lines and their corresponding leakage-safe values are stored somewhere accessible to the RTOS controlling the system (or to the application itself if no RTOS is used). When going to enter standby mode (e.g. when `OSTaskIdleHook()` function is called in case of uC OS [18]), this list is consulted to store safe values in leaky cache lines (Fig. 2).

This gives a power saving proportional to total number of leaky cells cured, in return for an additional energy consumed to store leakage-safe values in the leaky lines (denoted by E_{lock}). Further extra power, and performance, must also be paid if the content of the leaky cache line was valid before entering standby mode and is accessed again just after exiting this mode. In such case, since the valid and later-accessed data is overwritten by a leakage-safe value, it will need to be re-fetched to the cache when accessed resulting in an extra energy E_{fetch}. The values of E_{lock} and E_{fetch} are application-independent and only depend on the processor used and its architecture. Thus the worst-case price paid per cured cache line is constant, and hence, the achieved energy saving only depends on the time spent in standby mode. As long as this energy saving surpasses the cost, the technique becomes *viable*. Section IV-A presents an analysis to formulate answers to problems presented in Section A above.

IV. Analysis and Experimental Results

A. *Analysis*

Without loss of generality, we assume a unified data and instruction cache. We define the following additional symbols for one cycle of running a program and going to standby mode:

$E_{sg}(t)$: Gross energy saving per cured cache line after spending *t* time units in standby mode.

E_{lock}: Energy consumed by a cache lock instruction,

E_{fetch}: Energy consumed to re-fetch data to a cache line,

$E_{sn}(t)$: Net energy saving in the entire cache after spending *t* time units in standby mode.

T_c: Access time of the cache.

T_M: Access time of next memory hierarchy beyond cache.

m, m': Total program cache misses before (*m*) and after (*m'*) applying our technique.

T_e, T'_e: Total execution time before (T_e) and after (T'_e) applying our technique.

Net energy saving after *t* time units spent in standby mode can be given by subtracting energy consumed by extra operations from the gross energy saving:

$$E_{sn}(t) = N \times \left(E_{sg}(t) - E_{lock} - E_{fetch}\right) \quad (1)$$

Assuming the same P_{leak} for all leaky cache lines[1] gives:

$$E_{sg}(t) = P_{leak} \times t \quad (2)$$

The viability threshold is the minimum time beyond which $E_{sn}(t)$ is positive, which means:

$$t_{viable} = \left(E_{lock} + E_{fetch}\right) / P_{leak} \quad (3)$$

Now to formulate performance penalty, note that our technique can be viewed as inserting some mandatory cache misses (equal to the number of cache lines cured) into the original hit-miss pattern of the running program. However further note that such misses do not necessarily replace a *hit* of the original pattern, but may even fall on an original *miss*. The following two examples illustrate these two cases.

Example 1: Assume that the cache line corresponding to memory address 1000 is leaky. Further assume that the system enters standby mode just between executing the following two instructions:

```
mov r1, Mem[1000]
mov r2, Mem[1000]
```

Clearly, a hit occurs for the memory access of the second instruction in the original trace, while after applying our technique, a miss happens there.

Example 2: Now assume that the cache structure is such that addresses 1000 and 1128 share the same cache line. If the system enters standby mode just between the following two instructions:

```
mov r1, Mem[1128]
mov r2, Mem[1000]
```

a miss occurs for the second instruction even in the original trace.

In other words, our technique adds some misses to the access pattern of the program running on the processor such that the number of these misses is at most equal to *N*.

Total execution time before and after applying our technique is:

$$T_e = \left(mT_M + (1-m)T_c\right) + C \quad (4)$$

$$T'_e = \left(m'T_M + (1-m')T_c\right) + C \quad (5)$$

where *C* represents a constant time spent in all operations other than memory accesses; this is invariant since our technique only affects some of memory access operations and has no side effect on other operations of the processor. Thus the performance loss is:

$$T'_e - T_e = (m' - m)(T_M - T_c) \le N \times (T_M - T_c) \quad (6)$$

where *m'-m* actually represents the number of cache misses caused by our technique which is, in the worst case, equal to *N* for each entry to standby mode. Further note that the value of (6) is in the order of nano-seconds unless *N* is very high which corresponds to an extremely leaky chip.

Equations (1) and (6) clearly demonstrate that both energy saving and maximum performance loss are ascending linear functions of *N* for a given *t* bigger than t_{viable}. Depending on the acceptable performance loss or the desired energy saving,

[1] This can be easily relaxed in which case only the formula gets more complicated without gaining any more insight.

it is possible to choose corresponding number of leaky cache lines to be treated in this way.

B. Experimental results

We ran Monte Carlo simulations of 1000 cache memory chips in 90nm technology with V_{th}=320mv and σ_{Vth}=59.7mv to evaluate the scheme. Each cache has 512 lines with 256 bits of data and 20 bits of tag per line (=847872 transistors). The value of transistor V_{th} is the determining factor for its becoming *ultra leaky* (defined below). Assuming various upper limits for V_{th} of ultra leaky transistors, we analyzed each cache instance and measured the following items that are respectively given in the columns of Table II: the ratio of I_{off} of leaky transistors to the average one, the number of leaky transistors, leaky SRAM cells, leaky cache lines, the amount of power leakage per leaky cache line (the P_{leak} parameter in Section III-A), and also the yield. Table II gives the average values obtained over the 1000 chips. When generating random V_{th} values for the simulations, we set the minimum V_{th} to 5mv to avoid abnormally low and negative values. Thus, there is artificially no leaky transistor with V_{th} below 5mv. Table II shows that with lower V_{th} limits, the leakage per transistor increases (columns 2 and 6) but the number of leaky transistors decreases (columns 3 to 5), and hence yield increases.

A point to discuss here is the definition of *ultra high leakage*. In [2], 1uA (in 0.6u technology) is suggested which corresponds to 2900 times average I_{off} (=345pA) in the 90nm technology available to us. Given that the smallest current that our equipment can detect is 100nA, a per transistor leakage above 290 times the average will be detectable. Any value above 290 will be detectable and reasonable here. Choosing higher ratios results in detecting fewer, but leakier, transistors while also detectability is increased due to higher difference between I_1 and I_0 (see Section III-B). On the other hand, lower ratios result in higher number of leaky cache lines, suggesting that the cache had better be entirely turned off instead. Thus, as a tentative definition of *ultra leaky transistor* we suggest "those that leak more than 500 times higher than average". Note that having a yield of zero in Table II means that none of the cache instances are *suitable for long-standby low-power applications*; however, they may still be suitable for other applications. Our technique can make even such chips suitable for long-standby low-power applications, resulting in a yield of 100%.

To assess the costs vs. benefits in a real-life environment, practical values for the problem parameters are given here. Our implementation of M32R processor on a 0.18u process typically consumes 200mW at 50MHz, resulting in 4nJ per clock cycle. Assuming one instruction to store a leakage-safe value in a cache line, E_{lock} and E_{fetch} are less than 20nJ each. Note that this is an overestimate for E_{lock} and E_{fetch} since M32R has a 5-stage pipeline, and hence, the per clock energy consumption corresponds to multiple instructions being executed in the pipeline. For T_M and T_c typical values can be 10 and 1ns respectively in 0.18u [19]. Although practical values for 90nm implementations were not available to us to use here, all these values are conservatively higher than 90nm so that benefits are not overestimated. Fig. 3 shows the energy-performance tradeoff curves for varying number of cured cache lines, and for varying V_{th} limits defining leaky transistors. The number of data points for each V_{th}-limit corresponds to maximum number of leaky cache lines per chip obtained by Monte Carlo simulation of 1000 chips. Note that power saving as well as performance loss are application-independent and only depend on the number of cured cache lines and the choice of V_{th} limit (or equivalently, the I_{off} ratio).

Different processors exhibit different savings in this scheme due to their different power consumption per instruction. Fig. 4 compares M32R and ARM920 both implemented in 0.18u technology. Power savings are reported for 80mv V_{th} limit. ARM920 implementation can save more due to less power consumption per instruction (800uW/MHz with cache [20]). We assumed the same cache and memory configurations for both processors, resulting in the same performance loss for a given number of cured cache lines irrespective of the processor used. The viability threshold (t_{viable}) also changes by the processor, as Table III shows, but it is always just a fraction of a second, proving

TABLE II
Monte Carlo simulation results for 1000 chips of 128Kb cache.

V_{th} limit (mv)	I_{off} ratio	#leaky trans.	#leaky cells	#leaky lines	P_{leak} (nW)	Yield (%)
100	379	103.697	103.664	93.868	144.1	0
80	602	26.572	26.572	25.918	212.6	0
50	1205	3.037	3.037	3.029	416.5	6.20
20	2260	1.123	1.123	1.118	782.6	78.8
10	2566	1.030	1.030	1.030	884.9	90.1
5	2627	1.014	1.014	1.014	906.0	92.9

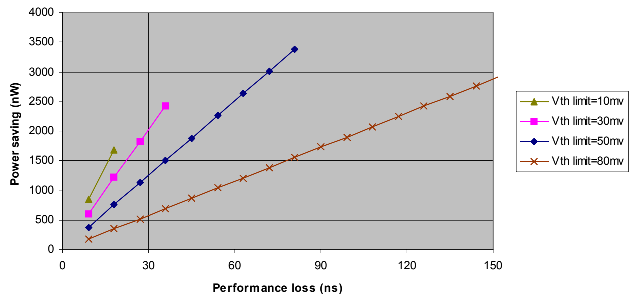

Fig. 3. Worst case energy-performance tradeoff curves for a 0.18u M32R processor.

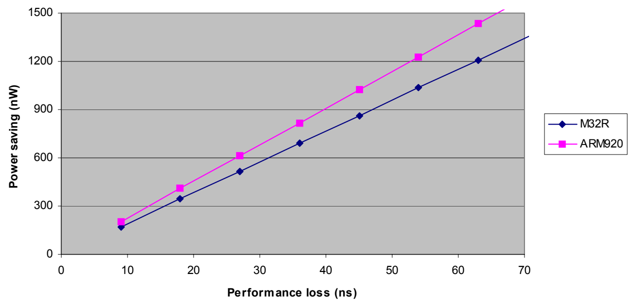

Fig. 4. Energy-performance tradeoff curves comparing M32R and ARM920 processors.

TABLE III
Viability threshold for ARM920 and M32R processors

Viability threshold (s)	V_{th} limit (mv)			
	10	20	50	80
M32R	0.0452	0.0513	0.0963	0.1929
ARM920	0.0090	0.0103	0.0193	0.0386

the usefulness of the technique for long-standby applications. Here, P_{leak} is taken from simulation results in Table II.

Comparing 3rd and 4th columns of Table II, it can be seen that more than one leaky transistor may exist in the same SRAM cell when the V_{th} limit is set to 100mv or higher. This violates our initial assumption and suggests 100mv as the upper bound of applicability of our technique. However note that although our technique is not specifically designed for such multiple-leaky cases, it can still be helpful here by finding less leaky values for such cells (see Section III-B).

V. Summary and Conclusions

In this paper, we presented a first software technique to improve yield by suppressing leakage current of ultra leaky transistors of cache in standby mode. One major advantage of the technique is its dynamic nature which enables it to handle dynamic effects, such as increased leakage due to aging, that cannot be addressed by static techniques such as fuse cutting and spare replacement as part of manufacturing process. Consequently, it addresses leaky cells caused by aging, it reduces production time and cost by eliminating fuse-cutting, and it can also be used for suppressing abnormal leakages due to any other causes (e.g. dust or electromigration) and then replacing the leaky cache line with spare ones (using programmable address decoders) without the expensive and slow fuse-blowing circuit-level techniques.

The applicability limits of the technique were presented and it was shown that the significance of the technique will even increase in future technologies. If the standby mode is longer than a tiny fraction of a second (depending on the chip manufacturing process and the processor used), our technique becomes viable with negligible performance penalty. We characterized the power saving and performance penalty of our technique with respect to number of leaky cache lines cured, so that it is possible to tune the number of cured cache lines versus maximum desired performance loss. We are developing leakage-aware compiler techniques so as to reduce leakage even in the active mode of system operation. Elaborating the test techniques for diagnosing ultra leaky cells is another part of our future work.

Acknowledgements

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.

References

[1] N.H.E. Weste, D. Harris, *CMOS VLSI Design : A Circuits and Systems Perspective*, Addison Wesley, 2004.

[2] K. Kanda, N. Duc Minh, H. Kawaguchi, and T. Sakurai, "Abnormal leakage suppression (ALS) scheme for low standby current SRAMs," *Proc. IEEE International Solid-State Circuits Conference*, pp. 174-176, 2001.

[3] J. Montanario, et al., "A 160-MHz 32-b 0.5-W CMOS RISC microprocessor," *IEEE Int'l Solid-State Circuits Conference*, 1996.

[4] G. Sohi, "Cache memory organization to enhance the yield of high performance VLSI processors," *IEEE Trans. on Computers,* vol. 38, no. 4, pp. 484-492, April 1989.

[5] T. Ishihara, F. Fallah, "A cache-defect-aware code placement algorithm for improving the performance of processors," *Proc. Int'l Conference on Computer-Aided Design*, pp. 995-1001, 2005.

[6] J.T. Kao, A.P. Chandrakasan, "Dual-threshold voltage techniques for low-power digital circuits," *IEEE Journal of Solid State Circuits*, Vol. 35, pp. 1009-1018, July 2000.

[7] M.D. Powell, et al., "Gated-V_{dd}: a circuit technique to reduce leakage in cache memories," *International Symposium Low Power Electronics and Design*, 2000.

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

[9] K. Flautner, et al., "Drowsy caches: simple techniques for reducing leakage power," *Proc. Int'l Symposium on Computer Architecture,* pp. 148-150, 2002.

[10] F. Fallah, 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.

[11] V. De, S. Borkar, "Low power and high performance design challenge in future technologies," *Proc. the 10th Great Lake Symposium on VLSI*, pp. 1-6, 2000.

[12] T. Kuroda, T. Fujita, F. Hatori, and T. Sakurai, "Variable threshold-voltage CMOS technology," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences,* vol. E83-C, pp. 1705-1715, 2000.

[13] A. Abdollahi, F. Fallah, M. Pedram, "Runtime mechanisms for leakage current reduction in CMOS VLSI circuits," *Proc. Int'l Symposium on Low Power Electronics and Design,* August 2002.

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

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

[16] Int'l Tech. Roadmap for Semiconductors, http://www.itrs.net

[17] C. Thibeault, "On the Comparison of Delta I_{DDQ} and I_{DDQ} Testing," *Proc. VLSI Test Symp.*, pp. 143-150, 1999.

[18] uC Operating System, http://www.ucos-ii.com/

[19] P. Keltcher, S. Richardson, S. Siu, "An equal area comparison of embedded DRAM and SRAM memory architectures for a chip multiprocessor," *HP Labs. 2000 Technical Reports*, April 2000.

[20] ARM920T, http://www.arm.com/products/CPUs/ARM920T.html