Partitioned Cache Architectures for Reduced NBTI-Induced Aging

[†]Andrea Calimera, [‡]Mirko Loghi, [†]Enrico Macii, [†]Massimo Poncino

[†]Politecnico di Torino, 10129, Torino, ITALY
[‡]Universitå di Udine, 33100, Udine, ITALY

***Abstract*—Conventional power management knobs such as voltage scaling or power gating have been shown to have a beneficial effect on the aging phenomena caused Negative Bias Temperature Instability (NBTI). Such a benefit can be especially exploited in SRAM memories, which are particularly sensitive to NBTI effects: given their symmetric structure, they cannot in fact take advantage of value-dependent recovery.**
We propose an architectural solutions that is based on the idea of partitioning a memory into multiple banks of identical size. While this organization has been widely used for reducing both dynamic and static power, its exploitation for aging benefits requires proper management of the existing idleness of the various banks. This can be achieved by means of a sort of time-varying addressing scheme in which addresses are mapped to different banks over time in such a way that the idleness is uniformly distributed over all the banks.
Experimental analysis shows that it is possible to simultaneously reducing leakage power and aging in caches, with minimal overhead and without modifying the internal structure of the SRAM arrays.

I. Introduction

Power and reliability have traditionally been considered as conflicting metrics, since most design solutions for improving reliability (redundant circuits, strong signals, large devices) are intrinsically power inefficient. The recent emergence of reliability issues in the form of *aging* (i.e., temporal drift of performance) of devices has opened a new perspective of this dichotomy.

The most critical source of device aging in sub-65nm technologies is Negative Bias Temperature Instability (NBTI) [1], which affect pMOS devices under negative bias (i.e., $V_{gs} < 0$, i.e., when a "0" is applied on the gate input of a pMOS). The physical effect is actually a temporal drift of the threshold voltage, which translates into a delay increase over time. Such an increase is partially mitigated by the application of a logic "1" to the pMOS gate: under this condition, the device will partially recover the delay [1].

Recent works have shown that the two traditional power management knobs may help alleviating the aging. *Voltage scaling* has a beneficial effect because supplying a device with a smaller V_{dd} translates into a smaller V_{gs}, and therefore in a smaller magnitude of negative bias [2].

Power gating, when implemented through a footer transistor, is an even more powerful lever: when a logic block is disconnected from the ground network the floating nodes inside the block are in fact pulled to a logic "1", thus completely nullifying the aging effects [3].

This link between power management and aging establish thus a novel type of power/reliability tradeoff: the latter are not conflicting metrics anymore; rather, the tradeoff that now emerges is due to the fact that the application of such power management comes at the price of some performance penalty. For voltage scaling, because device delay is inversely proportional to supply voltage; for power gating, because the on-resistance of the sleep transistor causes a voltage drop across it, which reduces the effective voltage swing, causing in turn a performance penalty of the gated block in the active state. As a result, the mitigation of the temporal drift in performance provided by these power management knobs must be weighted against the time-zero delay penalty they introduce.

Some solutions for managing this tradeoff have been proposed in the literature for logic circuits [4], [2], [3] as well as for memory structures [5], [6], [7].

In this work, we focus on SRAM memories, and specifically on caches, by proposing a purely architectural approach based on a multi-bank, partitioned cache implementation, and which achieves maximum aging reduction at no penalty in power consumption. More precisely, we build upon the work of [7], in which an innovative time-varying cache indexing scheme called *dynamic indexing* was proposed. By modifying the cache indexing function over time, it is possible to achieve ideal (i.e., uniform) distribution of accesses to the cache lines. This implies that every cache line can be put into a low-power state (through voltage scaling or power gating) for the same amount of time so that all cache lines will become unreliable at the same time.

This technique achieves optimal results with minimal overhead; however, it is suitable for situations where the modification of the internal structure of a cache is allowed; in many cases, however, this is not feasible. SRAM memories are normally very highly-optimized designs, and in custom design flows memories are obtained from a memory compiler that generates standard memory structures.

In this work we propose an implementation of dynamic indexing suitable for standard caches, which is based on the idea of partitioning a memory into multiple banks in order to maximize the potential for power management [8], [9], [10]. Unlike these works, however, in which the banks have non-uniform sizes and are calculated by profiling of the application code, in our approach all sub-blocks have the same size (a power of two) in order to simplify the hardware complexity of "remapping" one sub-block onto another. Another difference is that our scheme inherits from dynamic indexing the property of being general-purpose: the same architecture is applicable to any workload and does not require any customization or profiling.

Results show that a time-varying reindexing allows to significantly improve the lifetime of power-managed caches: our scheme provides average aging improvements between 22% (for the worst configuration) and 2x (for the best one) with respect to a monolithic cache, compared to a mere 9% improvement obtained with a conventional power-managed cache architecture.

II. Background and Related Work

A. Background

For an in-depth analysis of NBTI effects and models we refer the reader to classical tutorial papers on NBTI (e.g., [1]). We summarize here the basic issues involved in NBTI-induced aging in SRAM cells and arrays.

Due to the symmetric structure of a cell, a SRAM cell ages in fact whatever the value it stores; therefore the effect of the dependency on logic values is immaterial in a memory cell. The best-case degradation occurs when both PMOS exhibit the same amount of degradation, that is when the cells stores a 0 and a 1 with equal probability [11].

Another important aspect is that the aging of the two inverters in the bitcell does not truly affects the *delay* of the cell. Rather, it impacts its stability. A conventionally accepted metric for the aging of a SRAM cell is the Static Noise Margin (SNM), defined as the minimum DC noise voltage necessary to change the state of an SRAM cell; when the SNM of a cell falls below a threshold that allows safe storage of data it cannot be safely read or written. This threshold strongly depends on the technology and the specific design of the memory cell (e.g., transistor W/L ratios).

B. Related Work

Techniques for the *reduction* of NBTI effects follow two main approaches: (i) *compensating* solutions, where the circuit is designed with a tighter delay constraint so that the aging will still be within the original timing constraint after some time; (ii) *mitigating* solutions, which act directly on the variables that affect NBTI aging: V_{th} and/or V_{dd}, gate sizes, and signal probabilities [4], [12].

A distinct class of solution that combines power (static, in particular) and aging reduction interacts with the power states of the circuit under analysis. These schemes exploit the availability of a "sleep" signal that indicates when the circuit is entering the standby state. The low-power state can be exploited either by using a sort of "gated" version of standard library cells thus allowing to minimize the number of logic 0's in the circuit [13], or by using special vectors to be applied during standby [14].

Concerning the mitigation of NBTI effects in SRAMs, most approaches attempt at structurally or functionally maximazing the conditions under which the degradation in a memory cell does not occur or it is minimal. One first approach was proposed in [11]: since a 50% probability of storing a value provides minimum aging, they provide hardware and software schemes to periodically invert the entire content of a memory so as to guarantee a perfectly balanced probability. A similar idea was proposed by [15], yet at a word granularity and with a much shorter inversion frequency (thousands of cycles). A *flip signal* determines whether values are to be read or written in inverted form; each memory word has a *flip bit* which is copied from the flip signal upon access.

The method of [16] proposes a new memory cell structure consisting of a set of NAND gates arranged in such a way that minimum degradation ratio for all PMOS transistors in the cell can be obtained.

Another solution called *recovery boosting* [18] allows both pMOS devices in the memory cell to be put into the recovery mode by raising the ground voltage and bitlines to the nominal voltage through modification of each memory cell.

A different class of solutions is based on the exploitation of the above mentioned benefit provided by low-power states. In [17], the authors assess the aging benefits provided by the application of power gating to a memory cell, observing that it has a much higher impact than controlling the value probability. Benefits at the architectural level on entire memory blocks of power management solutions (both based on DVS and power-gating) were evaluated in [5], [6].

The work of [7] proposes a *dynamic indexing* scheme in which the cache indexing function is modified over time in order to achieve an uniform distribution of idleness over the cache lines; in this way all the leakage savings opportunities can also be used for aging reduction. This work yields optimal results (i.e., all cache lines have identical lifetime), but adopts the architectural template of many popular cache leakage optimization solutions (e.g., [19], [20]), in which a cache line is the unit of power management. This choice implies the requirement that the *internal* cache structure should be modified, which is not always feasible, e.g. in the case standard design flows using memory blocks obtaned by memory compilers.

III. Aging-Aware Cache Partitioning

The architecture we propose in this paper can be viewed as a coarse-grain implementation of the scheme of [7]. The choice of granularity is dictated by the possibility of using standard memory blocks generated automatically by memory compilers. In practice, we implement a multi-banked cache as done in [8]–[10]; unlike these approaches, however, which use blocks of variable size, we use banks with uniform sizes. This choice has three advantages: first, all decoding operations are much simpler in hardware; second, it allows us to implement a general-purpose, application-independent architecture (all the above mentioned architectures are application-specific and require profiling of the application). Third, no degradation of miss rate is experienced. Since idleness is distributed over the various sub-blocks, all blocks will fail approximately at the same time and the cache will work as the non-partitioned one until the failing time is reached.

On the other hand, restricting the sizes of the blocks to pre-determined sizes can exploit only partially the existing idleness; therefore, both power and aging reductions will be lower than those achievable with non-uniform partitions.

A. Coarse-Grain Dynamic Indexing

1) Preliminaries: Let us assume a direct-mapped cache with $L = 2^n$ lines $(l_0, \ldots, l_{L-1})$; n is the number of the index bits of the cache address. We partition the cache into M blocks $B_0, \ldots, B_{M-1}$, where $M = 2^p$ is a power of two for obvious practical reasons. Each block will contain exactly 2^{n-p} lines. M will be in general a small number (≤ 16 in our experiments) because an arbitrarily fine partitioning will excessively increase the wiring overhead. Further discussion on the partitioning overhead will be discussed in the experimental section.

We assume each block can be turned into a low-power state in which dynamic and/or leakage power is reduced. In our work, such a low-power state is implemented by reducing the supply voltage as described in [10]: as a matter of fact, this is the only viable choice for standard memory blocks provided by memory compilers; implementing power-gating on a standard block would require accessing the internals of the memory [10]. Furthermore, using voltage scaling allows to preserve the contents of the memory block in the standby state; although a non state-preserving option would be feasible in caches (lost values could be retrieved in farther levels of the memory hierarchy), the results of [7] have shown that a voltage-scaled

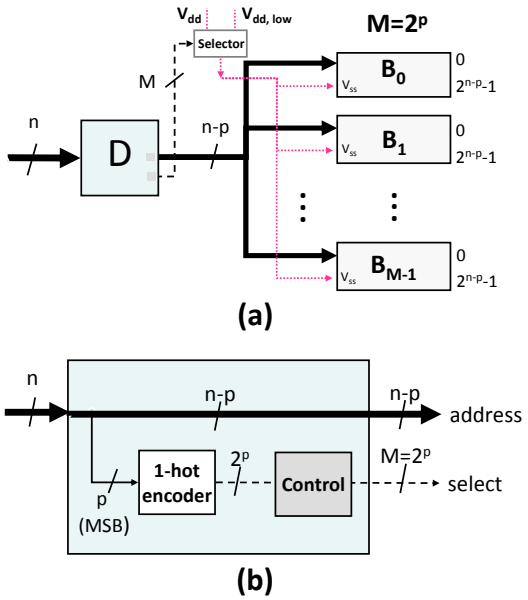

Fig. 1. M-Block Uniformly Partitioned Cache (a) and Structure of Decoder D (b).

implementation has better power/delay characteristics due to lower state transition overhead.

Figure 1-(a) shows a conceptual block diagram of a M-block partitioned cache; solid lines represent address lines, dashed ones are "activation" signals, and dotted lines denote power supply signals. Turning off a block is done by asserting its activation signal `select`, which in our architecture corresponds to the selection of the $V_{dd,low}$ supply voltage.

The block labeled **D** (Figure 1-(b)) implements these two operations: remapping the address on the proper block and asserting the `select` activation signals for the M blocks.

Address signals to each block are simply derived by routing the $n-p$ LSBs of the address to each sub-block. Activation signals are obtained by taking the p MSBs and transforming them into a 1-hot code (block **1-hot encoder**) onto 2^p bits (e.g., Bank 0 corresponds to the M-bit encoding $00\ldots1$, Bank $M-1$ corresponds to $100\ldots0$). Block **Control** implements the block de-activation mechanism. As done in similar implementation, the decision is based on its access frequency: if a block is not accessed for some number of cycles it is turned into a low-power state. This threshold, known as the *breakeven time* in the power management terminology, is non-zero because turning on and off a block has some non-zero overhead. The value of the breakeven time depends essentially on (i) the size of the block to be turned off, and (ii) the ratio between the energy spent in the off and in the on state. Therefore, the decision implemented by Block **Control** is: *turn a block into a low-power state. if it is not accessed for a number of cycles greater than the breakeven time.*

In order to realize this, Block **Control** contains M counters which are incremented upon a non-access (a 0 on the 1-hot encoded signal), and reset upon an access (a 1 on the 1-hot signal). When a counter saturates, its terminal count signal (1 if the counter saturates) is used as the output selection signal. The width of the counter obviously depends on the value of the breakeven time: in our case is in the order of a few tens of cycles (it clearly depends on M). Therefore, 5- or 6-bit counter suffice. Finally, Block **Selector** drives the correct value of supply voltage (V_{dd} or $V_{dd,low}$) to each block according to the encoding on the `select` signals.

Notice that the performance overhead of this encoder is negligible; the longest combinational input/output delay in the **1-hot encoder** goes through a single logic gate corresponding to the binary encoding of the corresponding minterm.

2) Motivation: The architecture of Figure 1 will save dynamic and leakage power because, thanks to the locality of accesses, it is quite common that one or more blocks are idle for a significant amount of time and can then be turned into a low-power state. We define a compact metric to measure the energy saving potential, i.e., the *useful idleness* of a block. This is defined as the percentage of idle intervals of a block that are longer than its breakeven time. Let $(I_0,\ldots I_{M-1})$ be these values. The various $I_j, j=0,\ldots,M-1$ will have in general quite different values. For power saving purposes, this is not so relevant since the total power saving is related to the average idleness. For aging, however, it is the worst-case idleness that matters; thus, if one block has very little idleness, it will fail before the others. Table I shows, for a $M=4$ partition the worst-case idleness of each block, for the benchmarks used in our simulations. Column *Average* is the average idleness over the four banks, and is a measure of the achievable power saving.

	I_0	I_1	I_2	I_3	**Average**
`adpcm.dec`	2.46%	99.98%	99.98%	3.75%	51.54%
`cjpeg`	22.64%	53.24%	59.37%	9.51%	36.19%
`CRC32`	18.54%	2.19%	44.38%	2.88%	16.99%
`dijkstra`	12.06%	18.55%	50.65%	56.28%	34.38%
`djpeg`	67.66%	29.23%	27.89%	24.97%	37.44%
`fft_1`	49.35%	48.34%	61.32%	9.12%	42.03%
`fft_2`	54.78%	51.82%	58.03%	6.96%	42.90%
`gsmd`	6.92%	90.81%	92.82%	0.40%	47.74%
`gsme`	49.17%	72.88%	89.34%	0.37%	52.94%
`ispell`	66.36%	55.63%	44.82%	21.04%	46.96%
`lame`	58.78%	32.94%	38.62%	13.74%	36.02%
`mad`	37.25%	48.74%	34.00%	28.10%	37.02%
`rijndael_i`	82.35%	31.72%	22.61%	3.71%	35.10%
`rijndael_o`	20.59%	19.45%	91.78%	3.63%	33.86%
`say`	88.53%	85.51%	26.59%	12.42%	53.26%
`search`	66.57%	23.43%	48.00%	57.78%	48.95%
`sha`	4.91%	98.62%	94.09%	3.13%	50.19%
`tiff2bw`	33.88%	17.43%	67.38%	70.49%	47.29%
Average					**41.71%**

TABLE I
DISTRIBUTION OF IDLENESS IN A 4-BANK CACHE.

We notice that in many cases the differences in the I_i's are quite significant; as an example, in benchmark `adpcm.dec` Banks 0 and 3 have very low idleness ($< 4\%$), whereas Banks 1 and 2 can be virtually put to sleep all the time (99.98%). The two banks with very low idleness basically nullify the chance of exploiting idleness for aging reduction. Conversely, the average idleness is sizeable (more than 51%), allowing significant power savings (about 42% on average over all the benchmarks). Similar consideration apply to most of the benchmarks.

A possible solution to this problem could be that of implementing some form of graceful, stepwise management of the aging of the cache. For instance, we could progressively disable cache sub-blocks that become progressively unusable. However, such an architecture has several drawbacks. First, as cache blocks are progressively disabled, the application will use a progressively smaller cache, with consequences on the overall performance of the program. Moreover, such a scheme relies on the availability of some sort of aging "detector",

which can warn (and disable) the architecture that a given block cannot be used reliably.
For these reasons, we use another approach which tries to *uniformly distributing the idleness over the blocks*.
3) Implementation: One way of uniformly distributing idleness could be that of *changing the mapping of the addresses to the various blocks as time progresses*. Let us first describe the conceptual operations of this time-varying indexing.
Consider address i (cache index value), and suppose that (at time 0) this address maps to the k-th line in Block j; In order to distributed the accesses, after some time the mapping function will change and the same address i will map to the the k-th line of another block. Which block is used depends on the chosen indexing policy.
Figure 2 shows the modifications required to the decoder **D** of Figure 1-(b); they consist of the addition of an extra block before the one-hot encoder; block **f()** implements the dynamic indexing, that is, it modifies the index value whenever the signal `update` is triggered.

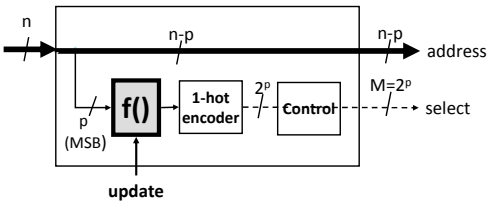

Fig. 2. Generic Dynamic Indexing Architecture.

Two are the issues to be addressed for a precise definition of the implementation of this architecture: the choice of functions **f()**, and the issue of the updating mechanism.
Choice of Indexing: The problem of spreading a value uniformly over a range shares many similarities with other engineering and computer science problems such as uniform hashing, cryptographic codes, or scrambling in coding theory. Although the problem is not new and many solutions are available, in our context we must privilege simplicity of implementation since every cache access undergoes this transformation and it affects the cache access time.
We borrow some of the schemes proposed in [7], where re-indexing was done on the entire set of cache index bits. Here, as shown in Figure 2, the re-indexing only involves the p MSBs of the address. We experimented with two schemes:

- *Probing*, which mimic the *linear probing* used in open addressing in hash tables (Figure 3-(a));
- *Scrambling*, which mimic the de-correlation operation used to minimize conflict misses in caches [21] (Figure 3-(b)).

The *Probing* scheme implements the re-mapping of lines of Bank i to Bank $i+1$ (modulo M). At time 0, address i maps to the $(i \mod 2^{n-p}-1)$-th line of Bank $(i/2^{n-p}-1)$. Therefore, upon receiving the first update, address i will map to the same $(i \mod 2^{n-p}-1)$-th line, this time in Bank $(i/2^{n-p}-1+\mathbf{1})$ After R updates intervals address i will be mapped to $(i \mod 2^{n-p}-1)$-th line of Bank $(i/(2^{n-p}-1+\mathbf{R}) \mod M)$.
Example 1: Assume $N = 256$ lines partitioned into $M = 4$ banks. Each block has $N/M = 64$ lines. Consider address $i = 70$. At time 0 this will correspond to line $70 \mod 63 = 7$ of Bank $70/7 = 1$. Upon first update, line 70 will map to line 7 of Bank $70/7 + 1 = 2$; upon second update, line 70 will map to line 7 of Bank 3; upon third update, line 70 will map to line 7 of Bank 0.
In hardware, *Probing* can simply be implemented by summing the p-bit bank address and a value incremented by the update signal. Modulo M operations are automatically achieved by restricting all signals to p bits. (Figure 3-(a)).

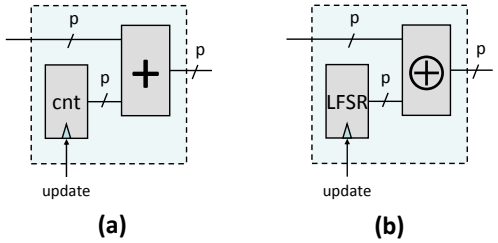

Fig. 3. Probing (a) and Scrambling Architectures (b).

The *Scrambling* scheme is even simpler to describe. When the `update` signal is triggered, these scheme maps bank address i to a randomly chosen bank $(0, \ldots, M-1)$. Uniform distribution properties of the random generator guarantees that a quasi-uniform distribution of idleness is obtained. In hardware (Figure 3-(a)), this can be implemented by bitwise XOR-ing the p-bit bank address with a randomly generated number (e.g., by means of a LFSR).
Concerning the effectiveness of these schemes in uniformly distributing the idleness, in [7] it was proven that a *Probing* scheme with an increment of 1 (as proposed in this work) provides perfectly uniform distribution of values, provided that a number of updates greater than or equal to the number of slots is executed. Since M is in our case quite small, uniformity will be achieved very quickly.
Concerning *Scrambling*, it can asymptotically approach the quality of *Probing*. The speed of this convergence depends on the number of repeated values in the RNG (the LFSR in our work) [7].
Updating the Indexing: It is obvious that every time the indexing is updated through the `update` signal, the entire cache content becomes unusable and a cache flush is required. While this appears as a potentially critical issue, in practice it is not so.
Updates can in fact be activated with a very low frequency (e.g., once a day or even less frequently) given the typical time horizons of aging (i.e., years). Furthermore cache flushes occur regularly in cache (e.g., on a context switch); we can thus simply associate the update event to any cache flush occurring in the system. This results in a true zero energy overhead, but for the small amount of energy dissipated by f.

IV. Experimental Results

A. Experimental Setup

The proposed methodology has been implemented and tested on a set of traces extracted from the simulation of the MediaBench suite [22] with in-house cache simulator. The latter has been augmented with aging and power/energy models derived from an industrial 45nm design kit provided by STMicroelectronics.
As described in Section II, a commonly accepted metric for SRAMs aging is given by the degradation of SNM vs. time. We define the lifetime of a memory cell as the time after which the SNM has decreased by more than 20%. However, since such SNM data are not integrated into standard design flows, we implemented a dedicated SPICE-based characterization

framework which predicts, under user-defined PVT operating conditions, the aging profile of a 6T-SRAM cell. Such a framework uses physical characteristics of the cell (netlist, size of the transistors, process parameters) as well as functional information (the probability, p_0, to store a '0' logic, and the idleness, P_{sleep}, of the cell) and performs the evaluation in two phases: the *pre-stress* and the *post-stress* simulation.
In the first phase, the aging of the pMOS transistors is computed on the base of HSPICE built-in aging models, fitted to the technology parameters provided by silicon vendor. The aging information sampled during pre-stress simulation are then translated into device parameter degradation (i.e., threshold voltage degradation ΔV_{th}) and annotated into the SRAM cell netlist as DC-controlled voltage sources on the gate terminal of each pMOS transistor [23].
After the netlist has been annotated, the post-stress simulation yields the values of SNM. Specifically we refer to the *read* SNM, (i.e., when the cell operates with access nMOS transistors on), which represents the worst case condition for aging [23]. By comparing the pre-stress and post-stress SNMs, the aging curves are profiled and the lifetime of the cell calculated. The collected data are stored in a lookup table, which is used by the cache simulator to estimate the aging of the cache banks, and thus, of the entire cache.

B. Simulation Results

1) Impact of Cache Parameters: Table II shows energy and aging results for a power-managed cache split in $M = 4$ blocks. We first report the energy saving, with respect to a non-partitioned cache, and the lifetime (in years) for 8kB, 16kB and 32kB caches (with 16-byte lines) without re-indexing (LT_0) and when re-indexing is applied (LT). In the used technology, the lifetime of a standard memory cell is 2.93 years.

	8kB			16kB			32kB		
	E_{sav} [%]	LT_0 [yrs]	LT [yrs]	E_{sav} [%]	LT_0 [yrs]	LT [yrs]	E_{sav} [%]	LT_0 [yrs]	LT [yrs]
adpcm.dec	30.6	2.98	4.82	43.8	3.04	3.76	55.7	3.04	4.03
cjpeg	31.5	3.18	4.07	44.0	3.17	4.32	55.6	3.11	4.75
CRC32	33.3	2.98	3.40	45.0	2.93	3.88	56.1	2.93	4.00
dijkstra	31.2	3.26	3.99	44.4	3.31	4.31	55.5	3.29	3.99
djpeg	32.2	3.61	4.12	44.2	3.36	4.02	55.2	3.52	4.35
fft_1	32.2	3.17	4.30	44.2	2.96	4.46	55.6	3.24	4.44
fft_2	32.2	3.11	4.34	44.2	2.97	4.42	55.6	3.18	4.40
gsmd	31.3	2.94	4.59	44.2	3.08	3.81	55.2	3.03	5.10
gsme	31.5	2.94	4.90	43.9	2.94	4.50	55.1	3.03	4.37
ispell	33.6	3.50	4.55	45.2	3.40	4.74	55.9	3.42	4.75
lame	32.1	3.31	4.06	44.4	3.55	4.12	55.7	3.33	4.49
mad	32.1	3.73	4.10	43.7	3.74	4.76	55.0	3.72	4.59
rijndael_i	32.9	3.02	4.02	44.4	3.11	4.10	55.0	3.26	4.90
rijndael_o	33.1	3.01	3.96	44.4	3.13	4.16	55.2	2.96	5.23
say	31.9	3.27	4.92	43.9	3.06	5.09	55.4	3.38	4.43
search	33.4	3.57	4.67	45.3	3.58	4.27	56.1	3.07	4.24
sha	31.1	3.00	4.74	43.6	3.03	4.48	55.0	3.02	6.09
tiff2bw	33.4	3.41	4.57	44.7	3.13	4.31	55.6	3.09	4.98
Average	**32.2**	**3.22**	**4.34**	**44.3**	**3.19**	**4.31**	**55.5**	**3.20**	**4.62**

TABLE II
ENERGY SAVINGS AND LIFETIME WHEN VARYING CACHE SIZE (LINE SIZE IS 16 BYTES).

The energy savings (that are independent of the re-indexing strategy) show how the available idleness provides chances of putting the cache blocks in low-power state. Increasing the cache size causes higher energy savings, since for the technology we adopted, larger memory blocks have a higher ration of static over dynamic energy consumption. In addition, running a given application (i.e., a fixed number of accesses) on a larger cache implies a distribution of the accesses over a large space. Therefore, larger blocks gain more benefit from power management (ranging from 32.2% for the 8KB cache to 55.5% for the 32KB cache).
The idleness, and therefore the time spent in the low-power state, conversely, is not directly impacted by the cache size, since it depends on the idleness distribution over the cache lines, that is strongly application dependent. Hence, the lifetime extension, when enlarging the cache, does not show a clear trend. Considering the average on all the benchmarks, however, such unpredictability tends to disappear, showing that the cache size has a limited impact on the lifetime of a power managed cache.
To assess the benefit of the re-indexing scheme, we can notice that just applying the power management yields a small benefit on the lifetime; if we consider for instance the case of the 8KB cache, the extension is from 2.93 years (lifetime of monolithic cache) to 3.22 years, on average (a modest 9% of lifetime extension). The impact of the re-indexing is evident from Column LT: balancing the idleness among blocks provides a further 38% of lifetime extension with respect to regular power managed caches, on average. This corresponds to an average 48% lifetime extension for the 8KB cache, 47.1 for the 16KB cache, and 57.6% for the 32KB cache. In some cases such a benefit is much larger, as for sha where we obtain a 2x lifetime extension.
Another set of results concerns the effects of cache line size. Table III shows the results; columns have the same meaning as in Table II.

	LS=16B		LS=32B	
	E_{sav} [%]	LT [yrs]	E_{sav} [%]	LT [yrs]
adpcm.dec	43.8	3.76	31.0	3.61
cjpeg	44.0	4.32	31.20	4.26
CRC32	45.0	3.88	33.5	3.82
dijkstra	44.4	4.31	31.0	4.17
djpeg	44.2	4.02	31.7	3.95
fft_1	44.2	4.46	31.9	4.38
fft_2	44.2	4.42	31.9	4.35
gsmd	44.2	3.81	31.6	3.71
gsme	43.9	4.50	31.7	4.46
ispell	45.2	4.74	33.3	4.66
lame	44.4	4.12	32.1	4.07
mad	43.7	4.76	31.2	4.66
rijndael_i	44.4	4.10	31.6	3.99
rijndael_o	44.4	4.16	31.6	4.03
say	43.9	5.09	31.4	5.05
search	45.3	4.27	33.1	4.17
sha	43.6	4.48	31.20	4.47
tiff2bw	44.8	4.31	33.0	4.32
Average	**44.3**	**4.31**	**31.9**	**4.23**

TABLE III
ENERGY SAVINGS AND LIFETIME WHEN VARYING LINE SIZE (CACHE SIZE IS 16 KB).

For lifetime the same considerations about the dependency on cache size do hold: enlarging the line size has a negligible impact on idleness: for a 4-block, 16kB cache, the average idleness over the four blocks is 41% and 40% for 16 bytes and 32 bytes line size, respectively, resulting in lifetimes of 4.31 and 4.23 years (as reported in the table).
The energy saving, conversely, gets smaller as line size increases because it implies larger tag arrays. Tag arrays have fewer bits than the data array, and have a larger reactivation penalty. Therefore, as tag bits increase, the energy share of tags becomes more relevant, and turning a cache block in the low-power state must be compensated by more long idle intervals. Hence, the same average idleness provides a smaller energy benefit.

2) Impact of Re-indexing Policy: The *Probing* and *Scrambling* schemes described in Section III-A3 have comparable efficiency. The *Probing* scheme is in fact optimal by construction [7], while the results of *Scrambling* depend on the quality of the random number generator (RNG), specifically the number of repeated values.

Consider a RNG that generates N numbers in the range $[0, \ldots, M-1]$. Ideally, each of the M possible values repeats itself N/M times (notice that in our context, $N \gg M$). The quality of the RNG is related to the ratio between the actual rate of repetition and the ideal value N/M, which we call the *error* of the RNG.

For a uniformly distributed generator, it can be shown that the error in reshaping is inversely proportional to $\sqrt{N}$. Since N is very large during the lifetime span of the cache, the sub-optimality of the *Scrambling* scheme is indeed negligible.

The net result is that, in spite of the different theoretical properties, *Probing* and *Scrambling* provide de facto identical results. The choice between them is only driven by considerations about their implementation on the available technology.

3) Impact of Number of Banks: From the architectural point of view, the most important dimension in the exploration is the analysis of the granularity of the partitioning.

Size	2 blocks		4 blocks		8 blocks	
	Idleness [%]	*LT* [yrs]	*Idleness* [%]	*LT* [yrs]	*Idleness* [%]	*LT* [yrs]
8kB	15	3.34	42	4.34	58	5.30
16kB	15	3.35	41	4.31	64	5.69
32kB	25	3.68	47	4.62	68	5.98

TABLE IV
AVERAGE IDLENESS AND LIFETIME WHEN VARYING CACHE SIZE AND NUMBER OF BLOCKS.

Table IV show the results for the case of $M = 2, 4$, and 8 blocks and for the cache sizes of 8KB, 16KB, and 32KB. For each entry, both the percentage of idleness (average over the M blocks) and lifetime (LT) are reported. As intuitively we expect, increasing M results in a higher idleness and, as a consequence, in longer lifetime. Specifically, for $M = 8$ the lifetime of the cache is increased by about 2x, while partitioning the cache in only 2 block, yields no more than a 26% of lifetime extension, for the best case condition.

Clearly, it is not possible to arbitrarily decrease the granularity of the partitioning: increasing the number of blocks implies more overhead, in particular wiring. The address and data bus, as well as the control signals must be routed to all the blocks. This also translates into area overhead. Previous works on memory partitioning ([8], [10]) indicated that partitioning into more than 4/5 blocks consumes all the energy saved thanks to partitioning. In our case, however, placing blocks of identical sizes in the floorplan is easier than placing non-uniform (and often very different) shapes. Therefore, we consider feasible the partitioning into up to $M = 16$ blocks.

Energy figures reported in the table accounts also for this overhead, characterized from the data reported in [10].

V. CONCLUSIONS

Partitioned caches, which have proven an effective and low-overhead solution for reducing dynamic and static power, can also be leverage for reducing the aging caused by NBTI.

Due to the different nature of the two metrics, however, (power is cumulative, aging is a worst-case quantity), an appropriate solution for uniformly distributing cache accesses is need to exploit the resulting idleness.

Our architecture, which does not require the internal modification of caches, provides average aging improvements between 22% (for the worst configuration) and 2x (for the best one) with respect to a monolithic cache.

REFERENCES

[1] M.A.Alam, "Reliability- and process-variation aware design of integrated circuits," *Microelectronics Reliability*, Vol. 48, No. 8, August 2008, pp. 1114–1122.

[2] L. Zhang, R. P. Dick, "Scheduled Voltage Scaling for Increasing Lifetime in the Presence of NBTI," *ASPDAC'09*, pp. 492–497, Jan. 2009.

[3] A. Calimera, E. Macii, M. Poncino, "NBTI-Aware Power Gating for Concurrent Leakage and Aging Optimization", *ISLPED '09: International Symposium on Low power Electronics and Design*, pp. 127–132, August 2009.

[4] R. Vattikonda, et.al. "Modeling and minimization of PMOS NBTI effect for robust nanometer design," *DAC-44*, pp. 1047–1052, 2006.

[5] A. Ricketts, J. Singh., K. Ramakrishnan, N. Vijaykrishnan, D. K. Pradhan. "Investigating the Impact of NBTI on Different Power Saving Cache Strategies," *DATE'10: Design, Automation and Test in Europe*, pp. 592–597, March 2010.

[6] A. Calimera, M. Loghi, E. Macii, M. Poncino, "Aging Effects of Leakage Optimizations for Caches," *GLSVLSI'10: IEEE Great Lakes Symposium on VLSI*, pp. 95–98, May 2010.

[7] A. Calimera, M. Loghi, E. Macii, M. Poncino, "Dynamic Indexing: Concurrent Leakage and Aging Optimization for Caches", *ISLPED '10: International Symposium on Low power Electronics and Design*, pp. 343–348, August 2010.

[8] L. Benini, L. Macchiarulo, A. Macii, E. Macii, M. Poncino, "Layout-Driven Memory Synthesis for Embedded Systems-on-Chip," *IEEE Transactions on VLSI Systems*, Vol. 10, No. 2, pp. 96–105, April 2002.

[9] O. Ozturk, M. Kandemir, "Nonuniform Banking for Reducing Memory Energy Consumption," *DATE'05: Design, Automation and Test in Europe*, pp. 814–819, Mar. 2005.

[10] M. Loghi, O. Golubeva, E. Macii, M. Poncino, "Architectural Leakage Power Minimization of Scratchpad Memories by Application-Driven Subbanking," *IEEE Transactions on Computers*, Vol. 59, No. 7, pp. 891–904, July 2010.

[11] S.V. Kumar, K.H. Kim, S.S Sapatnekar, "Impact of NBTI on SRAM read stability and design for reliability," *ISQED'06*, March 2006, pp. 213–218.

[12] S. V. Kumar, et al., "NBTI-Aware Synthesis of Digital Circuits," *DAC-45*, pp. 370–375, June 2007.

[13] Y. Wang et al., "Gate replacement techniques for simultaneous leakage and aging optimization," *DATE'09: Design Automation and Test in Europe*, pp. 328–333, March 2009.

[14] Y. Wang et al., "On the efficacy of input Vector Control to mitigate NBTI effects and leakage power," *ISQED'09: International Symposium on Quality of Electronic Design*, pp. 19–26, March 2009.

[15] Y. Kunitake, T. Sato, H. Yasuura, "A case study of Short Term Cell-Flipping technique for mitigating NBTI degradation on cache," *ISQED'10: International Symposium on Quality Electronic Design*, pp. 660–666, March 2010.

[16] J. Abella, X. Vera, O. Unsal and A. González, "NBTI-Resilient Memory Cells with NAND Gates for Highly-Ported Structures", *Workshop on Dependable and Secure Nanocomputing*, June 2007.

[17] A. Calimera, E. Macii, M. Poncino, "Analysis of NBTI-induced SNM degradation in power-gated SRAM cells," *ISCAS'10: International Symposium on Circuits and Systems*, pp. 785–788, May 2010.

[18] T. Siddiqua, S. Gurumurthi, "Recovery Boosting: A Technique to Enhance NBTI Recovery in SRAM Arrays," *ISVLSI'10: IEEE Annual Symposium on VLSI*, July 2010.

[19] M. Powell, et al. "Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories," *ISLPED'00: International Symposium on Low power Electronics and Design*, July 2000, pp. 90–95.

[20] K. Flautner, N. Kim, S. Martin, D. Blaauw, T. Mudge, "Drowsy caches: Simple techniques for reducing leakage power," *ISCA'02: International Symposium on Computer Architecture*, May 2002, pp. 148–157.

[21] A. Gonzalez, et al., "Eliminating cache conflict misses through XOR-based placement functions," *International Conference on Supercomputing*, pages 76–83, July 1997.

[22] M. R. Guthaus et al., "MiBench: A free, commercially representative embedded benchmark suite", *IEEE 4th Annual Workshop on Workload Characterization*, pp. 3–14, Dec. 2001.

[23] K.Kang, H. Kufluoglu, K. Roy, M.A. Alam, "Impact of Negative-Bias Temperature Instability in Nanoscale SRAM Array: Modeling and Analysis," *IEEE Transactions on CAD*, Vol. 26, No. 10, pp. 1770–1781, Oct. 2008.