An Aging-Aware GPU Register File Design Based on Data Redundancy

Alejandro Valero, Francisco Candel, Darío Suárez-Gracia, *Member, IEEE,* Salvador Petit, *Member, IEEE,* and Julio Sahuquillo, *Member, IEEE*

Abstract—Nowadays, GPUs sit at the forefront of high-performance computing thanks to their massive computational capabilities. Internally, thousands of functional units, architected to be fed by large register files, fuel such a performance. At deep nanometer technologies, the SRAM memory cells that implement GPU register files are very sensitive to the Negative Bias Temperature Instability (NBTI) effect. NBTI ages cell transistors by degrading their threshold voltage V_{th} over the lifetime of the GPU. This degradation, which manifests when a cell keeps the same logic value for a relatively long period of time, compromises the cell read stability and increases the transistor switching delay, which can lead to wrong read values and eventually exceed the processor cycle time, respectively, so resulting in faulty operation. This work proposes architectural mechanisms leveraging the redundancy of the data stored in GPU register files to attack NBTI aging. The proposed mechanisms are based on data compression, power gating, and register address rotation techniques. All these mechanisms working together balance the distribution of logic values stored in the cells along the execution time, reducing both the overall V_{th} degradation and the increase in the transistor switching delays. Experimental results show that a conventional GPU register file suffers the worst case for NBTI, since a significant fraction of the cells maintain the same logic value during the entire application execution (i.e., a 100% '0' and '1' duty cycle distributions). On average, the proposal reduces these distributions by 58% and 68%, respectively, which translates into V_{th} degradation savings by 54% and 62%, respectively.

Index Terms—Data compression, duty cycle, GPU architectures, NBTI, register files, threshold voltage degradation.

1 Introduction

The role GPUs play in high-performance computing is growing in importance due to their excellent performance per watt compared to conventional processors [13]. For instance, the most energy-efficient supercomputers in the world, ranked in the Green500 list [2], include GPU devices.

GPUs are designed for improving system throughput, and their design is aimed at exploiting Thread Level Parallelism (TLP) by supporting the concurrent execution of a vast number of threads. The number of threads that a GPU can simultaneously execute exceeds, in several orders of magnitude, the number of hardware contexts supported by advanced processors like the IBM Power9 [32] or the Intel Knights Landing [33]. This feature is especially important for the execution of parallel scientific applications that rely on a high number of threads.

In this regard, GPUs have dramatically evolved during the last years to support massive numbers of threads, which implies that they must incorporate huge register files to feed the computation performed by these threads. For example, the NVIDIA Tesla P100 (Pascal GP100) GPU features a 14MB register file, 3.5 times larger than its shared L2 cache (4MB) and several orders of magnitude bigger than typical CPU register files or CPU private L1 and L2 caches.

On the other hand, technology advances are allowing the semiconductor industry to implement fabrication nodes whose size is so small that process variations threaten system reliability. Process variations make transistors less reliable in low-power modes and intensify transistor aging phenomena, which affects modern computing devices, especially those that implement a large number of transistors, such as GPUs.

This work focuses on attacking aging in those transistors implementing the SRAM memory cells of GPU register files. In particular, the effect that most accelerates aging in SRAM cells is known as Negative Bias Temperature Instability (NBTI). NBTI degrades the threshold voltage V_{th} of the PMOS transistors that are on (i.e., their gate is connected to a logic '0'). In an SRAM cell, this happens when a logic value is stored for a relatively long period of time, known as duty cycle. In turn, the V_{th} degradation, or simply dV_{th}, compromises the cell read stability, which is measured as Static Noise Margin (SNM), and can lead to wrong read values. In addition, the dV_{th} slows down the transistor switching delay T_s, and, since this effect can affect several transistors along the critical path of a digital circuit, it can cause operation faults if the critical path delay exceeds the clock cycle. Overall, mitigating the dV_{th} results in a reduction of the SNM degradation and T_s. In fact, both SNM and T_s closely follow the trend of the dV_{th} with a near constant scale factor throughout the whole device lifetime [27], [37], [17].

To guarantee the cell read stability, designers incorporate additional transistors to the typical 6T cell, resulting in 7T and 8T cells with area overheads of 13% and 30%, respectively, compared to the 6T cell [35]. On the other hand, to avoid the increase of the critical path delay, designers include guardbands (i.e., lower operating frequencies) [37], [24]. As a consequence, the maximum frequency for a given device decreases with time [8]. For instance, for a 45nm technology node, NBTI can cause a 25% performance degradation after 3 years [12]. Moreover, as technology scales down and aging intensifies, NBTI is becoming the principal

- *A. Valero and D. Suárez-Gracia are with the Departmento de Informática e Ingeniería de Sistemas, Instituto Universitario de Ingeniería de Aragón, Universidad de Zaragoza, Spain. E-mails: {alvabre, dario}@unizar.es.*
- *F. Candel, S. Petit, and J. Sahuquillo are with the Department of Computer Engineering, Universitat Politècnica de València, Spain. E-mails: fracanma@inf.upv.es, {spetit, jsahuqui}@disca.upv.es.*

source of transistor degradation [21], [36], compromising not only performance, but power consumption and area due to enlarged cell designs and transistor design margins required to compensate the NBTI effect [10].

A straightforward strategy for coping with transistor aging is powering off memory cells. Fortunately, when turned off, transistors not only stop degrading, but they partially recover from the NBTI effect [19]. Therefore, some works have attacked NBTI in GPUs by switching off those memory structures that are not used by GPU applications (kernels) [27], [12] or have severely aged [21]. Other works propose kernel compilation techniques based on aging information [24].

On the other hand, data redundancy has been used in the past to improve performance [7] and to save energy [22]. However, to the best of our knowledge, this is the first time that it is used in the GPU register file to mitigate the NBTI effect. This work makes two main contributions:

- A data compression/decompression mechanism, inspired by the Base-Delta-Immediate (BDI) algorithm [29], which identifies redundant data patterns and allows whole compressible registers to be switched off.
- A mechanism that rotates physical register addresses with the aim of evenly distribute switch-off cycles among all the registers.

These contributions can be applied to any modern GPU architecture either from NVIDIA or AMD, since i) both vendors implement huge register files with a similar organization and ii) the leveraged data redundancy comes from the single-instruction multiple-thread programming model provided by CUDA and OpenCL; thus, it is independent of GPU architectural details.

This paper extends the work in [11] in four main ways: i) a new data pattern has been identified for 2D kernels, and the proposed compression and decompression units have been extensively redesigned accordingly, ii) these units have been synthesized using Synopsis Design Compiler, which provides accurate timing, energy, and area numbers, iii) the proposed mechanisms are quantitatively compared against state-of-the-art techniques, and iv) the experimental evaluation quantifies the impact on performance and energy consumption.

Experimental results show that, in a conventional register file, there are memory cells that keep the same logic value (either ‘0’ or ‘1’) for the entire execution; that is, a 100% maximum duty cycle distribution, which causes severe V_{th} degradation. In contrast, the proposed techniques reduce the maximum ‘0’ and ‘1’ duty cycle distributions of the conventional register file design by 58% and 68%, respectively. Such duty cycle reductions imply V_{th} degradation savings by 54% and 62%, respectively. Further, compared to a conventional design, powering off registers allows the proposal to reduce the total energy consumption by 20%. These benefits come with less than 0.5% and 5.4% performance degradation and area overhead, respectively.

The rest of the article is organized as follows. Section 2 presents a background about GPU register file organization, the NBTI effect on SRAM cells, and the BDI algorithm. Section 3 motivates the use of data compression strategies. Section 4 defines the compression scheme implemented in our proposal. Section 5 introduces the proposed architectural techniques and state-of-the-art mechanisms. Section 6 discusses experimental results. Section 7 comments on related work, and finally, Section 8 concludes this paper.

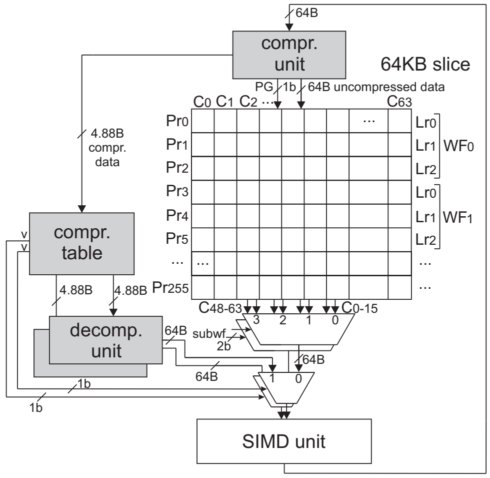


Fig. 1. GPU slice organization and additional compression/decompression units (colored gray). Pr_i, C_i, Lr_i, and WF_i refer to a physical register, register component, logical register, and wavefront, respectively.

2 Background

2.1 GPU Register Files

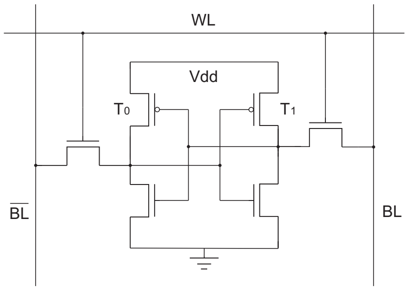
This section summarizes the register file architecture of modern GPUs. Since this paper uses the AMD Graphics Core Next (GCN) family of GPUs as a driving example, AMD terminology is used throughout this work.

Current GCN GPUs include tens of Compute Units (CUs). Each CU has a 256 KB register file and 4 SIMD units; and each SIMD works with a 64 KB slice of the register file, such as the one depicted in Figure 1.

Each slice has 256 registers (labeled as Pr_i in the figure) of 256 bytes. In turn, each register is composed of 64 components of 4 bytes (C_i in the figure). To access to these registers, threads or work-items are organized into groups of up to 64 threads called wavefronts. All threads belonging to the same wavefront access the same register but with a component shift based on the thread id in the wavefront. This way, although each thread works with a different C_i component of the same register, referring to each component is avoided in the ISA. Since a SIMD unit consists of 16 lanes, the threads of a wavefront execute a given instruction in a pipelined fashion, forming 4 bundles of 16 threads called subwavefronts (i.e., C_0-C_{15}, C_{16}-C_{31}, C_{32}-C_{47}, and C_{48}-C_{63}), which access the involved components of a register in successive cycles.

The 256 registers of the slice are distributed among the wavefronts running on the corresponding SIMD unit. To do that, when a wavefront starts executing, it receives the physical address of its base register (reg_{base}) and the number of registers that its execution requires (N), which is a constant value for all the wavefronts of a given kernel. Then, wavefront instructions refer to *logical registers* (Lr_i in the figure) with an index that is added to reg_{base}. This way, a target physical register address is calculated as $reg_{phys} = reg_{base} + index$, where $0 \leq index < N$. We refer to the set of registers that a wavefront can access as the register window of the wavefront. Note that each running wavefront receives a different reg_{base}, so that register windows do not overlap.

Figure 1 also shows an example where wavefronts WF_0 and WF_1 are assigned 2 different 3-register windows ($N = 3$), with base register Pr_0 ($reg_{base} = 0$) and Pr_3 ($reg_{base} = 3$), respectively. For those instructions of wavefront WF_0, the logical register Lr_2 is mapped to the physical register Pr_2 ($reg_{phys} = 0 + 2$), while for WF_1 instructions, Lr_2 is mapped to Pr_5 ($reg_{phys} = 3 + 2$).



Duty cycle

	'0'	'1'	switch-off
T0	NBTI	recovery	recovery
T1	recovery	NBTI	recovery

Fig. 2. Implementation of a 6T SRAM cell and duty cycle effects on T_0 and T_1 PMOS transistors.

2.2 SRAM Cell Aging and Power-Gating Circuit

To help understand how the logic value (‘0’ and ‘1’) distribution affects the lifetime of the cell transistors used to implement GPU register files, this section shows the implementation of a typical SRAM cell and how it suffers from the NBTI phenomenon.

Figure 2 presents a typical SRAM cell consisting of 6 transistors (6T cell). The labeled transistors (T_0 and T_1) refer to the PMOS transistors that partially form the inverter loop to store a logic value, while the remaining 4 transistors are NMOS. Two NMOS transistors are used to implement the inverter loop together with the PMOS transistors, whereas the other pair are pass transistors controlled by the *wordline* (WL) signal to allow read and write operations through the *bitline* (BL) and its complementary ($\overline{\text{BL}}$).

When a cell is under a ‘0’ duty cycle (i.e., when the cell is stable and stores a logic ‘0’), the T_0 transistor is under stress and suffers from NBTI. On the contrary, under a ‘1’ duty cycle, T_1 is the transistor affected by NBTI. The aging effects induced by each type of duty cycle are complementary. This means that, for a given duty cycle, the PMOS transistor that is not under stress can partially recover from the degradation caused by NBTI. Therefore, if every bit cell of the register file experiences a balanced duty cycle distribution (50% for each logic value), even though the cell still ages, this effect is evenly distributed among the cell PMOS transistors and minimized compared to other cells with a more biased duty cycle distribution. However, balanced duty cycle distributions rarely turn out in a conventional design.

Previous works have attacked the NBTI effect by periodically inverting the stored values in CPU register files [3], [41]. However, by storing a given logic value, either T_0 or T_1 transistors are continuously under stress. In addition, frequent inversions might exacerbate the phenomenon referred to as Hot Carrier Injection (HCI), which also contributes, although in a lesser extent than NBTI, to transistor aging [40]. An alternative approach addressing these issues consists in using the power-gating technique [31], which has been widely adopted to save energy in memory structures [18]. In particular, a power-gating design focused on NBTI mitigation should include an NMOS high-V_t sleep transistor connecting the 6T cell to ground, resulting in the cell ground terminals tied together to a virtual ground. When the sleep transistor is on (active state), the cell operates as usual, yet with a ground voltage equal to the virtual ground, which does not affect the SNM since by definition it is a DC noise voltage quantity. On the contrary, when the sleep transistor is off (*switch-off* state), the cell is disconnected from the ground and both T_0 and T_1 transistors remain in a partial recovery mode at the same time, since their gates are simultaneously connected to logic ‘1’ [9]. Notice too that the sleep transistor is NBTI-free since it is NMOS.

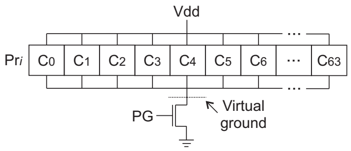


Fig. 3. Implementation of the NBTI-aware power-gating technique.

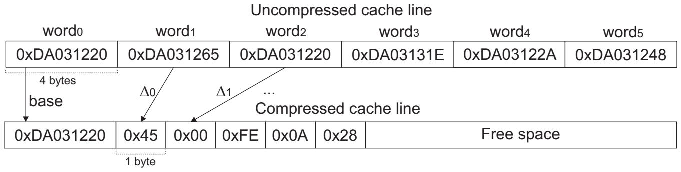


Fig. 4. Compressed CPU cache line using the BDI algorithm.

The power-gating technique can be applied at different levels of granularity [31]. As depicted in Figure 3, the proposed approach incorporates a transistor for each 256-byte register to power off entire registers in the GPU register file. This results in a coarser granularity and eases the implementation compared to previous techniques powering off CPU registers [34]. The sleep transistor is controlled by a *PG* signal (see Figure 1), which is set to ‘0’ by the compression unit when a register is found to be compressible. This way, all the register cells remain in the active or switch-off state when this signal bit is ‘1’ or ‘0’, respectively.

2.3 BDI Compression Algorithm

Our proposed data compression scheme is inspired by the BDI algorithm [29]. BDI was designed to compress CPU cache lines by calculating the arithmetic differences or deltas (Δ) between the first word (referred to as base *b*) and the rest of the cache line words, and storing the base and all the obtained deltas (e.g., 15 deltas for a 64-byte line composed of 4-byte words) in the same line. The compression factor depends on the sizes of the obtained deltas. Figure 4 shows an example of how the different deltas are obtained in a 24-byte cache line. The 4-byte words (excluding $word_0$) are replaced by 1-byte deltas in the compressed cache line and can be reconstructed with Equation 1.

$$word_i = b + \Delta_{i-1} \tag{1}$$

The result is that the system performance can be boosted, since additional compressed data blocks can be stored in the free space of the cache line. From a cell aging perspective, this space could be power gated to mitigate aging in these cells. However, those cells storing the compressed data would still be exposed to NBTI.

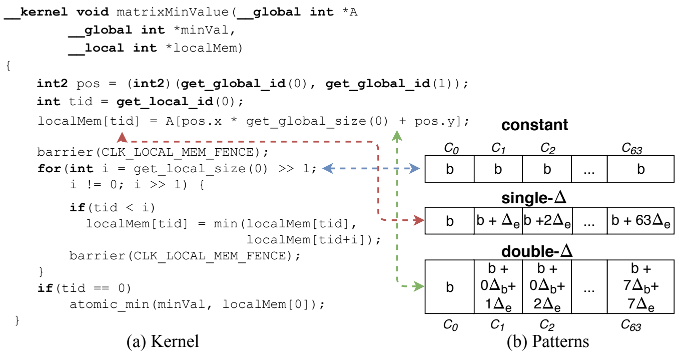
3 GPU Register Pattern Characterization

Best practices for GPU programming recommend the use of regular memory access patterns and the avoidance of branch divergence [28], [16], [5]. Skilled programmers follow these guidelines and write code that, when executed, stores regular data patterns in GPU registers. The proposed aging-aware techniques mainly try to exploit such regular patterns.

Compressing GPU registers requires to find compressible patterns and a compression scheme. This section characterizes a set of candidate patterns, and the next section describes the proposed compression scheme.

3.1 Identification of Compressible Patterns

GPU register data patterns can be classified into 4 categories: *constant*, *single*-Δ, *double*-Δ, and other. The first three exhibit



```
__kernel void matrixMinValue(__global int *A
        __global int *minVal,
        __local int *localMem)
{
    int2 pos = (int2)(get_global_id(0), get_global_id(1));
    int tid = get_local_id(0);
    localMem[tid] = A[pos.x * get_global_size(0) + pos.y];

    barrier(CLK_LOCAL_MEM_FENCE);
    for(int i = get_local_size(0) >> 1;
        i != 0; i >> 1) {

        if(tid < i)
          localMem[tid] = min(localMem[tid],
                              localMem[tid+i]);
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    if(tid == 0)
        atomic_min(minVal, localMem[0]);
 }
```

(a) Kernel

(b) Patterns

Fig. 5. OpenCL kernel to find the minimum value in a matrix with three data types: scalar, vector, and matrix, and their corresponding pattern.

regular compressible patterns and correspond to accesses to scalar, vector, and matrix data types, respectively. Such regularity is increased by locality optimization programming techniques, such as tiling, commonly used in GPU programming.

Below, we discuss the identified three common patterns. Next, we illustrate them through working examples, including locality optimization programming techniques.

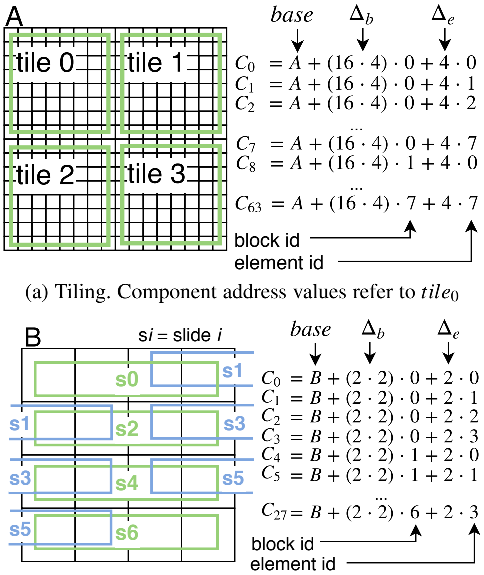
The *constant* pattern appears when all threads from a wavefront are accessing the same scalar value, and hence, all C_i register components are equal to a given base value b. Although the compiler can avoid this pattern by statically figuring out that it is a scalar operand and thus storing the value in the scalar register file, in practice, such patterns are common due to divergence control [42].

The *single*-Δ pattern occurs when a register stores a sequence of values where the difference, element delta (Δ_e), between successive *elements* of the sequence is constant; for example, when storing the addresses of a vector.

Finally, the *double*-Δ pattern mostly corresponds to linearly stored matrices in a continuous memory region. Each matrix element address, and hence, its corresponding component C_i, can be obtained from the coordinates of the element in the matrix and two different deltas, element (Δ_e) and block (Δ_b), which refer to the element size and the row (i.e., second dimension) size of the matrix, respectively. In addition, this pattern appears when using programming techniques such as tiling or sliding window.

To provide a sound understanding of these patterns, let us illustrate them through three examples. The first example is a simple OpenCL 2D kernel that finds the smallest element in a 2D matrix as depicted in Figure 5(a). The highlighted code lines produce the patterns plotted in Figure 5(b).

The *constant* pattern appears in several registers like those storing the initial value of `i`, `get_local_size(0) >> 1`, and the constant `0`. Regarding the *single*-Δ pattern, the register storing the variable `tid` contains the thread unique global work-item id values. At the initial kernel launch, the first thread id is zero or an integer offset, and, by definition, the difference between consecutive thread id values in a wavefront equals to 1; therefore, all the `tid` values can be reconstructed from the tuple: $b = 0$ or *offset* and $\Delta_e = 1$. The *single*-Δ pattern is also present in the register used for addressing the locally stored array `localMem` because all consecutive pairs of components of this register differ by an offset equal to 4; i.e., the size in bytes of an integer value. In general, any register referencing a vector in a continuous memory region can be encoded with a *single*-Δ pattern, being b and Δ_e the initial array address and the element size, respectively. The *double*-Δ pattern appears accessing the matrix A where the address difference between elements depends on the element and row sizes. In this case, b, Δ_e, and Δ_b contain the values A (address of the first element), 4, and matrix row size × 4, respectively.



A
tile 0
tile 1
tile 2
tile 3
base
Δ_b
Δ_e
$C_0 \;= A + (16 \cdot 4) \cdot 0 + 4 \cdot 0$
$C_1 \;= A + (16 \cdot 4) \cdot 0 + 4 \cdot 1$
$C_2 \;= A + (16 \cdot 4) \cdot 0 + 4 \cdot 2$
...
$C_7 \;= A + (16 \cdot 4) \cdot 0 + 4 \cdot 7$
$C_8 \;= A + (16 \cdot 4) \cdot 1 + 4 \cdot 0$
...
$C_{63} = A + (16 \cdot 4) \cdot 7 + 4 \cdot 7$
block id
element id

(a) Tiling. Component address values refer to $tile_0$

B
s*i* = slide *i*
s0
s1
s1
s2
s3
s3
s4
s5
s5
s6
base
Δ_b
Δ_e
$C_0 = B + (2 \cdot 2) \cdot 0 + 2 \cdot 0$
$C_1 = B + (2 \cdot 2) \cdot 0 + 2 \cdot 1$
$C_2 = B + (2 \cdot 2) \cdot 0 + 2 \cdot 2$
$C_3 = B + (2 \cdot 2) \cdot 0 + 2 \cdot 3$
$C_4 = B + (2 \cdot 2) \cdot 1 + 2 \cdot 0$
$C_5 = B + (2 \cdot 2) \cdot 1 + 2 \cdot 1$
...
$C_{27} = B + (2 \cdot 2) \cdot 6 + 2 \cdot 3$
block id
element id

(b) Sliding window. Component address values refer to sequential slide windows

Fig. 6. Examples of *double*-Δ patterns. A and B refer to the memory address of the first matrix elements.

The second and third examples, Figures 6(a) and 6(b), illustrate other occurrences of the *double*-Δ pattern under common programming techniques: tiling and sliding window. For tiling, we assume a 16×16 matrix consisting of 4-byte elements. Instead of accessing the whole matrix at once, programmers usually write code accessing disjoint *tiles* of the matrix to maximize locality. For example, dividing the access to matrix A into four 8×8 tiles, a register storing the addresses of the $tile_0$ elements can be compressed using $b = A$, $\Delta_e = 4$ (element size), and $\Delta_b = 64$ (row size = 16 elements × 4 bytes). In the sliding window example, different sets of accesses intersect among them. The size of a sliding window is 1×4 elements, so there are 7 sliding windows in the 4×4 matrix B containing 2-byte elements with a 2-element stride. In this case, a register stores addresses of elements of sequential slides, being $\Delta_e = 2$ and $\Delta_b = 4$. The latter refers to the shift of 2 times the element size. Unlike tiling, the sliding window technique can also be used for accessing single dimensional arrays.

3.2 Register Pattern Characterization in GPU Kernels

To check the potential of our approach, we characterized the described data patterns in a representative subset of GPU applications from the OpenCL SDK 2.5 benchmark suite [1] that highly stress the register file.

Figure 7 shows the distribution of write operations in the whole register classified according to the three patterns for the studied applications. Label *other* refers to register write operations that do not follow the identified patterns. The reader is referred to Section 6 for further details about the experimental environment.

The previously identified patterns are frequent in GPGPU kernels. Those kernels that more benefit from our approach are *MatrixT*, *QRandS* and *RadixS*, which show a percentage of writes with such patterns surpassing 70% of the total performed writes. Moreover, the percentage in *MatrixT* is over 90%. As opposite, *BlackS* is the only kernel where the identified patterns are less than 25% of writes. This is due to this kernel uses a high amount of

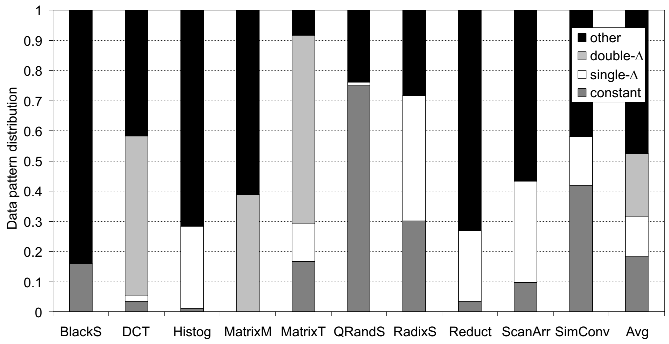


Fig. 7. Data pattern distribution for the GPU register file.

floating-point and random data. Constant patterns in *BlackS* mainly appear due to all threads from a wavefront accessing to the same memory address.

In summary, the characterization study results show a high potential of compressing/decompressing registers dynamically at run-time, since, on average, the identified patterns represent more than half (i.e., 52%) of the total writes.

4 Compression/Decompression Scheme

The previous section has defined the intuitive foundations in which our approach relies. This section formalizes the proposed compression scheme. To simplify the hardware implementation, the three patterns are encoded with Equation 2, where j and k are derived indexes from the i-th work-item id.

$$C_i = b + j \times \Delta_b + k \times \Delta_e \qquad (2)$$

For the *double*-Δ case with matrices, j and k refer to the row and column of the i-th element, and can be computed as $\lfloor i \ / \ \mathrm{BS} \rfloor$ and i % BS, respectively, where BS (block size) refers to the row size, tiling width, or sliding window size. For the *single*-Δ pattern, the difference between all the components is Δ_e, i.e., $\forall i \in [0,63]$: $C_i = b + i \times \Delta_e$, which is obtained when $j = \lfloor i \ / \ \mathrm{BS} \rfloor$, $k = i$ % BS, and $\Delta_b = \mathrm{BS} \times \Delta_e$ in Equation 2. Finally, the *constant* pattern leaves both deltas to 0 and only stores the base; i.e., $\forall i \in [0,63] : C_i = b$.

For illustrative purposes, the right side of Figure 6 shows how Equation 2 is applied to compute the components of the example in two *double*-Δ pattern examples (tiling and sliding window). Labels *block id* and *element id* correspond to j and k, respectively. Whilst *block id* identifies either tile rows or slides, *element id* identifies each element within a tile row or slide.

Note that for 2D kernels, BS is an input parameter specified by the programmer, which is often a power of two to properly fit the kernel into the register file and does not change throughout the kernel execution. Thus, Δ_e and Δ_b can be computed as the offsets between two consecutive components within a block and the offsets between the first component of consecutive blocks, respectively. For *double*-Δ patterns, we have observed that benchmarks operate with a BS set to 8 by default, so this is the value that will be used for the rest of the paper.

From an implementation perspective, this approach presents an important advantage over BDI. While BDI requires to store a base plus $n-1$ deltas, where n is the number of register components, our compression scheme only stores a base and two deltas in a tiny auxiliary table before turning off a complete register for aging mitigation purposes, which makes feasible to dynamically compress/decompress registers with a simple hardware module.

5 Techniques to Mitigate NBTI in Register Files

This section presents the proposed architectural mechanisms for mitigating NBTI in register file cells; namely, Register Compression and Switch Off (RC) and Register Address Rotation (RAR). Besides, in order to make the paper self-contained, we describe two state-of-the-art approaches, namely, Warped-Compression (WC) [22] and ARGO [27], that have been implemented and evaluated for comparison purposes.

5.1 RC: Register Compression and Switch Off

As mentioned above, the proposed approach is inspired by the BDI algorithm [29] for CPU caches. Given that most instructions have one destination register and two source registers, RC includes one compression unit and two decompression units for each slice in the register file (see Figure 1).

5.1.1 Compression Unit

The compression unit sits at the output of the SIMD ALU and checks whether the register components are compressible. Given our delta-based compression scheme, it has to compute Δ_e and Δ_b values. As explained in Section 2.1, the register file and SIMD ALU widths do not match in an AMD GCN GPU. A register consists of 64 components, whereas the ALU operates each cycle with a subwavefront of 16 components from the register. Handling this mismatch requires to have two modules in the compression unit, that is, a computing deltas module and a control module managing the state. On the contrary, for NVIDIA architectures, a control module would not be necessary since the slice access is not pipelined.

The computing deltas module has to be fast enough to fit in the cycle time of the GPU, but small and energy-efficient enough to pay off for the extra area and energy consumption. A simple yet efficient approach is plotted in Figure 8. The figure also includes the control module, which enables different signals for the computing deltas module depending on the current state.

The control module tracks whether either a new pair of delta candidates from the first subwavefront (components from C_0 to C_{15}), Δ_e and Δ_b, are stored when an ALU operation starts, or the stored pair of deltas are compared with deltas from successive subwavefronts (either C_{16} to C_{31}, C_{32} to C_{47}, or C_{48} to C_{63}) to drive the computing deltas module operation. The control module is implemented as a finite state machine consisting of 5 states as depicted in Figure 9. Each state in turn enables the signals from the computing deltas module presented in Table 1. When the ALU ends computing each subwavefront, the control module checks the *compr* signal from the computing deltas module, and either transits to the next delta state or returns to the *idle* state, deactivating the delta computation until the ALU operates with a new destination register.

The computing deltas module consists of a single level of subtractors that compute, for the two blocks of a given subwavefront, the offset between consecutive components, $\Delta_{e_j} = C_{i+1} - C_i$, and the offset between the first components of consecutive blocks, $\Delta_{b_k} = C_{i\times 8} - C_{i\times 8-8}$. The subsequent level of comparators determines if all Δ_{e_j} and all Δ_{b_k} from the current subwavefront are equal to Δ_e and Δ_b, respectively, from the first subwavefront. For this purpose, Δ_e and Δ_b are stored in two intermediate buffers controlled by the W_{en_Δ} signal, which is only activated in the *deltas* C_0-C_{15} state. Similarly, since computing Δ_{b_k} requires $C_{i\times 8-8}$

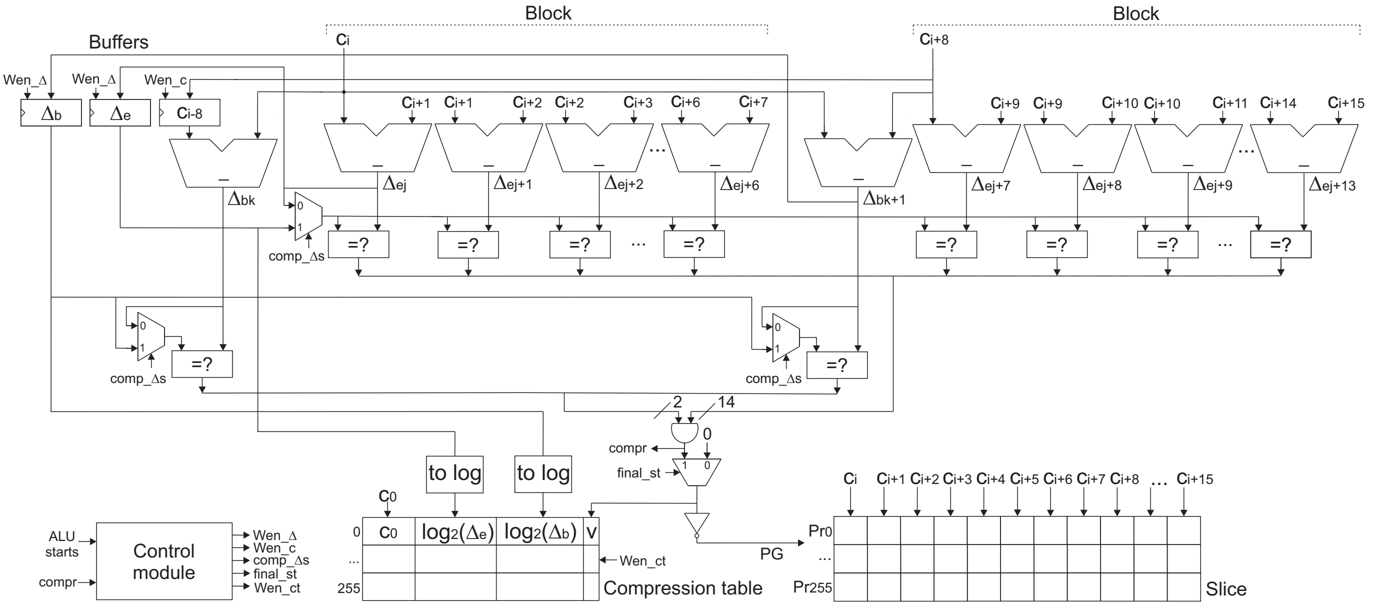


Fig. 8. Computing deltas module from a compression unit. A control module is included in the bottom left.

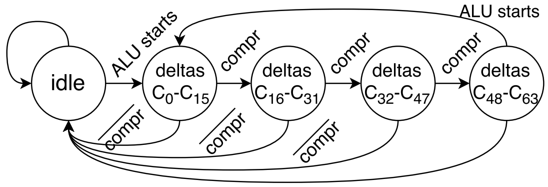


Fig. 9. Finite state machine from the control module of a compression unit. The *compr* signal is active when components can be compressed.

TABLE 1
Signal values for each state of the control module.

State name	Signals				
	W_{en_Δ}	W_{en_C}	$comp\_\Delta s$	$final\_st$	W_{en_ct}
idle	0	0	0	0	0
deltas C_0-C_{15}	1	1	0	0	0
deltas C_{16}-C_{31}	0	1	1	0	0
deltas C_{32}-C_{47}	0	1	1	0	0
deltas C_{48}-C_{63}	0	0	1	1	1

from the previous subwavefront, this component is stored in the buffer controlled by W_{en_c}. This signal is only activated in those states referring to a possibly compressible register (i.e., *deltas* C_0-C_{15}, *deltas* C_{16}-C_{31}, and *deltas* C_{32}-C_{47}). The intermediate buffers are implemented with 1T1C eDRAM cells, which are NBTI-free by design as they do not include PMOS transistors [14].

Depending on the current state, the first input of a comparator comes either from the above subtractor or from the buffers storing deltas. In particular, the subtractor outputs are only used in the *deltas* C_0-C_{15} state, since there is not a valid pair of deltas from a previous state. Otherwise, the buffer outputs are used. This is accomplished by enabling the $comp\_\Delta s$ signal. If both deltas are zero or positive and a power of two, and all comparators output '1', the register components of the current subwavefront are found as compressible by enabling the *compr* signal. Meanwhile, regardless of this signal is enabled or not, the 16 register components are stored in the slice.

In the *deltas* C_{48}-C_{63} state, $final\_st$ is enabled, meaning that *compr* is forwarded to the multiplexer output instead of '0'. If the *compr* signal is active, the entire destination register is found to be compressible. In such a case, the register is powered off using the power-gating technique ($PG =$'0', see Section 2.2), whereas C_0, Δ_e, Δ_b, and a valid v bit are stored in an auxiliary compression table by enabling W_{en_ct}. Likewise the intermediate buffers, the compression table is implemented with NBTI-free eDRAM cells.

The compression table keeps as many entries as registers in a slice; i.e., 256. For each entry, the v bit indicates whether the entry contains compressed data. Note that this table does not store Δ_e and Δ_b, but $log_2(\Delta_e)$ and $log_2(\Delta_b)$, speeding up decompression and reducing the table size. For example, on a 2D sliding window implementation, a kernel could generate a *double*-Δ pattern like $2-4-6-8-10-12-14-16-10-12-14-16-18-20...$, in which the destination register would be switched off and the associated table entry would store $C_0 = 2$, $log_2(\Delta_e) = 1$ (i.e., $\Delta_e = 2$), and $log_2(\Delta_b) = 3$ (i.e., $\Delta_b = 8$).

A compression table entry comprises the v bit, 4 bytes to store C_0, 3 bits to store $log_2(\Delta_e)$, and another 3 bits to store $log_2(\Delta_b)$. Experimental results show that 3 bits for each logarithm are enough since considering higher values than 64 (2^6) for Δ_e and Δ_b provides minor compression coverage improvement. This implies that those patterns with either Δ_e or Δ_b exceeding such a limit are not compressed. We use the binary value "111" to code the special case in which Δ_e or Δ_b are zero, since the binary logarithm is not defined for this value.

Overall, as a register consists of 64 components operated in 4 subwavefronts of 16 components, a compression unit requires 16 subtractors and 16 comparators ($14+2$ to compare Δ_e and Δ_b values, respectively), whereas the compression table size is 1.22 KB. Two crucial observations enable to simplify the compression circuitry: i) all valid deltas are one-hot values, that is, single '1' bit and the rest '0' since valid deltas are power of two values, and ii) the '1' bit can only be present in bit positions from 0 to 6 because the largest delta is 64. Therefore, the output bits 7 to 31 from the subtractors have to be always '0' to compress a register.

Finally, write operations from divergent instructions must act on uncompressed registers, since they might update only a subset of their components, leaving the remaining components untouched. To do so, similarly to as done in [22], we dynamically issue a special MOV instruction ahead of a divergent write only if the destination register was compressed. The aim of this special

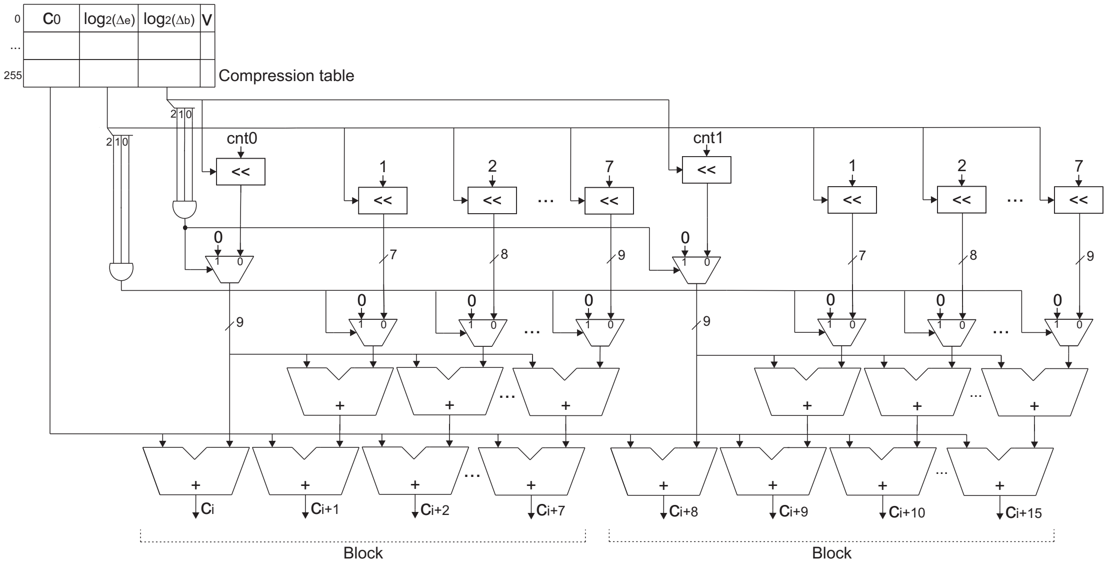


Fig. 10. Decompression unit.

instruction is to decompress the register contents and store them in the destination register before the divergent write is performed. Experimental results show that the overhead caused by these additional instructions is minimal, since, on average, less than 0.7% of total register writes imply such MOV insertions.

5.1.2 Decompression Unit

The decompression circuit unwinds 16 components on each subwavefront read access. A possible implementation is shown in Figure 10. The inputs to this unit are C_0, $log_2(\Delta_e)$, and $log_2(\Delta_b)$ from the compression table. For subwavefronts from 0 to 3, cnt_0 and cnt_1 constants are updated to 0 and 1, 2 and 3, 4 and 5, and 6 and 7, respectively, and refer to the *block id* of the source register.

Given that Δ_b is zero or positive and a power of two, calculating the offset between C_0 and the first component of the involved block is equivalent to binary shift cnt_i to the left Δ_b times; i.e., $cnt_i << log_2(\Delta_b)$, which largely simplifies the required logic. When $log_2(\Delta_b)$ is encoded as the binary value "111" ($\Delta_b = 0$), the corresponding AND gate controlling the below multiplexer is active and a zero is forwarded instead of $cnt_i << log_2(\Delta_b)$. Within a block, seven shift operations are also performed to calculate the offsets between the first component of the block and C_i. In this case, constants from 1 to 7 refer to the *element id*. Likewise the previous shift, an AND gate and multiplexers are used to either forward zero or non-zero offsets.

The first level of adders calculate the offset between C_0 and C_i, whereas the second level of adders sum C_0 to the first-level adder output to finally restore each component. For instance, if $C_0 = 2$, $log_2(\Delta_b) = 3$, and $log_2(\Delta_e) = 1$, the decompression circuit computes $C_1 = 2+0 << 3+1 << 1$, $C_2 = 2+0 << 3+2 << 1$... $C_8 = 2+1 << 3+0 << 1$, $C_9 = 2+1 << 3+1 << 1$..., which results in the data pattern $2-4-6-8-10-12-14-16-10-12-14-16-18-20$... Overall, a decompression unit is composed of 16 shifters, 16 multiplexers, 14 first-level adders, and 16 second-level adders.

5.1.3 Timing, Energy, Power, and Area Estimations

This section quantifies the timing, energy, power, and area estimations of the memory structures and additional logic used to implement the proposed RC mechanism. The memory structures have been modeled with CACTI-P [23], whereas the combinational logic has been synthesized with Synopsis Design Compiler and simulated with Mentor Graphics Modelsim. The technology library corresponds to a low-power 32nm technology available to European universities, featuring eight metal layers. For timing closure, we set a target of a 1 GHz clock frequency like that of the AMD GCN HD 7770 GPU considered in the experiments. Table 2 shows the results for the main design components.

TABLE 2
Timing, energy, power, and area values for a 32nm technology node and a 1GHz clock frequency. N/A: Not applicable.

	Slice	Compr. table	Compr. unit	Decompr. unit
Access time (ns)	0.92	0.29	0.70	0.70
Read energy (pJ)	295.86	1.25	N/A	0.96
Write energy (pJ)	365.91	66.49	1.10	N/A
Leak. power (mW)	75.86	0.13	8.46	8.00
Ret. time (cycles)	N/A	119K	N/A	N/A
Area (mm^2)	0.889	0.020	0.007	0.007

On a conventional GCN GPU, the execution of a subwavefront from an instruction takes 4 cycles. The first and fourth cycles are devoted to register read and write operations, whereas the remaining cycles are for SIMD operation [4]. Once the first subwavefront has written to the destination register, the remaining 3 subwavefronts write to the same register in the successive 3 cycles.

Adding a compression unit does not imply any additional cycle on a slice write operation, since both the slice and the compression unit are accessed in parallel. Recall that, even though the destination register can be found as possibly compressible in intermediate states, the involved components are written to the slice on each state. On the other hand, updating the tiny compression table in the *deltas* C_{48}-C_{63} state when the destination register is compressible fits within the cycle time ($0.29\,ns + 0.70\,ns < 1\,ns$).

Regardless of whether the source registers are compressed or not, adding two decompression units does not impact on the slice read access time. This is accomplished by accessing in parallel the source registers and the compression table. Besides, the sum of the compression table and decompression unit delays does not surpass the cycle time. However, we conservatively assume the wakeup delay of a register to be 10 cycles [22], [27], which may impact on performance since an instruction writing a non-compressible pattern must switch on a powered-off destination register before writing to it (see Section 6.5). On the other hand, switching off a register does not impact on performance since this action does not prevent subsequent instructions from accessing the slice.

Accessing in parallel the slice and the proposed compression and decompression units might increase the dynamic energy consumption with respect to a conventional design. Fortunately, the pipelined slice access helps reduce the overall dynamic consumption, since as soon as a register is found as not compressible or not decompressible in any state, these units are deactivated for successive subwavefronts accessing the register. Similarly, on a read access, if a register is found as compressed in the *deltas* C_0-C_{15} state, successive subwavefronts accessing the register are canceled. Besides, the dynamic energy contributions of the decompression and compression units on read and write operations, respectively, are minimal compared to those of the slice. Notice too that the dynamic energy expenses of the compression table are higher than those of the compression and decompression units, especially for write operations, but much lower than those of the slice.

Regarding leakage power, one compression and two decompression units represent about 32% of the slice leakage power consumption. On the other hand, the leakage contribution of the compression table is almost zero since it is implemented with eDRAM cells. The reader is referred to Section 6.6 for a deeper energy analysis.

The eDRAM cells require from periodic refreshes to maintain their data. This might negatively impact on energy consumption and performance, since refreshes compete with regular accesses to the compression table. To minimize the impact on energy and performance, the refresh mechanism is implemented as distributed [25] and eDRAM cells are built with typical 20 fF capacitors, which translates into a retention time of 119048 cycles [23]. This implies a large refresh period of 465 cycles for the 256-entry compression table. On the other hand, the buffers from the compression unit do not require from additional refresh cycles since their contents are just retrieved in the successive cycles of a slice write operation. Notice too that the area overhead of the compression table and additional logic is by 3.82% of the much larger slice.

Finally, CACTI-P has been also used to model the power-gating circuit required to shut off registers, including all the sleep transistors and interconnections. This technique imposes a register wakeup energy by 232.88 pJ and an area overhead by 0.013 mm^2 (1.53%) over the slice, which is akin to other power-gating designs from the industry reporting area overheads [30].

5.2 RAR: Register Address Rotation

As explained in Section 2.1, during its execution, a wavefront is assigned an N-register window. When the wavefront ends its execution, it releases its register window, which is ready to be assigned to a new incoming wavefront. Taking into account that wavefronts from the same kernel usually show a similar behavior, it is very likely that RC will end up powering off the same registers when a window is reassigned. This behavior impacts on the register file lifetime as some registers will not be switched off at all (see Section 6.2). To address this problem, this section presents a simple yet effective register address rotation mechanism, RAR, aimed at distributing switch-off cycles among registers.

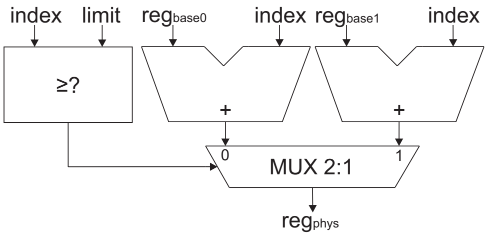


Fig. 11. RAR circuit.

TABLE 3
Timing, energy, power, and area values for the WC compression and decompression units. N/A: Not applicable.

	Compression unit	Decompression unit
Access time (ns)	0.70	0.65
Read energy (pJ)	N/A	0.79
Write energy (pJ)	0.76	N/A
Leakage power (mW)	7.01	8.03
Area (mm^2)	0.006	0.008

RAR modifies the address of the base register reg_{base} at window assignment. This way, the N registers are evenly powered off. When a window is reassigned, RAR defines a new base register as $reg_{base}+s$, where s is a value falling in between 0 and $N-1$ that is incremented by 1 `modulo` N on each window re-assignation. Note that in order to avoid accesses to registers from other windows, the address of the target physical register must be computed as $reg_{phys} = reg_{base} + (s + index)\,\%\,N$.

For instance, assuming $N = 10$, $s = 0$, and the execution of a wavefront with $reg_{base} = 50$; if the data stored into the logical register Lr_9 are compressed, then, the physical register Pr_{59} $(50 + (0+9)\,\%\,10 = 59)$ will be powered off. Then, when the window is reassigned s is incremented $(s = 1)$, thus Lr_9 will map to Pr_{50} $(50 + (1+9)\,\%\,10 = 50)$, which will be powered off instead.

Figure 11 plots a possible implementation of the logic to calculate reg_{phys}. To avoid the use of adders supporting any possible `modulo` N on a window reassignment, two different base register addresses are calculated, namely $reg_{base0} = reg_{base} + s$ and $reg_{base1} = reg_{base} + s - N$. Note that reg_{base1} is used instead of reg_{base0} on overflow situations, that is, when $s + index \geqslant N$. The comparator in the circuit is used to determine which base register address will be used by checking if $index \geqslant limit$, where $limit = N - s$. Notice too that the hardware overhead of RAR is minimal as it just includes an additional comparator, a 2-input multiplexer, and an adder with respect to the original logic required to calculate the target physical register. Furthermore, the impact on the critical path is minimal as the 2-input multiplexer delay is just 13 ps as synthesized with Synopsis.

5.3 State-of-the-Art: WC and ARGO

The recently proposed WC and ARGO mechanisms use the power-gating technique on GPU register files for energy and aging savings, respectively. For the sake of fairness, all the studied approaches apply this technique at the same granularity of register.

WC uses the BDI algorithm to compress GPU registers storing data patterns; thus, WC calculates a base value and as many deltas as components within a register minus one, that is, 63 deltas in our modeled GPU architecture. The base value is always stored in the first component of a compressible register, the subsequent cells store the 63 deltas, and the remaining cells are switched off. The percentage of powered off cells within a compressible register depends on the size of the obtained deltas. If they are all zero, just the base is stored. In case that each delta can be represented with one byte, the subsequent 63 bytes after the base store the

TABLE 4
GPU configuration and memory hierarchy.

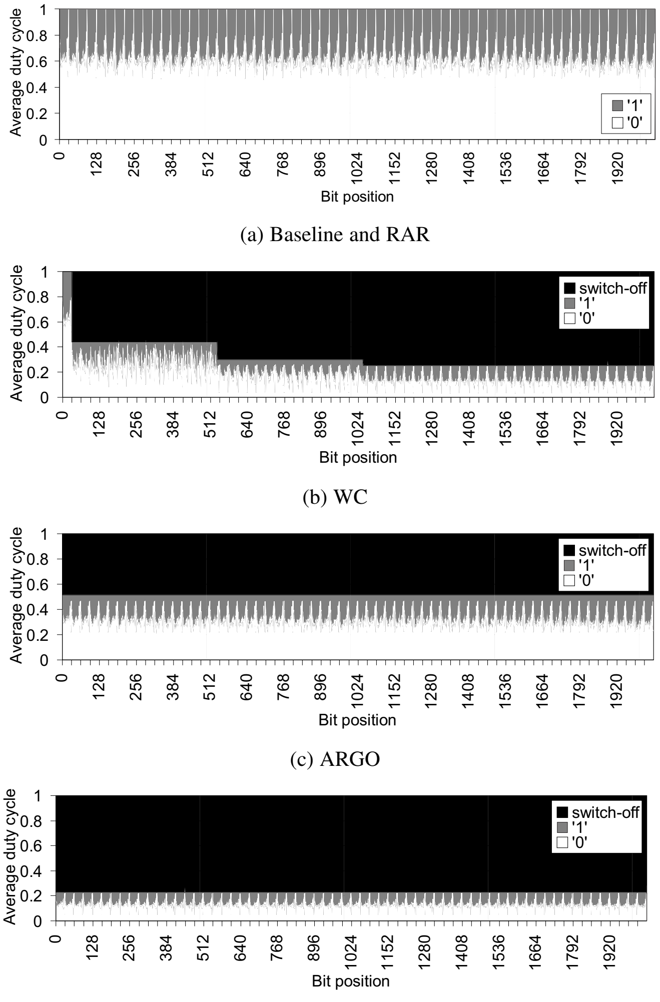
AMD GCN HD 7770 GPU	
Clock frequency	1 GHz
Compute Units (CUs)	10
Vector memory unit	32 entries, 1 per CU
LDS unit	64 KB, 1 per CU, 1 cycle
Scalar unit	2 KB, 1 per CU 1 cycle per instruction
Slice	64 KB, 4 per CU 4-1-1-1 cycles per instruction
Max. wavefronts per slice	16
Work-items per wavefront	64
All caches	LRU, 64 B-line
Scalar L1 caches	16 KB, 4-way, 1 per CU 1 cycle
Texture L1 caches	16 KB, 4-way, 1 per CU 1 cycle
L2 caches	128 KB, 16-way per module 2 modules, 10 cycles
Main Memory	2 channels per L2 module 100 cycles

deltas. If any delta cannot be represented with one byte but two bytes are enough to represent each of them, then 126 bytes after the base store the deltas. Otherwise, the entire register remains on and uncompressed. Table 3 shows the timing, energy, power, and area requirements of the WC compression and decompression units, which have been also synthesized with Synopsis Design Compiler. Compared to those of the proposed RC approach (see Table 2), they mostly present slightly lower numbers since their implementation is simpler. However, the decompression unit shows marginally higher leakage and area since the wiring is increased by 42%. Please, refer to [22] for further implementation details.

The ARGO approach is based on the observation that GPU kernels do not usually occupy all the available registers in a slice. The slice utilization mainly depends on its number of physical registers, the maximum number of concurrent wavefronts allocated to a slice, and the register window size [6]. The two former depend on the GPU architecture, whereas the latter is determined at kernel compilation time. For instance, on a 256-register slice with a maximum of 16 concurrent wavefronts and a kernel with 4-register windows, the slice utilization is 25%. ARGO takes advantage of this fact by powering off entire unused register windows and by modifying the wavefront register allocation so that wavefronts eventually occupy all the register windows on a round-robin basis. The result is to evenly distribute unused windows along a register slice. The reader is referred to [27] for further details.

6 Experimental Evaluation

This section quantifies the duty cycle savings, V_{th} degradation, impact on performance, and energy consumption of the studied architectural mechanisms. To do so, the Multi2Sim simulation framework [38] has been extended to implement the proposed RC and RAR mechanisms, including the additional MOV instructions (see Section 5.1.1). The WC and ARGO approaches have been also modeled for comparison purposes. The V_{th} degradation has been quantified by using a standard formula [37], whereas energy results were calculated by combining processor statistics from Multi2Sim with energy numbers from CACTI-P and Synopsis. Table 4 summarizes the GPU configuration and memory hierarchy, which closely resembles the AMD GCN HD 7770 GPU.



(a) Baseline and RAR

(b) WC

(c) ARGO

(d) RC and RC+RAR

Fig. 12. Average duty cycle distribution on each bit position of the registers of all the studied applications.

6.1 Average Duty Cycle Distribution

The '0' and '1' duty cycles stress T_0 and T_1 SRAM cell transistors, respectively, used to implement the register file (see Figure 2). This section provides insights on how the different duty cycles are distributed within the registers of a slice.

Figure 12 depicts, for each register bit position, the average duty cycle distribution across all the registers for all the analyzed approaches. Results are averaged for all the studied kernels. The '0' and '1' categories represent the amount of time that a given bit position stores logic '0' and '1', respectively, whereas *switch-off* refers to the amount of time that such a bit remains powered off.

The baseline approach refers to a conventional GPU register file design. Notice that this approach and RAR show the same duty cycle distribution as they do not switch off specific cells, and rotating register addresses in RAR does not change the plotted average distribution with respect to the baseline. Similarly to what occurs in CPU applications [15], [39], GPU memory structures usually store zero and narrow integer values, which implies that those cells storing the most significant bits within a register component hold logic '0' most of the time. This is the case for the baseline and RAR, where all components (32-bit each) present longer '0' duty cycles (e.g., 70-80%) as we approach to the most significant bit. In fact, for the most significant bit, which refers to the sign bit of the stored component data, the average duty

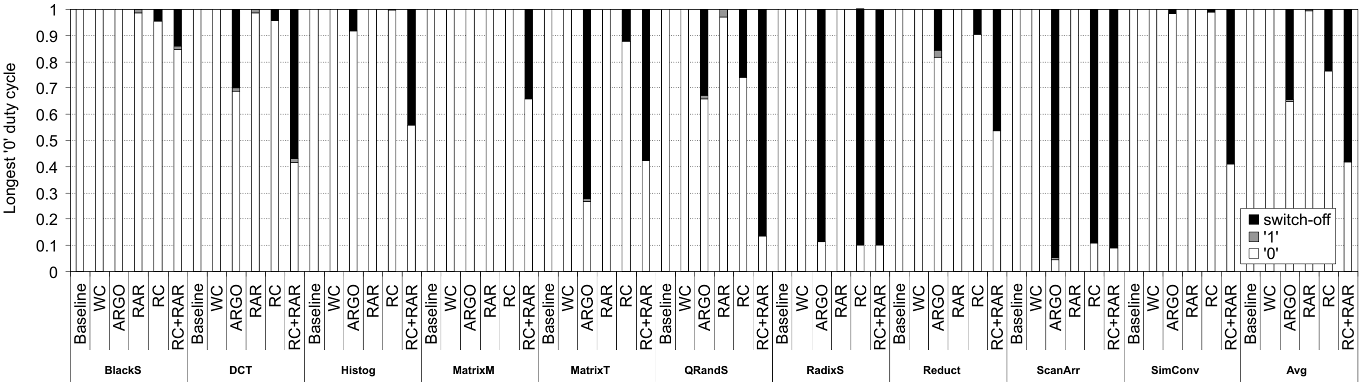


Fig. 13. Longest '0' duty cycle among the cells of the entire slice for the studied mechanisms.

cycle almost reaches 100%, confirming that most data are positive or zero. Results also highlight that logic '0' is more often stored than its counterpart logic '1', thus a usually larger degradation in transistors T_0 is expected.

The remaining mechanisms include the switch-off category as they power off memory cells. WC shows an uneven distribution of switch-off cycles across the register bits. Four different segments can be distinguished. Bit positions from 0 to 31 (first component) always remain on as they store a base value or uncompressed data. Bits from 32 to 535 experience switch-off cycles when a *constant* pattern is detected (i.e., just the base value is stored in the first component while the remaining register bits are powered off) or they are storing either deltas (63 1-byte or 63/2 2-byte deltas) or uncompressed data. Bits from 536 to 1039 can be also powered off when each of the 63 deltas occupies 1 byte. Otherwise, they store either 2-byte deltas or uncompressed data. Finally, bits from 1040 to 2047 are powered off or they just store uncompressed data.

Unlike WC, the ARGO approach completely power gates those register entries from register windows that are not being used by the kernel, and periodically rotates register addresses to spread switch-off cycles across all registers. The result is all bit positions experiencing the same amount of switch-off cycles.

Finally, likewise the baseline and RAR techniques, RC and RC+RAR show the same duty cycle distribution. The proposed approaches also show a uniform switch-off cycle distribution across the register bits by completely power gating entire register entries once a data pattern is detected. Nevertheless, the switch-off duty cycle is significantly larger than that of ARGO, meaning that exploiting data redundancy is more effective than switching off unused registers.

6.2 Longest '0' Duty Cycle Analysis

To quantify transistor aging savings brought by the proposed mechanisms in a memory structure, it is important to analyze the longest '0' and '1' duty cycle distributions, as they will cause the strongest transistor degradation within the memory. This section quantifies the longest '0' duty cycle among the cells of a whole slice across the analyzed benchmarks.

Figure 13 depicts the results for the studied approaches. For each bar, the '0' category represents the maximum percentage of time a cell is keeping a logic '0' value. The remaining categories represent, for the cell with this maximum percentage, the time it spent with a logic '1' value or switched off.

As expected, the longest '0' duty cycle for the baseline reaches 100% for all the kernels, meaning that at least one cell within the slice contains a '0' during all the kernel execution. The same results can be seen for the WC approach. This is due to, as shown in Figure 12(b), WC stores the base value in the first component of a register. Thus, these cells cannot be powered off and remain exposed to long '0' duty cycles.

TABLE 5
Slice utilization and number of unused register windows for each kernel.

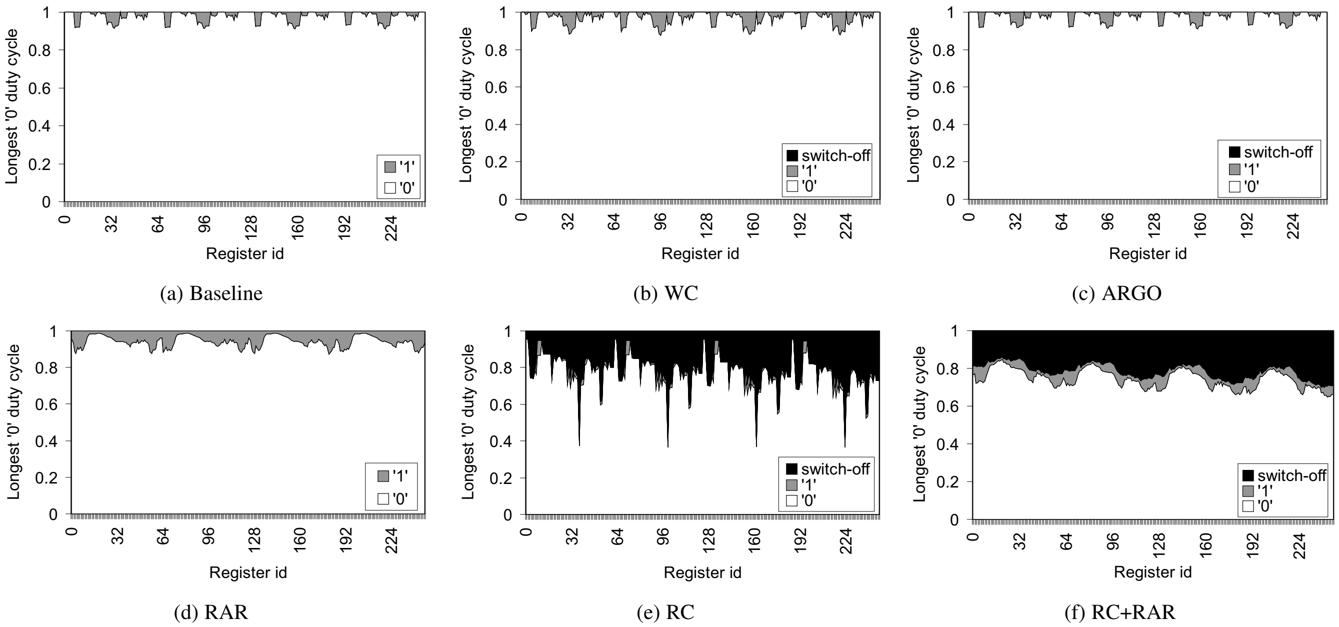
Kernel	Slice utilization	# of unused reg. windows
BlackS	97%	0
DCT	69%	7
Histog	81%	1
MatrixM	80%	0
MatrixT	25%	48
QRandS	58%	2
RadixS	60%	5
Reduct	81%	3
ScanLArr	19%	69
SimConv	94%	1

In contrast, ARGO reduces the longest '0' duty cycle by evenly distributing and switching off unused register windows along the slice. However, there are some applications like *BlackS*, *MatrixM*, and *SimConv* where ARGO is ineffective, leading to 100% '0' duty cycles. This is because, as shown in Table 5, those kernels with a high slice utilization have a very limited number of available unused register windows (if any), thus powering off opportunities are scarce or nonexistent.

Note that just rotating register addresses using the RAR-only approach does not reduce the longest '0' duty cycle. This is because RAR alone does not switch off registers, thus it cannot place both T_0 and T_1 transistors in partial recovery mode at the same time.

The RC-only scheme mitigates the '0' duty cycle by completely powering off the compressed register entries. However, RC is limited by those registers that do not store data patterns, and consequently, cannot be switched off. This leads to memory cells with long '0' duty cycles in kernels like *Histog*. In contrast, enhancing RC with RAR overcomes this problem, since rotating the register addresses distributes switch-off cycles among cells from different registers. This is especially effective in some kernels such as *QRandS* and *SimConv*. Overall, RC+RAR reduces the longest '0' duty cycle on average by 58%, followed by ARGO and RC with 34% and 24%, respectively.

To provide a deeper understanding on how the different techniques impact on the length of '0' duty cycles, figures 14 and 15 plot the longest '0' duty cycle per register in *BlackS* and *DCT*, which show low and medium duty cycle reductions, respectively. For both kernels, the baseline and WC approaches experience a highly biased '0' duty cycle in every register, which



(a) Baseline

(b) WC

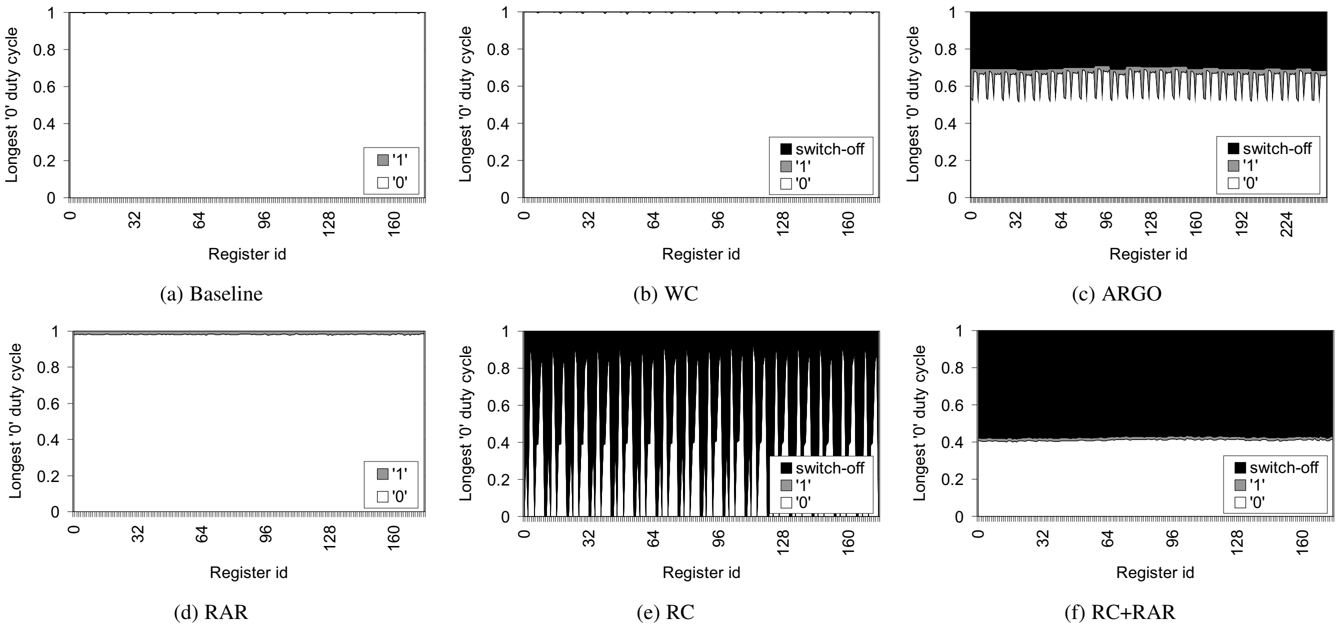
(c) ARGO

(d) RAR

(e) RC

(f) RC+RAR

Fig. 14. Longest '0' duty cycle distribution per register in *BlackS*.



(a) Baseline

(b) WC

(c) ARGO

(d) RAR

(e) RC

(f) RC+RAR

Fig. 15. Longest '0' duty cycle distribution per register in *DCT*.

confirms the presence of multiple memory cells storing a logic '0' for long periods of time. ARGO does not introduce switch-off cycles in *BlackS* due to its high slice utilization; therefore, the presented results are identical to those of the baseline. In contrast, switch-off cycles do appear in *DCT*, where according to Table 5, there are up to 7 unused register windows comprising registers from 176 to 255. Note that, in the x-axis of *DCT*, the number of used registers is reduced to 176 across all the remaining techniques. For them, including the baseline scheme, we assume that those registers not used by a given kernel are switched off during the entire kernel execution. The 100% switch-off duty cycles of such registers are not shown in the figures.

As expected, RAR-only marginally reduces the longest '0' duty cycle, since memory cells storing the most significant bits of a component keep holding logic '0' in spite of rotating register addresses. On the other hand, RC-only shows an uneven distribution of switch-off cycles among registers in both applications. For instance, RC compresses registers 37, 99, 161, and 223 for a long period of time in *BlackS*, but fails to do so in some other registers like 0-1, 62-63, and so on, leading to '0' duty cycle peaks near 100%. A similar reasoning can be made for *DCT*. In comparison, RC+RAR *steals* switch-off cycles from those registers that most benefit from compression and allows powering off more often those registers that, otherwise, would remain on for most of the time, resulting in a more uniform longest duty cycle across all registers. The duty cycle reduction is much more appreciable in *DCT* since it offers higher compression opportunities than *BlackS* (see Figure 7).

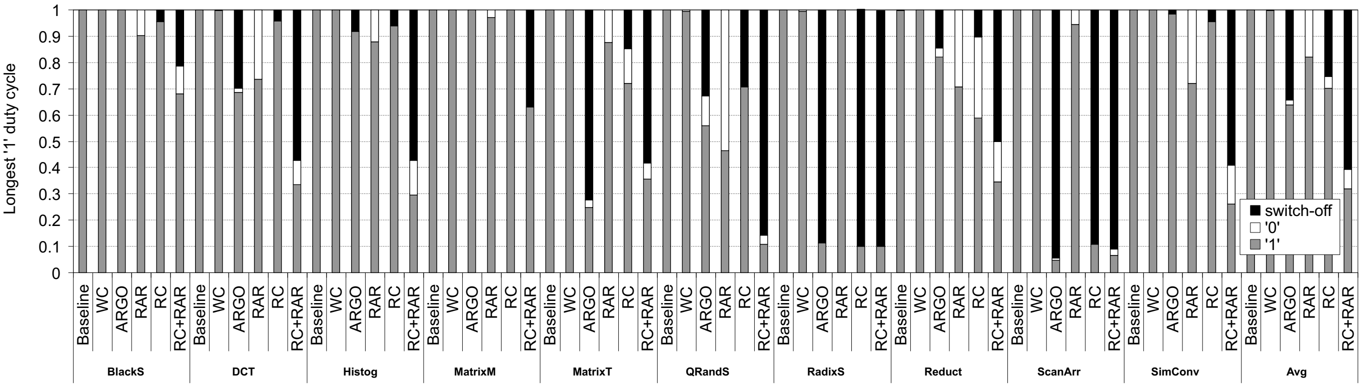
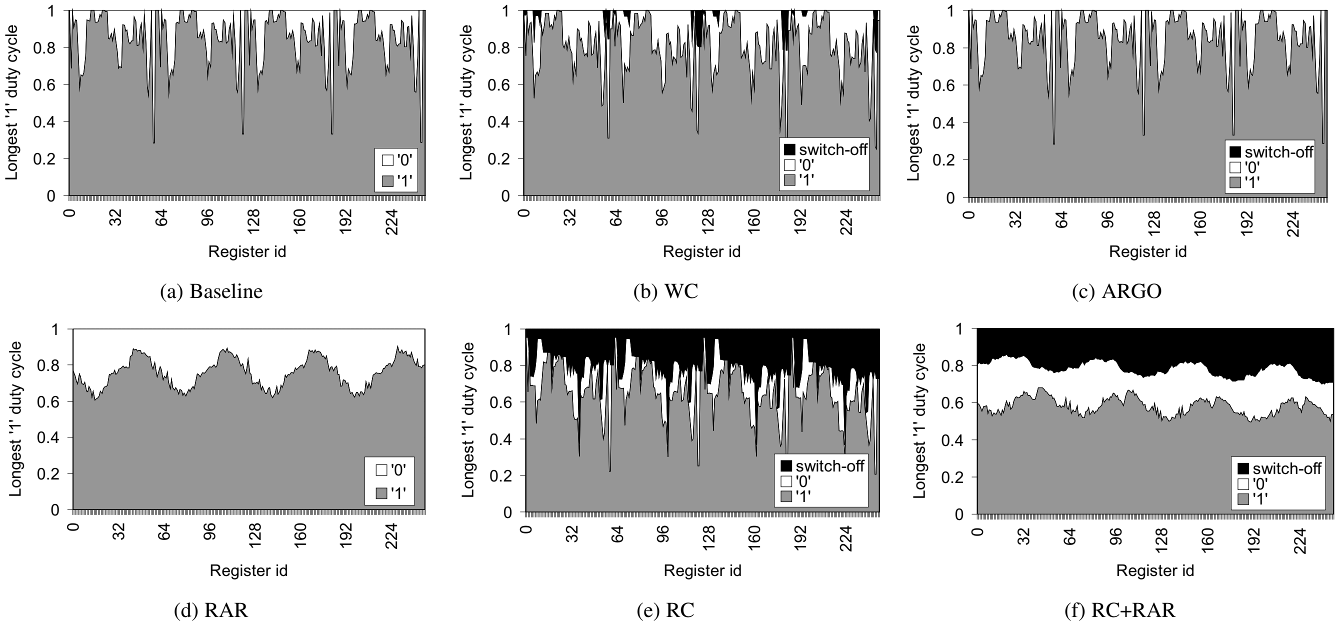


Fig. 16. Longest '1' duty cycle among the cells of the entire slice for the studied mechanisms.



(a) Baseline

(b) WC

(c) ARGO

(d) RAR

(e) RC

(f) RC+RAR

Fig. 17. Longest '1' duty cycle distribution per register in *BlackS*.

6.3 Longest '1' Duty Cycle Analysis

This section evaluates the contribution of logic '1' to the transistor aging. Figure 16 plots the longest '1' duty cycle for the entire slice across the studied benchmarks. In spite of logic '0' being dominant over logic '1' in memory structures, there exist memory cells in which the '1' duty cycle is also highly biased for the baseline and WC schemes in all the studied applications. ARGO also presents such '1' duty cycles in those kernels without unused register windows.

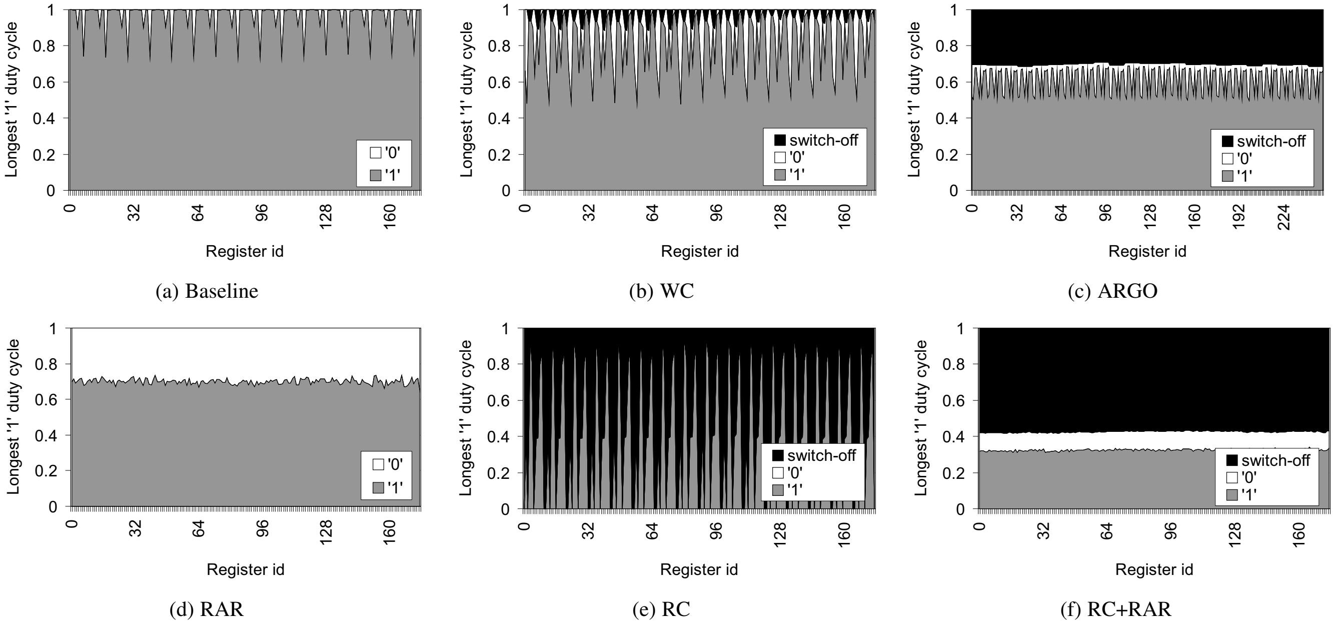
The benefits of RAR-only and RC-only are also modest in applications like *BlackS* or *MatrixM*. However, it is interesting to point out that, unlike the results presented in Figure 13, which show that applying these techniques scarcely introduce '1' duty cycles in those cells with the longest '0' duty cycles, Figure 16 shows that both RAR and RC generate substantial '0' duty cycles in some applications like *MatrixT* and *Reduct*. This is because applying the proposed mechanisms causes another cell to become the one with the longest '1' duty cycle and this cell presents '0' duty cycles. These results also highlight that '0' values are stored more often than '1' values.

RC+RAR largely reduces the longest '1' duty cycles across all applications. These reductions are even larger than in the previous analysis thanks to the introduction of '0' duty cycles. Overall, RC+RAR mitigates the '1' duty cycle on average by 68%. This percentage is by 36% and 30% for ARGO and RC, respectively.

Figures 17 and 18 depict the longest '1' duty cycle per register in *BlackS* and *DCT*. Results confirm that '1' duty cycles are not as critical as '0' duty cycles for NBTI degradation since the presence of logic '0' values is remarkable even for the baseline and WC approaches, which show some dips down to 30% in *BlackS*. Such dips are evenly distributed across the registers when using the RAR technique, and combined with the switch-off state brought by RC lead to significant '1' duty cycle reductions. Finally, notice too that, in this analysis, WC introduces switch-off cycles in some registers. This is due to, at least a memory cell that can be powered off in these registers has a longer '1' duty cycle than those cells always maintaining their state.

6.4 V_{th} Degradation Analysis

This section quantifies the V_{th} degradation (dV_{th}) caused by the analyzed duty cycles. The presented dV_{th} refers to the T_0 and T_1 SRAM cell transistors with the highest dV_{th} within the entire slice, that is, those transistors suffering the highest dV_{th}. This degradation was calculated assuming a 3-year lifetime [26]. Such an execution period is obtained by repeating the kernel execution over and over until the established lifetime is reached [37].



(a) Baseline

(b) WC

(c) ARGO

(d) RAR

(e) RC

(f) RC+RAR

Fig. 18. Longest ‘1’ duty cycle distribution per register in *DCT*.

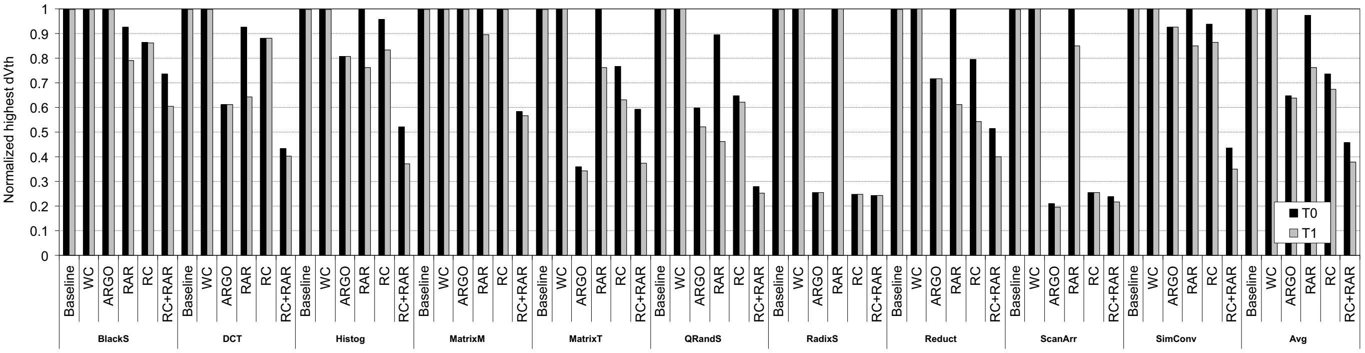


Fig. 19. Normalized highest dV_{th} considering the T_0 and T_1 SRAM cell transistors of the entire slice.

$$dV_{th} = A \times t_{ox} \times \sqrt{C_{ox} \times (V_{dd} - V_{t_0})} \times (1 - \frac{V_{ds}}{\alpha \times (V_{dd} - V_{t_0})}) \times e^{\frac{V_{dd}}{t_{ox} \times E} - \frac{Ea}{k \times T}} \times {t_{stress}}^{0.25} \times (1 - \sqrt{etha \times \frac{t_{rec}}{t_{stress} + t_{rec}}}) \quad (3)$$

The dV_{th} was computed using the standard Equation 3 [37]. All the parameters from the equation are constant values, apart from t_{stress} and t_{rec}, which refer to the amount of time that transistors are under stress and recovery modes, respectively. The reader is referred to [37] for further details.

Figure 19 plots the normalized highest dV_{th} with respect to the theoretical maximum voltage degradation that PMOS transistors can suffer from the NBTI effect after the 3-year lifetime. Results are divided into dV_{th} and T_s coming from ‘0’ and ‘1’ duty cycles, which refer to the stress time in transistors T_0 and T_1, respectively.

The highest dV_{th} comes from the baseline and WC approaches, which reach the theoretical maximum dV_{th} in all the applications according to the discussed 100% ‘0’ and ‘1’ duty cycle distributions. The ARGO technique largely reduces the dV_{th} in kernels like *DCT* and *MatrixT*, but fails to do so in those applications occupying a large portion of the slice.

In contrast, the proposed RC+RAR mechanism attacks dV_{th} by ensuring that the long duty cycles are reduced across all the SRAM cells. The degradation savings are consistent with the presented duty cycle analysis, obtaining larger dV_{th} savings than ARGO in almost all kernels. Notice too that the T_0 degradation is more remarkable than that of T_1 in most applications, confirming that ‘0’ duty cycles are usually longer than ‘1’ duty cycles. On average, the dV_{th} caused by ‘0’ and ‘1’ duty cycles is saved by RC+RAR by 54% and 62%, respectively, whereas these percentages drop down to 35% and 36%, respectively, for ARGO.

Recall that both the SNM degradation and the increase of the transistor switching delay (T_s) strongly depend on the dV_{th}, showing a very similar trend with a near constant scale factor throughout the whole device lifetime [27], [37], [17]. For instance, we have measured that the increase of the T_s is on average by 13.9%, 8.7%, and 6.0% for the baseline, ARGO, and RC+RAR, respectively, compared to a register file design with ideal NBTI-free transistors.

6.5 Impact on Performance

Compared to a conventional register file design, the proposed RC+RAR technique may negatively impact on performance due to three main design issues: i) additional MOV instructions are inserted to deal with branch divergence, ii) a power-gated register needs 10 cycles to be woken up, and iii) the eDRAM compression table requires from refresh cycles.

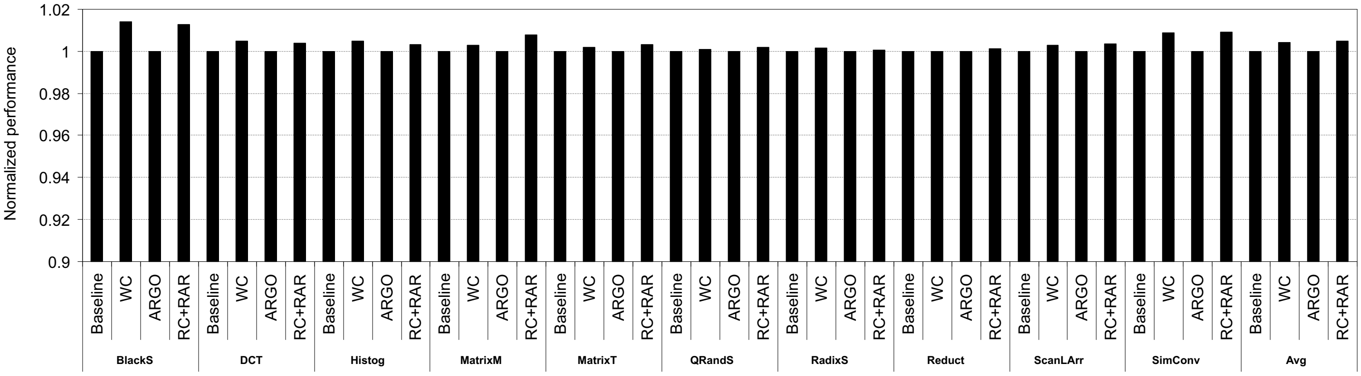


Fig. 20. Normalized performance with respect to the baseline.

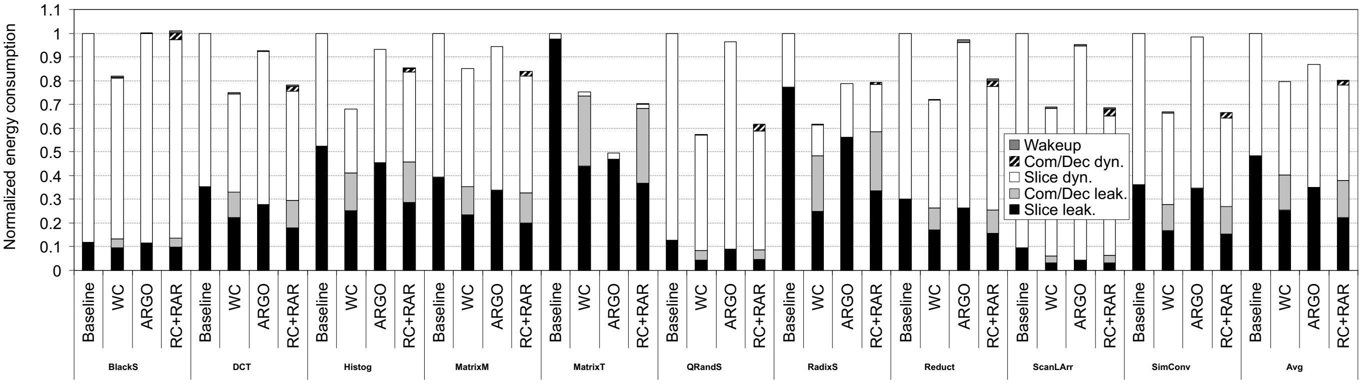


Fig. 21. Normalized energy consumption over a baseline approach without register switch off.

Figure 20 shows the normalized performance of WC, ARGO, and RC+RAR over the baseline. The ARGO approach is not affected by any of the mentioned design issues since just unused registers are switched off and register windows are just rotated during the wavefront register allocation. Thus, the execution time remains unaffected compared to the baseline scheme. In contrast, minimal performance losses can be appreciated for WC and RC+RAR. For both approaches, most of the performance degradation comes from the power-gating technique, since the number of additional MOV instructions is also negligible for WC (see Section 5.1.1), and even though WC does not require a compression table, the number of refresh operations performed by RC+RAR is rather low thanks to the distributed refresh mechanism and the relatively large eDRAM capacitors (see Section 5.1.3).

Overall, the performance differences between WC and RC+RAR mainly depend on the number of woken up registers. Compared to the conventional design, the performance degradation of WC and RC+RAR is only by 0.43% and 0.48% on average, respectively.

6.6 Energy Estimation

This section quantifies the slice energy consumption of the studied approaches. For this analysis, we assume that the baseline scheme does not switch off unused registers. Figure 21 plots the results, which correspond to a whole execution of each kernel. The consumption has been split into leakage and dynamic expenses. For WC and RC+RAR, these expenses are in turn divided into energy consumption coming from the slice and from the additional logic, that is, mainly two decompression units and one compression unit. In addition, the expenses of the compression table fall into these categories in the case of RC+RAR. The last category, namely *wakeup*, refers to the energy required to power on registers.

Compared to the baseline, the slice leakage savings of the other techniques are due to powering off unused registers. Despite obtaining a slightly higher execution time than the baseline and ARGO, the WC and RC+RAR approaches show larger slice leakage reductions since these schemes not only switch off unused registers but also power gate compressible ones. Since WC does not entirely power off registers (see Figure 12), there are some kernels like *MatrixM* and *Reduct* where RC+RAR consumes less slice leakage. In a few other kernels such as *Histog*, WC obtains better results because it compresses more registers than RC+RAR. On the other hand, the leakage consumed by the additional circuitry may nullify the overall leakage savings compared to the baseline. This is only seen in *BlackS*, which shows a high slice utilization and relatively low register compression opportunities as discussed above.

Regarding dynamic energy, WC and RC+RAR reduce this consumption with respect to the baseline and ARGO. This is mainly due to, on a read access of the first subwavefront, if the source register is compressed, the successive read accesses to the register are deactivated. The rest of dynamic expenses of the additional logic are found to be minimal with respect to the slice dynamic consumption. The same can be said for the wakeup energy, since the number of register power-on operations is on average two orders of magnitude lower than the number of slice accesses. Overall, WC, ARGO, and RC+RAR save around 20.2%, 13.1%, and 19.9% of the total slice energy consumption with respect to the baseline approach.

7 Related Work

Previous research works have attacked the NBTI effect in GPUs by dynamically powering off unused resources at runtime; e.g., the ARGO approach switches off unused register windows and modifies the wavefront register allocation to evenly distribute the unused windows along a register slice [27]. However, this approach strongly depends on the GPU architecture and the slice utilization. Those kernels with a high slice utilization have very limited power-off opportunities and aging mitigation as discussed in Sections 5.3 and 6. At a higher granularity level, there are some proposals that dynamically power off whole compute units [12], [21]. In contrast, our fine-grain proposals only depend on register compression capabilities, which allow powering off registers in spite of being actually used by the kernel.

Lofti *et al.* [24] propose software techniques to recompile kernels from the point of view of the aging impact of each instruction type and the process variation characteristics of each hardware structure. This way, the most stressful instructions are distributed among those compute units less sensitive to degradation. Tan *et al.* [36] classify register banks into fast and slow according to their access time. Based on this observation, they propose a technique that estimates at runtime the register latencies taking into account the additional delay induced by NBTI, and consequently rename register banks to extend their lifetime.

NBTI has been also attacked in CPU register files. Wang *et al.* [41] split the integer register file in half and periodically flip the contents of the half where cells store the most significant bits stuck-at '0', whereas Gong *et al.* [15] implement such cells with NBTI-resilient 8T SRAM technology. Kothawade *et al.* [20] modify the register decoding scheme to distribute the NBTI stress across the register file. The Penelope processor [3] inverts the register file contents during idle times to mitigate NBTI. However, this technique cannot be directly applied to GPU register files since the register management is performed by the compiler and assigned registers are live during all the kernel execution time.

The BDI compression algorithm has been recently used in GPU register files to save energy [22]. The compressed data are stored in the slice, resulting in memory cells that remain on throughout the kernel execution, and consequently, they are strongly affected by the NBTI phenomenon as discussed in Sections 5.3 and 6. In contrast, our proposed design makes use of an additional NBTI-free auxiliary table to store the compressed data, which allows to power off entire compressed registers. In addition, to reduce the size of such a table, our proposed compression algorithm just calculates two different deltas at most thanks to the presence of regular memory access patterns in GPU workloads, resulting in a higher compression factor than BDI.

8 Conclusions

Negative Bias Temperature Instability (NBTI) is the main deleterious effect that accelerates transistor aging over the lifetime of GPU memory structures. NBTI manifests when memory cells store a given logic value for an extended period of time, degrading the cell transistor threshold voltage V_{th} and increasing the switching delays, which could eventually result in faulty operation. This work has enhanced the GPU register file design to reduce transistor aging caused by NBTI.

Data patterns showing similarities among the register components have been analyzed. According to these observations, a data compression mechanism has been proposed to enable switching off entire registers, which induces a partial recovery from NBTI. In order to ensure an even aging across all the registers in the huge register file, a second mechanism that periodically rotates register addresses has been proposed, which distributes switch-off cycles across all the registers.

Experimental results have shown that a conventional implementation of the register file maintains memory cells storing a given logic value during the entire execution of the kernels, which implies 100% '0' and '1' duty cycle distributions, causing severe V_{th} degradation. In contrast, the proposed mechanisms are able to reduce the longest '0' and '1' duty cycle distributions by 58% and 68%, respectively, which translates into V_{th} degradation savings by more than half. Moreover, switching off registers allows the proposed mechanisms to reduce the total energy consumption by 20% with respect to the conventional design, and with minimal performance degradation (0.5% on average) and area overhead (5.4%).

Acknowledgments

This work was supported by the *Gobierno de Aragón* and the European ESF (gaZ: T58_17R research group), and by the *Ministerio de Economía y Competitividad* (MINECO) and AEI/FEDER (EU) funds under Grants TIN2016-76635-C2-1-R and TIN2015-66972-C5-1-R.

References

[1] *AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK).*

[2] *TOP500 Supercomputer Sites, available online at http://www.top500.org/.*

[3] J. Abella, X. Vera, and A. González. Penelope: The NBTI-Aware Processor. In *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarch.*, pages 85–96, 2007.

[4] Advanced Micro Devices, Inc. *AMD Graphics Cores Next (GCN) Architecture*, 2012.

[5] Advanced Micro Devices, Inc. *OpenCL Optimization Guide*, 2013.

[6] Advanced Micro Devices, Inc. *AMD Graphics Core Next Architecture, Generation 3, Reference Guide*, 2016.

[7] E. Atoofian. Compressed L1 Data Cache and L2 Cache in GPGPUs. In *Proc. IEEE 27th Int'l Conf. Appl.-Spec. Syst. Arch. Processors*, pages 1–8, 2016.

[8] K. Bernstein, D. J. Frank, A. E. Gattiker, W. Haensch, B. L. Ji, S. R. Nassif, E. J. Nowak, D. J. Pearson, and N. J. Rohrer. High-performance CMOS Variability in the 65-nm Regime and Beyond. *IBM J. Res. Dev.*, 50(4/5):433–449, 2006.

[9] A. Calimera, E. Macii, and M. Poncino. Analysis of NBTI-Induced SNM Degradation in Power-Gated SRAM Cells. In *Proc. IEEE Int'l Symp. Circuits Syst.*, pages 785–788, 2010.

[10] B. J. Campbell, G. M. Hess, and H. Huang. Leakage and NBTI Reduction Technique for Memory. *US Patent No. 8395954 B2*, 2013.

[11] F. Candel, A. Valero, S. Petit, D. Suárez-Gracia, and J. Sahuquillo. Exploiting Data Compression to Mitigate Aging in GPU Register Files. In *Proc. 29th Int'l Symp. Comp. Arch. High Perf. Comp.*, pages 57–64, 2017.

[12] X. Chen, Y. Wang, Y. Liang, Y. Xie, and H. Yang. Run-Time Technique for Simultaneous Aging and Power Optimization in GPGPUs. In *Proc. 51st ACM/EDAC/IEEE Design Automat. Conf.*, pages 1–6, 2014.

[13] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra. A Step towards Energy Efficient Computing: Redesigning A Hydrodynamic Application on CPU-GPU. In *Proc. IEEE 28th Int'l Paral. and Distr. Processing Symp.*, pages 972–981, 2014.

[14] S. Ganapathy, R. Canal, A. González, and A. Rubio. iRMW: A Low-Cost Technique to Reduce NBTI-Dependent Parametric Failures in L1 Data Caches. In *Proc. IEEE 32nd Int'l Conf. Comput. Design*, pages 68–74, 2014.

[15] N. Gong, S. Jiang, J. Wang, B. Aravamudhan, K. Sekar, and R. Sridhar. Hybrid-Cell Register Files Design for Improving NBTI Reliability. *Elsevier Microelec. Reliab.*, 52(9–10):1865–1869, 2012.

[16] Intel Corporation. *Intel® FPGA SDK for OpenCL. Best Practices Guide*, 2017.

[17] K. Kang, H. Kufluoglu, K. Roy, and M. A. Alam. Impact of Negative-Bias Temperature Instability in Nanoscale SRAM Array: Modeling and Analysis. *IEEE Trans. Comput.-Aided Design Integr. Circuits and Syst.*, 26(10):1770–1781, 2007.

[18] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *Proc. 28th Annu. Int'l Symp. Comp. Arch.*, pages 240–251, 2001.

[19] N. Khoshavi, R. A. Ashraf, and R. F. DeMara. Applicability of Power-Gating Strategies for Aging Mitigation of CMOS Logic Paths. In *IEEE 57th Int'l Midwest Symp. Circuits Syst.*, pages 929–932, 2014.

[20] S. Kothawade, K. Chakraborty, and S. Roy. Analysis and Mitigation of NBTI Aging in Register File: An End-To-End Approach. In *Proc. 12th Int'l Symp. Quality Electron. Design*, pages 1–7, 2011.

[21] H. Lee, M. Shafique, and M. A. Al Faruque. Low-overhead Aging-aware Resource Management on Embedded GPUs. In *Proc. 54th ACM/EDAC/IEEE Design Automat. Conf.*, pages 1–6, 2017.

[22] S. Lee, K. Kim, G. Koo, H. Jeon, M. Annavaram, and W. W. Ro. Improving Energy Efficiency of GPUs Through Data Compression and Compressed Execution. *IEEE Trans. Comput.*, 66(5):834–847, 2017.

[23] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. CACTI-P: Architecture-level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques. In *Proc. Int'l Conf. Comput.-Aided Design*, pages 694–701, 2011.

[24] A. Lotfi, A. Rahimi, L. Benini, and R. K. Gupta. Aging-Aware Compilation for GP-GPUs. *ACM Trans. Archit. Code Optim.*, 12(2):24:1–24:20, 2015.

[25] Micron Technology, Inc. *Various Methods of DRAM Refresh*, 1999.

[26] E. Mintarno, V. Chandra, D. Pietromonaco, R. Aitken, and R. W. Dutton. Workload-Dependent NBTI and PBTI Analysis for a sub-45nm Commercial Microprocessor. In *Proc. IEEE Int'l Reliab. Physics Symp.*, pages 1–6, 2013.

[27] M. Namaki-Shoushtari, A. Rahimi, N. Dutt, P. Gupta, and R. K. Gupta. ARGO: Aging-aware GPGPU Register File Allocation. In *Proc. Int'l Conf. Hard/Soft. Codesign Syst. Synthesis*, pages 1–9, 2013.

[28] Nvidia Corporation. *NVIDIA OpenCL Best Practices Guide*, 2009.

[29] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Base-delta-immediate Compression: Practical Data Compression for On-chip Caches. In *Proc. 21st Int'l Conf. Paral. Arch. Compil. Tech.*, pages 377–388, 2012.

[30] H. Pilo, C. A. Adams, I. Arsovski, R. M. Houle, S. M. Lamphier, M. M. Lee, F. M. Pavlik, S. N. Sambatur, A. Seferagic, R. Wu, and M. I. Younus. A 64Mb SRAM in 22nm SOI Technology Featuring Fine-Granularity Power Gating and Low-Energy Power-Supply-Partition Techniques for 37% Leakage Reduction. In *Proc. IEEE Int'l Solid-State Circuits Conf. Digest Tech. Papers*, pages 322–323, 2013.

[31] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-submicron Cache Memories. In *Proc. Int'l Symp. Low Power Electron. Design*, pages 90–95, 2000.

[32] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke. IBM Power9 Processor Architecture. *IEEE Micro*, 37(2):40–51, 2017.

[33] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46, 2016.

[34] H. Tabkhi and G. Schirner. Application-Guided Power Gating Reducing Register File Static Power. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 22(12):2513–2526, 2014.

[35] K. Takeda, Y. Hagihara, Y. Aimoto, M. Nomura, Y. Nakazawa, T. Ishii, and H. Kobatake. A Read-Static-Noise-Margin-Free SRAM Cell for Low-VDD and High-Speed Applications. *IEEE J. Solid-State Circuits*, 41(1):113–121, 2006.

[36] J. Tan, M. Chen, Y. Yi, and X. Fu. Mitigating the Impact of Hardware Variability for GPGPUs Register File. *IEEE Trans. Paral. Dist. Syst.*, 27(11):3283–3297, 2016.

[37] A. Tiwari and J. Torrellas. Facelift: Hiding and Slowing Down Aging in Multicores. In *Proc. 41st Annu. IEEE/ACM Int'l Symp. Microarch.*, pages 129–140, 2008.

[38] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proc. 21st Int'l Conf. Paral. Arch. Compil. Tech.*, pages 335–344, 2012.

[39] A. Valero, N. Miralaei, S. Petit, J. Sahuquillo, and T. M. Jones. On Microarchitectural Mechanisms for Cache Wearout Reduction. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 25(3):857–871, 2017.

[40] R. Vattikonda, W. Wang, and Y. Cao. Modeling and Minimization of PMOS NBTI Effect for Robust Nanometer Design. In *Proc. 43rd ACM/IEEE Design Automat. Conf.*, pages 1047–1052, 2006.

[41] S. Wang, T. Jin, C. Zheng, and G. Duan. Low Power Aging-Aware Register File Design by Duty Cycle Balancing. In *Proc. Design, Automat. Test Europe Conf. Exhibit.*, pages 546–549, 2012.

[42] P. Xiang, Y. Yang, M. Mantor, N. Rubin, L. R. Hsu, and H. Zhou. Exploiting Uniform Vector Instructions for GPGPU Performance, Energy Efficiency, and Opportunistic Reliability Enhancement. In *Proc. 27th Int'l ACM Conf. Supercomp.*, pages 433–442, 2013.

Alejandro Valero received the PhD degree in Computer Engineering from the Universitat Politècnica de València, Spain, in 2013. From 2013 to 2015, he was a Visiting Researcher with Northeastern University, Boston, MA, USA, and the University of Cambridge, UK. Since 2016, he has been an Assistant Professor with the Department of Computer and Systems Engineering at the Universidad de Zaragoza, Spain. His current research interests include GPU architectures, memory hierarchy design, energy efficiency, and reliability. He is a member of the Aragon Institute of Engineering Research (I3A) and the HiPEAC European NoE.

Francisco Candel received the BS and MS degrees in Computer Engineering from the Universitat Politècnica de València (UPV), Spain, in 2012 and 2014, respectively. He is currently working towards a PhD degree at the Department of Computer Engineering (DISCA) of the same university. His PhD research focuses on GPU modeling and efficient memory hierarchies for future GPUs.

Darío Suárez-Gracia (S08,M12) received the PhD degree in Computer Engineering from the Universidad de Zaragoza, Spain, in 2011. From 2012 to 2015, he was at Qualcomm Research Silicon Valley. Since 2015, he has been an Interim Associate Professor at the Universidad de Zaragoza. His research interests include parallel programming, heterogeneous computing, memory hierarchy design, and energy-efficient processor and network-on-chip microarchitectures. Dr. Suárez Gracia is a member of the IEEE, the IEEE Computer Society, the ACM, and the HiPEAC European NoE.

Salvador Petit (M'07) received the PhD degree in Computer Engineering for the Universitat Politècnica de València (UPV), Spain. Since 2009, he has been an Associate Professor with the Computer Engineering Department, UPV, where ha has been teaching several courses on computer organization. He has authored over 100 refereed conference and journal papers. His current research interests include multithreaded and multicore processors, memory hierarchy design, task scheduling, and real-time systems. Dr. Petit is a member of the IEEE Computer Society. In 2013, he received the Intel Early Career Faculty Honor Program Award.

Julio Sahuquillo (M'04) received the BS, MS, and PhD degrees from the Universitat Politècnica de València, Spain, all in Computer Engineering. He is a Full Professor with the Department of Computer Engineering at the Universitat Politècnica de València. He has taught several courses on computer organization and architecture. He has authored over 120 refereed conference and journal papers. His current research interests include multi- and manycore processors, memory hierarchy design, cache coherence, GPU architecture, and architecture-aware scheduling. Dr. Sahuquillo is a member of the IEEE Computer Society.