IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 19, 2020, accepted September 19, 2020, date of publication September 22, 2020,
date of current version October 1, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3025899

DC-Patch: A Microarchitectural Fault Patching Technique for GPU Register Files

ALEJANDRO VALERO, DARÍO SUÁREZ-GRACIA, (Member, IEEE), AND RUBÉN GRAN-TEJERO

Department of Computer Science and Systems Engineering, Universidad de Zaragoza, 50018 Zaragoza, Spain

Corresponding author: Alejandro Valero (alvabre@unizar.es)

This work was supported in part by the Agencia Estatal de Investigación (AEI) and European Regional Development Fund (ERDF) under Grant TIN2016-76635-C2-1-R and Grant PID2019-105660RB-C21, in part by the Aragon Government and European Social Fund (ESF) under Grant gaZ: T58_20R research group, in part by the ERDF under Grant 2014-2020 Construyendo Europa desde Aragón, and in part by the Universidad de Zaragoza under Grant JIUZ-2019-TEC-08.

ABSTRACT The ever-increasing parallelism demand of General-Purpose Graphics Processing Unit (GPGPU) applications pushes toward larger and more energy-hungry register files in successive GPU generations. Reducing the supply voltage beyond its safe limit is an effective way to improve the energy efficiency of register files. However, at these operating voltages, the reliability of the circuit is compromised. This work aims to tolerate permanent faults from process variations in large GPU register files operating below the safe supply voltage limit. To do so, this paper proposes a microarchitectural patching technique, DC-Patch, exploiting the inherent data redundancy of applications to compress registers at run-time with neither compiler assistance nor instruction set modifications. Instead of disabling an entire faulty register file entry, DC-Patch leverages the reliable cells within a faulty entry to store compressed register values. Experimental results show that, with more than a third of faulty register entries, DC-Patch ensures a reliable operation of the register file and reduces the energy consumption by 47% with respect to a conventional register file working at nominal supply voltage. The energy savings are 21% compared to a voltage noise smoothing scheme operating at the safe supply voltage limit. These benefits are obtained with less than 2 and 6% impact on the system performance and area, respectively.

INDEX TERMS Computer architecture, data compression, energy efficiency, fault tolerance, memory management, registers, SRAM cells, vector processors.

I. INTRODUCTION

For more than a decade, Graphics Processing Units (GPUs) have established themselves as a massively parallel computing device and have been adopted in multiple computing areas, from embedded systems to high-performance data centers. This success has ultimately resulted in a plethora of applications coded and optimized to run on GPU devices (GPGPU applications). Within those, emerging applications such as deep learning, analytics, or big data mining push toward GPU architectures with higher computational and memory resources while maintaining power under control. As a consequence, energy efficiency has become a major concern for modern GPU architecture designs.

The register file is one of the largest and most energy-hungry structures in a GPU, consumes approximately 20% of the total GPU energy [22], and grows generation after generation. For instance, the register file of the NVIDIA Tesla V100, with 20 MB, is more than 5× larger than its counterpart in the Tesla K40 [35]. Many approaches focus on energy efficiency in the register file, from traditional techniques like clock and power gating to recent research including the exploitation of the data access behavior [3], liveness analysis [26], prefetching [38], data redundancy [27], register sharing [21], or coalescing techniques [9].

On the other hand, like any other digital circuit, GPU register files are affected by static and dynamic variation effects. Static variations are a consequence of the chip manufacturing process (e.g., process variations), whereas dynamic variations come from the circuit operation (e.g., voltage noise and aging effects). Process variations impose a minimum safe supply voltage limit (V_{min}) for each memory cell to be reliable and, in turn, for an entire register file, V_{min} is set to the highest V_{min} from the worst-case cell.

The associate editor coordinating the review of this manuscript and approving it for publication was Junaid Shuja.

A high supply voltage (V_{dd}) over V_{min} ensures a sufficient guardband for a safer operation against sudden V_{dd} droops but speeds up circuit aging. In addition, a high V_{dd} results in energy wasting since energy scales quadratically with V_{dd} and large supply voltage noise is a rare event [29]. In this context, prior work has proposed voltage noise smoothing schemes for GPU register files that allow relaxing the guardband by pushing V_{dd} toward V_{min} at a fixed frequency [28], [29]. Scaling V_{dd} below V_{min} is a challenging task due to the high number of permanent faults as a result of surpassing the V_{min} of multiple cells [39]. Contrary to lone faults induced by either dynamic variations or particle strikes, the high number of uprising permanent faults when $V_{dd} < V_{min}$ are far from conventional Error-Correcting Code (ECC) capabilities, requiring not only larger storage capacities and energy consumption but also complex and slow encoders/decoders to ensure a reliable operation [25], [46].

With the aim to further improve energy efficiency without using costly ECCs, *patching* is being explored as a solution to tolerate permanent faults due to process variations in GPU register files operating at low voltages below V_{min}. This approach consists of disabling faulty register entries and providing alternative and reliable entries where faulty register accesses are redirected (patched). In this context, prior work has identified reliable entries containing useless data at compile-time, and has patched faulty accesses to such entries at run-time thanks to Instruction Set Architecture (ISA) modifications [42]. However, modifying the ISA has the following downsides: i) exposes unnecessary implementation details to the programmer, ii) requires software changes to exploit the mechanism, and iii) increases the ISA complexity (backwards-compatibility and future extensions). On the contrary, we present a novel pure-microarchitectural patching technique, DC-Patch, that enables faulty entries for patching purposes by exploiting the inherent data register compression of GPGPU applications at run-time.

Effective data compression requires the presence of regular patterns in the streams. Fortunately, many GPU programming idioms generate regular memory access patterns and avoid branch divergence, which entails that applications store regular data patterns in GPU registers [5], [18], [34], [47]. Such regular patterns can be compressed using variations of the Base-Delta-Immediate (BDI) algorithm originally proposed to compress CPU cache lines [37]. These compression algorithms can be applied to any modern GPU architecture from either NVIDIA or AMD, since the exploited data patterns exclusively come from the single-instruction multiple-thread programming model provided by CUDA or OpenCL.

Recent work focusing on GPU register files leverage data compression to save energy [27], [48], [50], to mitigate transistor aging phenomena [44], or to deal with transient faults [32]. However, to the best of our knowledge, compression has not been previously exploited to circumvent permanent faults in GPU register files. Moreover, this is the first work that proposes a patching mechanism for GPU register files without introducing any complexities or added burden to software or the programmer.

Modern GPUs from either NVIDIA or AMD usually divide the register file entries into smaller register *blocks*. This is because a Single Instruction, Multiple Data (SIMD) unit has a number of lanes equal to the block size. That is, a SIMD unit is capable of executing a single block in a given processor cycle, meaning that an entire register is accessed and executed in successive cycles [4], [31]. We exploit such an inherent GPU architectural organization to patch as much compressed registers as the number of blocks in a sole entry, which enables faulty entries with a minimal design overhead.

Experimental results show that DC-Patch guarantees a reliable operation of a register file with 39% of faulty register entries. Such a fault tolerance reduces energy on average by 47 and 21% with respect to a conventional register file working at nominal V_{dd} and at the safe V_{min} voltage, respectively, whereas the impact on performance and area is less than 2 and 6%.

The remainder of this paper is organized as follows. Section II presents the background of this work. Section III motivates the use of data compression strategies for patching purposes. Section IV introduces the proposed DC-Patch technique. Section V discusses experimental results. Section VI comments on related work, and finally, Section VII concludes this paper.

II. BACKGROUND

This section summarizes the GPU register file architecture, the reliability model and scenarios used to evaluate the proposal, and the exploited data compression strategy to combat process variations in the GPU register file.

A. GPU REGISTER FILE

Current GPUs consist of tens of in-order processors, also known as Streaming Multiprocessors (SMs) and Compute Units (CUs) in NVIDIA and AMD GPUs, respectively. Without loss of generality, this work uses the AMD Graphics Core Next (GCN) family of GPUs as a driving example [4]. Thus, AMD terminology is used throughout the paper.

A CU consists of 4 SIMD units. Each SIMD unit works with a 64 KB slice of the register file, totalling a 256 KB register file per CU. Figure 1 shows the pipeline stages of an access to a slice, and how 256 vector register entries compose a slice. In turn, every entry consists of 64 components of 4 bytes each totalling 256 B. To access to these register entries, 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 entry, but with a component shift based on the thread id in the wavefront. This way, although each thread works with a different component of the same entry, referring to each component is avoided in the ISA.

Since a SIMD unit consists of 16 lanes (64 B), the threads of a wavefront execute a given instruction forming 4 bundles of 16 adjacent threads, referred to as *blocks* (blk_i) in the

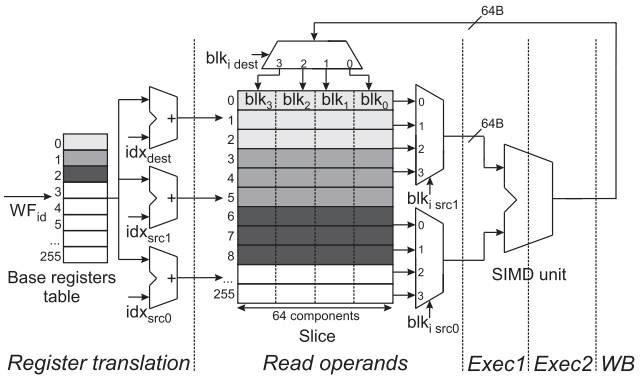

FIGURE 1. Base slice pipeline stages after fetch and decode stages.

figure. These blocks are accessed in lockstep mode over four successive cycles. Since AMD instructions usually have two source operands (*src0* and *src1*) and one destination operand (*dest*), a slice incorporates two read ports and one write port. Notice too that, according to the pipeline depth, writing *blk0* of the destination register entry into the slice and reading *blk3* from the source register entries take place at the same cycle.

The register entries of a slice are statically distributed among the wavefronts running on the corresponding SIMD unit. To do so, when a wavefront is allocated to a SIMD unit, it is given an id (WF_{id}), the physical address of its base register, and a number of contiguous allocated entries, which is a constant value for all the wavefronts of a given application and does not change throughout the application execution [7]. The set of entries accessed by a wavefront is referred to as the register window of the wavefront. For instance, Figure 1 shows 3-register windows in the slice highlighted with a greater gray gradient for wavefronts WF_0, WF_1, and WF_2.

As shown in the register translation stage, wavefront instructions refer to *logical registers* within their window with indexes (*idx*). These indexes are added to the base register to obtain the physical register entries. The base register entry is retrieved from a register base table indexed with the wavefront id. This table requires as much as 256 rows to support the uttermost case of wavefronts with 1-register windows. Finally, each running wavefront receives a different register base, so that register windows do not overlap.

B. RELIABILITY MODEL AND SCENARIOS

Lowering the V_{dd} of a circuit is an effective way to attain substantial energy savings. However, at low V_{dd}, transistors are more vulnerable to process variations, which manifest across a chip as static and spatial fluctuations around the nominal values in transistor parameters like the threshold voltage (V_{th}) and the channel length (L_{eff}) [10], [11]. Such fluctuations can lead to permanent transistor faults; for instance, when the V_{th} of a transistor surpasses the V_{dd} of the circuit.

Systematic strongly correlated effects such as lithographic aberrations and random uncorrelated effects like random dopant fluctuation induce process variations [23], [24]. Systematic variations cause neighboring transistors to share similar parameters, whereas random variations alter the transistor parameters uniformly across the chip [42].

The distribution of permanent faults in an SRAM memory array depends on the impact ratio of systematic and random variations. We evaluate the proposal under three different reliability scenarios: *common*, *clustered*, and *dispersed*. The *common* scenario refers to a commonly used ratio for process variations where both systematic and random effects are equally treated [23], whereas *clustered* and *dispersed* refer to scenarios with a greater impact of systematic and random variations, respectively.

TABLE 1. Supply voltage and percentage of register entries with different number of faulty bits for each reliability scenario [42].

Reliability scenario	V_{dd} (mV)	Reliable entries		Faulty entries		
		0-bit	1-bit	2-bit	3-bit	≥4-bit
Common	419	34	33	20	10	3
Clustered	497	43	20	12	10	15
Dispersed	371	26	35	23	12	4

Table 1 shows the supply voltage and the distribution of the register entries of a slice according to their number of faulty bits for each reliability scenario. We assume the 28-nm fault model proposed in [42] and obtained with VARIUS [24]. The undervolting margins from the nominal V_{dd} are 38, 48, and 54% for *clustered*, *common*, and *dispersed*, respectively. Such undervolting margins are close to those of a recent 14-nm FinFET reliability model [17]. However, we refer to a 28-nm model because that is the smallest-node with a synthesis library available to us (see Section IV-E). Anyway, since the undervolting margins are similar to [17], we believe that the presented energy savings would be similar in a 14-nm node.

The assumed reliability model focuses on the register file by assuming an implementation with dedicated voltage domains for logic and arrays, keeping logic at a high V_{dd} to avoid faults as described in previous academic work [14], [52], patents [13], and commercial devices [8]. Each reliability scenario operates at a fixed V_{dd} (below the safe 600 mV V_{min} for 28 nm [51]) throughout the execution of an application [36], [42]. For the sake of clarity, entries with four or more faulty bits are grouped together.

Error-Correcting Pointer (ECP) is an approach correcting permanent faults by encoding the locations of faulty bits into a table and assigning additional replacement bits to replace faulty bits [40]. In this work, ECP is employed at a reasonable granularity of register entry with a single replacement bit per entry. Thus, entries with less than two faulty bits are treated as reliable. We conservatively assume that the faulty bits are uniformly distributed across the four blocks of a register entry. That is, faulty entries with i faulty bits, $i \geq 2$, have i faulty blocks. Of course, when $i \geq 4$, all the blocks are faulty and the entry is considered as completely useless.

The *common* scenario shows a register distribution where the percentage of entries reduces with the number of faulty bits within an entry. In contrast, the *clustered* scenario, where

the systematic effect dominates over the random effect, shows a higher percentage of entries in the edges of zero and four or more faulty bits. The *dispersed* scenario, where faults are randomly distributed, presents a higher percentage of entries with at least one faulty bit, but does not show as much completely faulty entries as the *clustered* scenario. Overall, the percentage of faulty register entries in a slice is 33, 37, and 39% for *common*, *clustered*, and *dispersed*, respectively.

Finally, each reliability scenario has a different 256×4 faulty map per slice according to the location of the faulty register blocks. These maps are determined during a post-fabrication testing. At a given V_{dd}, all the slice cells are tested by comparing a bit to be written in a cell with the read-after-written bit from the cell. In the case of a mismatch, the cell fails the test and the corresponding block is marked as faulty in the map [36], [41], [42]. The faulty map will be used as input for DC-Patch to differentiate between reliable and faulty register entries and blocks. Note that these faulty maps apply to permanent faults from process variations. Other types of faults induced by either dynamic variations or particle strikes, which are out of the scope of this paper, can be covered with recent ECC approaches [45].

C. DATA COMPRESSION

This paper leverages the data compression mechanism proposed in [44], where three different regular data patterns are identified, namely *constant*, *single*-Δ, and *double*-Δ patterns. The *constant* pattern appears when all the components of a register entry store the same scalar value. This pattern is common in practice due to divergence control [47]. The *single*-Δ pattern occurs when a register entry stores a sequence of values where the difference, referred to as delta, between successive components is constant and greater than zero. Examples are registers storing thread ids or the addresses of an array. Finally, register entries showing a *double*-Δ pattern are those in which the register is divided in subsets of components. A first delta value refers to the constant difference between the first component of a subset and the first component from the successive subset, and so on. Like the previous pattern, a second delta value refers to the constant difference between adjacent components within the subsets. Register entries linearly storing the addresses of a matrix typically show this pattern. In addition, this pattern also appears when using programming techniques like tiling or sliding windows.

These patterns offer a high compression ratio, since a single base component and two delta values (totalling 4.88 bytes) are needed to completely unwind an entire 256-byte register. The required hardware units to compress/decompress a register at run-time consist of a number of adders, subtractors, comparators, and tiny memory buffers to cope with the pipelined 4-cycle read and write operations in a slice according to the number of blocks in a register entry.

A decompression unit receives a block with the compressed data of a register and outputs each uncompressed block in a successive cycle, completing the process after four cycles. Similarly, a compression unit receives each uncompressed block one after the other, computes the compressed data in the first cycle, and determines if the complete register can be compressed in the fourth cycle after all blocks have been examined. If the entire register cannot be compressed with a single base and two deltas, it is considered as uncompressible. Further implementation details of these units can be found in [44].

Divergent slice writes in a compressed register require to act on an uncompressed register before performing the write operation, since only some components are updated. In such cases, the compressor is disabled and an additional MOV instruction is injected into the pipeline to force the register decompression before the divergent write operation.[1]

III. MOTIVATION

This section explores the potential of a fault patching technique based on data compression. To do so, the coverage of the compression algorithm is firstly analyzed by studying the percentage of compressed and uncompressed register writes. Secondly, register entries are classified depending on the compression state of the contents and the reliability state of the entry.

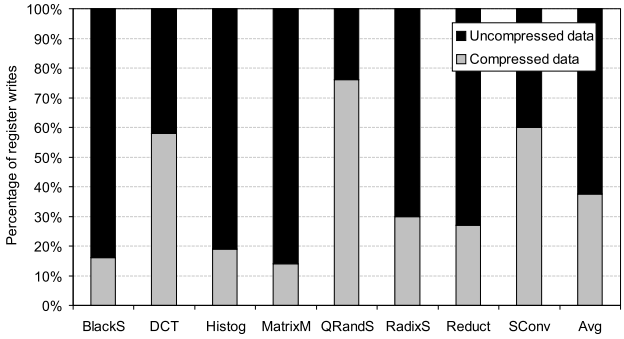

FIGURE 2. Percentage of register writes divided in writes storing compressed or uncompressed data.

Figure 2 illustrates the register writes split in those storing compressed or uncompressed data in a conventional register file exploiting the compression algorithm described in the previous section. Results are shown for a representative subset of applications from the OpenCL SDK 2.5 benchmark suite with various characteristics like different memory requirements, register file utilization and stress, or data compression opportunities [6]. The reader is referred to Section V for further details about the experimental environment.

Results show that the compression algorithm has a high potential, since almost 40% of the register writes on average compress the registers. The compression coverage largely differs between applications (from 14 to 76% in *MatrixM* and *QRandS*, respectively) mainly due to their different required

[1]Experimental results show that the impact on performance of the additional MOV instructions is negligible, since a 4% of writes on average are divergent but only a 0.13% refer to compressed registers.

data types and structures. Applications with a low compression coverage usually demand a high number of scalar and floating-point data types, which are less likely to be compressed. On the other hand, benchmarks with a high compression coverage usually exploit multiple data dimensions (vectors and matrices), integer data types, and thread ids.

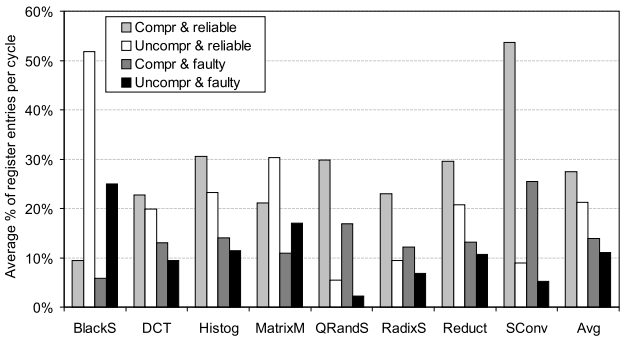

FIGURE 3. Average percentage of reliable and faulty entries per cycle storing compressed and uncompressed registers under the *common* reliability scenario.

Figure 3 classifies the register entries into four categories depending on the compression state of the register contents (i.e., compressed or uncompressed) and the reliability state of the entry (i.e., reliable or faulty). Each bar shows the average percentage of occupied entries per cycle. For illustrative purposes, results are only shown for the *common* reliability scenario.

Nearly 30% of the entries on average are reliable and store a compressed register by default in each cycle. These entries are prime candidates for patching, since the proposed DC-Patch technique preferably forces a reliable entry to be patched with an uncompressed register. Results also show that a 22% of the entries are reliable and already store an uncompressed register. Thus, they are not appropriate for patching purposes. A reduced percentage of 10% of the entries are faulty and store an uncompressed register. A register in such a faulty entry requires from a patch operation to a reliable entry, otherwise the data integrity would be compromised with the presence of faults. Finally, a 14% of the entries are faulty but store a compressed register. In this case, a patch would not be required as long as the compressed register is stored in a reliable block within the faulty entry. In fact, the remaining reliable blocks (if any) of such an entry can store other compressed registers, which results in a faulty entry storing multiple compressed registers.

Results for both *clustered* and *dispersed* scenarios are similar to those of the *common* scenario. The largest differences are seen between *common* and *dispersed*. Since the latter has a higher number of faulty entries, the percentage of faulty entries storing uncompressed and compressed registers increase on average to 13 and 17%, respectively, whereas the percentages of reliable entries storing uncompressed and compressed registers are reduced to 20 and 25%.

Overall, data compression provides promising opportunities for patching, since all the benchmarks, apart from *BlackS*, show a higher percentage of reliable entries storing compressed registers over faulty entries storing uncompressed registers. Note that unallocated entries also offer patching opportunities for either compressed or uncompressed registers. The register file utilization depends on: i) the GPU architecture, like the number of slice entries, the maximum number of concurrent wavefronts allocated to a slice, and the wavefront register window size [7], and ii) application optimizations [1]. In Figure 3, the sum of each category for a given application gives the percentage of its register file utilization. Depending on the benchmark, these percentages range from 54 (*QRandS*) to 93% (*SConv*), with an average of 74%. This percentage is higher than those from previous work reporting the average register file utilization [1], [44].

IV. DATA COMPRESSION-AWARE PATCHING TECHNIQUE

This section introduces our proposed DC-Patch approach. First, an overview of DC-Patch is presented, discussing how the patch selection algorithm works and identifying the main design components of the approach. Then, these components are described in detail, including the involved operations on each of them. Finally, timing, energy, power, and area overheads are also analyzed.

A. OVERVIEW

DC-Patch dynamically patches registers to slice entries depending on the register compression state at run-time. A patch operation is required when a register is written for the first time, and also when subsequent writes change its compression state from compressed to uncompressed and vice versa. This means that the previous patch entry of the register is released, and a new patch referring to a faulty or a reliable entry is assigned to the register depending on its new state. Uncompressed registers are always patched to reliable entries, whereas compressed registers can be patched to either reliable or faulty entries. The proposal prioritizes compressed registers to be patched in faulty entries, since otherwise reliable entries would be wasted.

The proposed approach exploits the inherent pipelined access of a 256-byte register entry in 64-byte blocks. That is, a reliable entry is able to patch up to four compressed registers, each one in a different block. On the other hand, faulty entries are able to patch compressed registers only in reliable blocks, whereas faulty blocks are disabled. This way, contrary to prior work [42], data compression enables the use of faulty entries for patching purposes. Note that a compressed register (4.88 bytes) occupies a small fraction of the much larger 64-byte block, leaving room for more than a single compressed register to be patched within a block. However, this would lead to a more complex design with a larger energy and area wasting.

Figure 4 plots the pipeline stages of a slice after the fetch and decode of an instruction, including the main additional DC-Patch design components colored gray. At a glance,

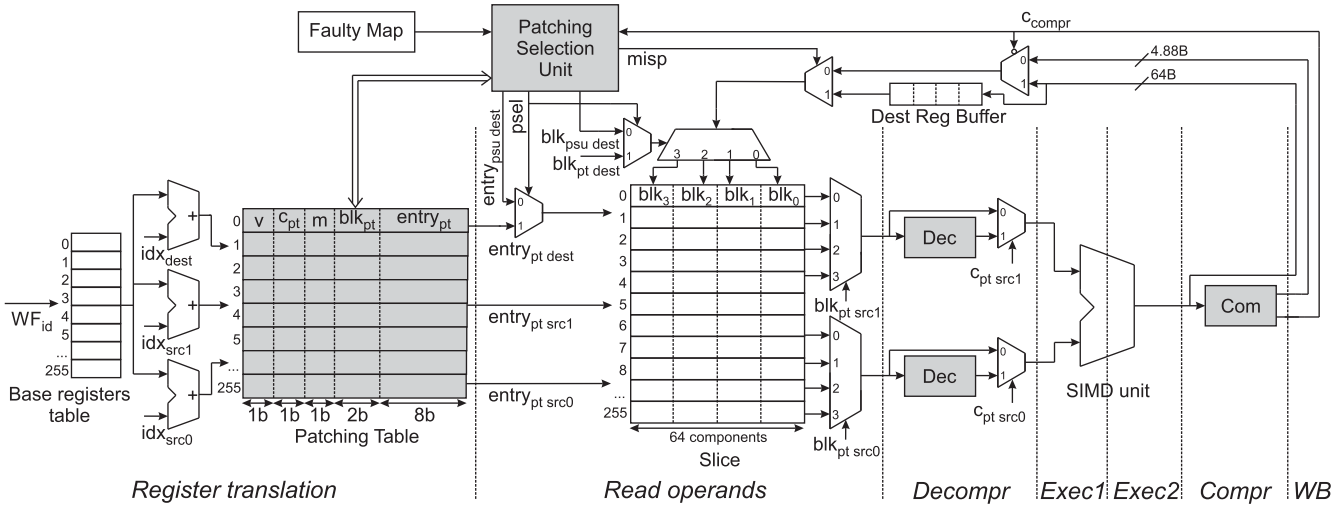

FIGURE 4. Slice pipeline stages including the main DC-Patch design components colored gray.

the Patching Table (PT) is located after the register translation and forwards to the next stage the patch entry of every register. The compression and decompression (Com/Dec) units have dedicated pipeline stages. The latter is placed after the read ports of the slice and ensures that the SIMD unit operates with uncompressed blocks from source registers each cycle. After examining a destination block, the compression unit outputs the compressed data (if any) to the slice write port. Finally, the Patching Selection Unit (PSU) is located at the writeback stage. Taking into account the faulty map of the reliability scenario, the PSU outputs a new patch when the compression state of a destination register changes. Below, we describe each of these components and the involved operations in detail.

B. PATCHING TABLE (PT)

The PT is 416-byte memory array indexed by the physical source and destination registers of an instruction. The number of rows in the PT equals the number of slice entries (i.e., 256), and they are also arranged in wavefront register windows. In fact, the PT completely decouples the register windows to the slice to maximize the chance to find a patch. In other words, patches allow to dynamically allocate registers from a wavefront in any available slice entry.

The contents of a PT row are updated by the PSU when a new patch is necessary. Each PT row contains the patch entry ($entry_{pt}$) where a register resides in the slice. For a compressed register, the blk_{pt} bits indicate in which block within the entry is located the compressed register. These bits are also used in the slice read ports ($blk_{pt\ src_i}$ bits) to forward a source block to the next stage in a sole cycle. Similarly, when the state of a destination register remains as compressed (i.e., a new patch is not required from the PSU), the blk_{pt} bits of such a register control the write port ($blk_{pt\ dest}$ bits) to store the compressed result from the SIMD unit to the proper slice block in a single cycle. In the case of an uncompressed register, the blk_{pt} bits are not used to control the slice ports. Instead, all the register blocks are accessed over four cycles.

The v valid bit and the c_{pt} compression bit define the state of a register. When set to logic '1', they refer to a valid and a compressed register, respectively. The PSU uses these bits to obtain a patch for a destination register. In addition, the c_{pt} bit drives the 2:1 muxes from the decompression stage to select between either uncompressed blocks from a read port or decompressed blocks from a Dec unit.

Note that if a patch entry cannot be found for either a compressed or an uncompressed register, a spilled patch is assigned to the register. That is, an additional memory structure stores the register contents. In this work, instead of polluting or reducing the effective storage capacity of the memory hierarchy [42], we exploit the Local Data Share (LDS) cache to cope with spilled registers.[2]

The m bit indicates whether the patch of a register refers to a spill ($m = 1$) or not ($m = 0$). If $m = 1$, instead of accessing the slice, an additional load access to the LDS cache is triggered to obtain the requested data. For this purpose, the LDS is partitioned in two halves, one for regular LDS data and the other for spilled registers. Spilled registers are arranged as a contiguous array in the LDS. In the PT, the $entry_{pt}$ bits of spilled registers refer to an offset from the base address of the spill partition where the register is located. The small impact on performance of spill register management by the LDS is quantified in Section V-A2.

C. COMPRESSION AND DECOMPRESSION (COM/DEC) UNITS

According to the number of slice ports, DC-Patch requires one Com unit and two Dec units. Contrary to the PT, the latency of these units forces to enlarge the pipeline depth in two additional stages. The impact on performance

[2]The LDS is shared among the SIMD units of a CU. For GPGPU applications, the use of the LDS is programmer-dependent. Applications can load or store data in the LDS to amplify the cache bandwidth, to avoid polluting the cache hierarchy with scatter and gather accesses, or to perform atomic operations within a work-group [4]. This cache is referred to as the shared memory in NVIDIA counterparts.

of the enlarged pipeline depth is discussed in Sections IV-E and V-A4. The input of a Dec unit comes from the read port and refers to a block containing the compressed data of a source register. In such cases, this unit unwinds the four register blocks and forwards one after the other to the SIMD unit in successive cycles. On the contrary, blocks from uncompressed registers simply bypass the Dec unit.

The Com unit receives from the SIMD unit an uncompressed block in each cycle. When this unit receives the first block of a register, it determines the compression state of the block and notifies such a state to the PSU with the c_{compr} bit, which is set to '1' or '0' when the contents are compressed or uncompressed, respectively. If $c_{compr} = 1$, the compressor also outputs the compressed data to the write port. Otherwise the four blocks of a register are simply forwarded to the write port. The c_{compr} bit also controls a 2:1 mux to forward either the compressed or uncompressed data to the write port.

Note that the compression of a register is speculative, since the register is not known to be definitely compressed until the fourth block is checked. For this reason, a Destination Register Buffer (DRB), as large as a slice entry, stores the four blocks from the SIMD unit while a register is partially identified as compressible. On a compression mispeculation, the PSU is notified by changing the state of the c_{compr} bit, and the pipeline is stalled during four cycles until the entire register from the DRB is written to the slice. The impact on performance of the mispeculation is analyzed in Section V-A3.

An alternative solution to remove the compression speculation would be to include an additional slice write port enabling two simultaneous writes, each one referring to a different compression state of a register. This way, once the final compression state is determined, the write operation referring to the wrong register would be invalidated. However, this solution may incur in a significant design complexity and additional energy and area overheads.

D. PATCHING SELECTION UNIT (PSU)

The PSU consists of a 256×4 bitmap referring to each slice block and two priority encoders with 1024 and 256 inputs each to select faulty and reliable entries, respectively. The bitmap is preset to the faulty map of the chosen reliability scenario and updated throughout the execution of an application with the occupied and released patches. Of course, faulty blocks will be permanently marked as occupied in the bitmap and not eligible for patching. According to the bitmap state, the 1024-input encoder selects a free block from a faulty entry to patch a compressed register, whereas the 256-input encoder selects an entire and free reliable register entry (four consecutive blocks) to patch an uncompressed register. The PSU outputs one of these selections depending on the compression state of the destination register from the compressor (c_{compr} bit input). The remaining inputs of the PSU are the PT contents of the destination register.

For each instruction with a destination register, the PSU enables an encoder to obtain a *preventive* patch while the instruction traverses the pipeline. This preventive patch refers to an inverse patch of the current register. For instance, if the current patch refers to a reliable entry ($c_{pt} = 0$), the PSU preemptively selects a reliable block from a faulty entry, assuming that the compression state will change. This way, when the first block of a register reaches the writeback stage, the PSU compares the c_{pt} and c_{compr} bits, and drives the write port accordingly (*psel* bit), either selecting the new patch ($entry_{psu\ dest}$ and $blk_{psu\ dest}$) or the previous patch from the PT. Preventive patches allow the PSU to be located in the writeback stage without increasing the pipeline depth. If a register is written for the first time ($v_{dest} = 0$), the PSU enables both encoders to obtain two preventive patches. Once the instruction leaves the pipeline, the PSU releases the unused patch and updates the PT if necessary.

In the case of a compression mispeculation, the PSU selects the reliable patch and the uncompressed blocks from the DRB are written to the slice one after the other ($misp = 1$). Notice too that, when an instruction writes to a spilled register ($m_{dest} = 1$), the PSU tries to find a new slice patch for the register in order to mitigate subsequent LDS latency penalties, obtaining two preventive patches as in the first write to a register. On the other hand, when a new patch cannot be found in the slice, the PSU sets the m_{dest} bit to '1' and updates the $entry_{pt\ dest}$ with the LDS spill partition offset where the register is allocated.

Finally, when a wavefront is deallocated from the slice because it has finished the execution, the PSU accesses the wavefront window of the PT, invalidating all the PT rows ($v = 0$) of the wavefront and releasing all the corresponding patches in the PSU bitmap.

E. TIMING, ENERGY, POWER, AND AREA ESTIMATIONS

This section estimates the timing, energy, power, and area of the main DC-Patch design components. The memory structures like the slice and the PT have been modeled with CACTI 7.0 [30], whereas the combinational logic and flip-flops present in the Com/Dec and PSU units have been synthesized with Synopsis Design Compiler and simulated with Mentor Graphics Modelsim. The technology library corresponds to a low-V_{th} 28 nm technology available to academia. For timing closure, we assume the 1 GHz clock frequency of the AMD GCN HD 7770 GPU, which is the GPU device considered in all the experiments.

Table 2 shows the results. The access time and dynamic energy (from read and write operations) of the slice refer to accessing a 64-byte block. For the Com/Dec units, PT, and PSU, these parameters refer to evaluating a block, accessing an entry, and obtaining a new patch, respectively. However, for the PSU, the access time refers to exclusively forwarding the new patch to the slice write port in the writeback pipeline stage, since preventive patches are obtained in prior stages. Notice too that the dynamic expenses from read and write operations are not applicable in some DC-Patch components due to they are not involved in a read or a write access to the slice. The results of the slice are shown for the three studied

TABLE 2. Timing, energy, power, and area values for a 28-nm technology node and a 1 GHz clock frequency. N/A: Not applicable.

	Slice @Conv	Slice @Comm	Slice @Clus	Slice @Disp	Com unit	Dec unit	Patching Table	Patching Sel. Unit
Access time (ns)	0.85				0.95	0.95	0.09	0.11
Read energy (pJ)	247.38	84.38	84.90	68.25	N/A	0.62	0.54	N/A
Write energy (pJ)	302.23	97.68	99.76	78.33	0.72	N/A	0.50	0.22
Static power (mW)	58.58	30.79	35.18	27.73	6.67	7.14	0.41	15.41
Area (mm^2)	0.680				0.005	0.007	0.005	0.014

reliability scenarios plus a conventional scenario where the GPU works at nominal V_{dd} to avoid faults. Like the conventional GPU, all the DC-Patch design components are kept at nominal V_{dd} for the same reason. Note that a 64-bit single-bit error correction and double-bit error detection (SECDED) ECC with 8 check bits could be also used to protect the small DC-Patch memory arrays from dynamic variations or particle strikes with an overhead of 544 check bits [49].

We have assumed that the access time of the slice remains constant regardless of the V_{dd} value. However, reducing V_{dd} may lead to an increase of the transistors switching delay [3]. In this sense, we have measured the average performance loss of pipelining the slice block access and increasing the access time from 1 to 3 additional cycles. Similarly to [27], [50], the impact on performance ranges from 0.58 to 1.52%. These results indicate that enlarging the pipeline depth has a relatively low impact on performance, particularly if there is enough thread-level parallelism from simultaneous wavefronts allocated to the slice.

As expected, the dynamic energy and static power of the slice decrease with V_{dd}. The largest reductions can be seen in the dynamic energy since it grows quadratically with V_{dd}. Compared to the slice, the proposed components largely reduce energy, power, and area. Power is the parameter that impacts the most, since all the DC-Patch components consume 36.77 mW, which is almost two-thirds of the slice power when working at nominal V_{dd}. The reader is referred to Section V-B for a deeper analysis of the power consumption. On the other hand, the area estimation of all the proposed components is by 0.038 mm^2, which corresponds to a 5.6% of the slice.

Recall that the access time of the Com/Dec units (0.95 ns) forces to enlarge the pipeline in two additional stages. On the contrary, the access time is small enough in both the PT (0.09 ns) and the PSU (0.11 ns) to fit them into the original register translation and writeback stages. Furthermore, the impact of the additional 2:1 muxes in the critical path is minimal since the delay of this component is 13 ps according to Synopsis.

V. EXPERIMENTAL EVALUATION

DC-Patch has been modeled in the cycle-accurate Multi2Sim simulator [43]. Results include the execution time of the applications and additional processor statistics required to estimate energy. The overall energy consumption has been calculated combining processor statistics from Multi2Sim with energy numbers from CACTI and Synopsis. The configuration and memory hierarchy parameters of the modeled GPU are shown in Table 3. All the benchmarks run until completion.

TABLE 3. GPU configuration and memory hierarchy.

Clock frequency	1 GHz
CUs	10
Vector memory unit	32 entries, 1 per CU
Scalar unit	2 KB, 1 per CU, 1 cycle/instr.
Slice	64 KB, 4 per CU, 4-1-1-1 cycles/instr.
Max. WFs per CU	16
Work-items per WF	64
All caches	LRU, 64B-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
LDS caches	64 KB scratch, 1 per CU, 1 cycle
2× L2 caches	128 KB, 16-way per module, 10 cycles
Main Memory	2 channels per L2 module, 100 cycles

A. IMPACT ON PERFORMANCE

To better understand the main sources of performance degradation, we firstly classify slice writes into normal accesses, writes requiring a new patch operation, and writes triggering an access to the LDS cache due to spilled registers. Then, we quantify the LDS capacity devoted to spills, the additional load accesses to this cache, and the mispeculation of the compressor. Finally, the impact on the system performance is presented and analyzed.

1) SLICE WRITES BREAKDOWN

Figure 5 plots a breakdown of slice writes divided in normal accesses, patches to faulty or reliable entries, and writes to spilled registers in the LDS. The first write of a register is always considered as a patch to either a reliable or a faulty entry, or an access to the LDS.

Results show that, in applications like *QRandS* and *MatrixM*, with disparate compression capabilities (see Figure 2), the percentage of normal writes without a patch reaches 90%. This percentage is on average by 70%, meaning that most registers do not change the compression state of their contents during the entire lifetime. In other words, once DC-Patch selects a patch entry, it usually refers to the same logical register for the entire execution of a wavefront. The fact that results are quite homogeneous for a given application

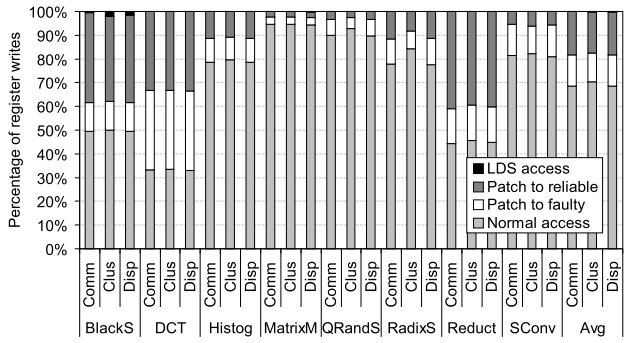

FIGURE 5. Percentage of register writes requiring a normal access without a patch, a patch operation to either a reliable or a faulty entry, and extra write accesses to the LDS.

under the different reliability scenarios also confirms these comments.

DC-Patch provides enough patching entries for the studied reliability scenarios, since the LDS writes are only noticeable in *BlackS*, where the percentage ranges between 1-2%. The combination of two key factors are the cause of the presence of LDS writes in this application. First, the percentage of compressed registers is quite low, which implies that a high number of faulty entries cannot be exploited for patching purposes. Second, the register file utilization is very high (see Figure 3), which limits the opportunities to find a patch entry.

2) IMPACT ON THE LDS CACHE

LDS cache writes due to unavailable register entries result in an LDS partition devoted to store spilled registers and additional load accesses to this partition when the processor requests such registers. Both the spill partition size and the percentage of additional loads increase with the number of faulty entries, and, hence, *dispersed* is the reliability scenario with the highest LDS requirements.

Only three benchmarks store spilled registers in the LDS. *BlackS*, which is the application with the least compression capabilities, requires the largest spill partition, 23 KB. Nevertheless, spilled registers are only responsible of a 1.2% of additional loads over the total number of the register file reads. More importantly, this does not imply any thrashing effects in the LDS since *BlackS* does not use this cache for regular data. *Histog* and *MatrixM*, which also present limited compression capabilities, require a spill partition size of 3 KB at most and show a low activity in the LDS with a 0.07% of additional loads due to spills at most. This is mainly due to the register file utilization of such applications is not as high as in *BlackS*. Like *BlackS*, *MatrixM* neither uses the LDS apart from managing spilled registers. On the contrary, *Histog* uses the LDS for regular data, but the percentage of additional loads is very low to cause noticeable thrashing effects.

3) COMPRESSION MISPECULATION

Compression mispeculation is another source of performance degradation, since an initially patched faulty entry has to be invalidated and the uncompressed register from the DRB has to be written to the inverse reliable patch indicated by the PSU, stalling the pipeline until the write operation completes.

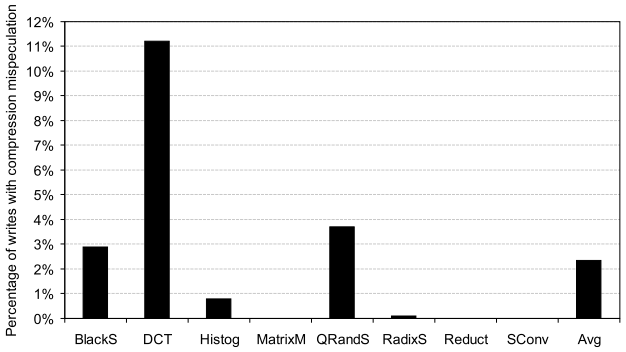

FIGURE 6. Percentage of register writes with a mispeculation on the compression phase.

Figure 6 shows the percentage of mispeculated register writes over the total number of writes. Results are independent of the reliability scenario. The prediction of the compression mechanism is quite accurate, since the percentage of mispeculated writes is on average by 2.4%. In fact, half of the benchmarks present a negligible percentage of write mispeculations or no mispeculations at all.

4) IMPACT ON THE SYSTEM PERFORMANCE

Figure 7 depicts the system performance degradation of the DC-Patch technique with respect to a conventional register file working at nominal supply voltage. Recall that, apart from the additional LDS load accesses and the mispeculation on compression operations, the increased pipeline depth with the added compression and decompression stages can also hurt performance.

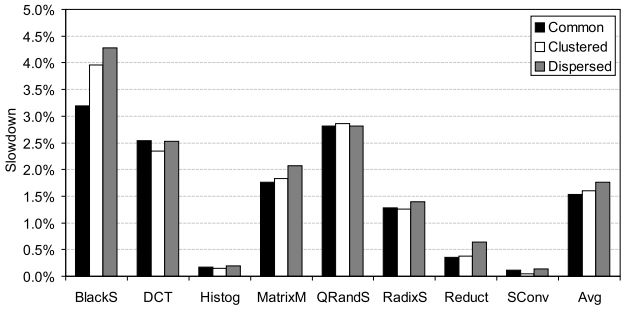

FIGURE 7. Slowdown of DC-Patch with respect to a conventional register file.

The performance impact largely varies across applications. As expected, *BlackS* shows the highest performance impact, since it suffers from the three main sources of performance losses. Nevertheless, the performance losses do not exceed 4.5% with respect to the conventional approach. For *DCT*, *Histog*, and *QRandS*, the performance impact is quite uniform regardless of the reliability scenarios. These losses are

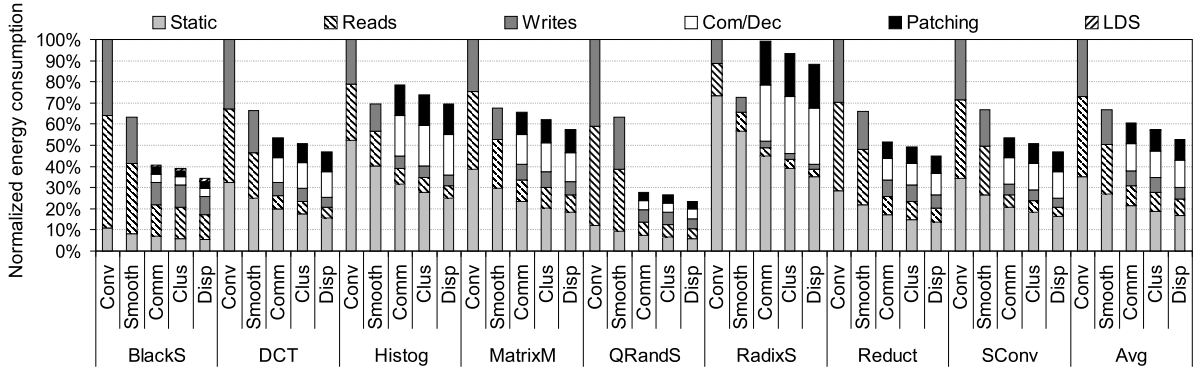

FIGURE 8. Normalized energy of DC-Patch and a voltage noise smoothing scheme with respect to a conventional register file design.

mainly attributed to the combined effects of the compression mispeculation and the increased pipeline depth. *MatrixM* suffers from both the additional pipeline stages and the number of additional LDS loads. The remaining benchmarks are mostly affected by the increased pipeline depth. Nevertheless, the results confirm that the scheduler is able to frequently dispatch independent instructions from different wavefronts, which largely masks the performance penalty of the enlarged pipeline. Notice too that, for an application with a low compression capability, a high register file utilization, and a high LDS requirement for regular data, an ultimate solution would be to raise V_{dd} to enlarge the effective register file storage capacity at the cost of limited energy savings but enhanced performance.

Overall, the impact on performance of the DC-Patch mechanism is on average by 1.53, 1.60, and 1.76% for *common*, *clustered*, and *dispersed* reliability scenarios, respectively, compared to the conventional register file.

B. ENERGY SAVINGS

Aggressively undervolting memory structures brings large energy savings at the cost of permanent faults. The DC-Patch mechanism effectively overcomes such faults. However, the required design components of this mechanism and the enlarged execution time of the applications incur in energy overheads.

Figure 8 plots the normalized energy consumption of the register file slice with respect to the conventional slice design. A voltage noise smoothing scheme (*Smooth*) operating at a safe 600 mV V_{dd} is included for comparison purposes. Energy is divided into static and dynamic expenses. In turn, dynamic energy is classified into expenses due to reads and write operations in the slice. The energy overhead of compression mispeculations is added to the *Write* category. Label *Com/Dec* refers to the static and dynamic expenses of the compression and decompression units, whereas static and dynamic consumption of the PT, PSU, and DRB are accumulated in the *Patching* category. Finally, the dynamic energy overhead of accessing the LDS due to spills is also quantified.

The contribution of static expenses over the total energy widely varies among applications. For example, applications with a relatively low register file usage show a smaller contribution of dynamic expenses but keep accumulating static energy on each cycle. This is the case of *RadixS*, where a high number of wavefronts are allocated during the application execution but their number of slice accesses are relatively low. Since the static consumption grows linearly with V_{dd}, *Smooth* and DC-Patch significantly reduce such expenses compared to the conventional scheme. DC-Patch achieves such energy savings in spite of the static energy overhead due to the enlarged execution time over *Conv* and *Smooth*. As expected, *dispersed*, with the lowest V_{dd}, obtains the highest static energy savings, followed by *common* and *clustered*.

The reduction of the supply voltage has a greater impact on the dynamic expenses, since V_{dd} has a quadratic effect on this type of consumption. In addition, DC-Patch also contributes to further reducing the dynamic expenses over *Conv* and *Smooth*. This is due to, on a slice read/write access to a compressed register, just a single block containing the compressed data is accessed instead of the whole register. Applications with a high register file usage and a high number of compressed registers, like *QRandS* and *SConv*, show large dynamic energy savings.

Recall that the DC-Patch components operate at nominal V_{dd} to avoid faults in them. This imposes an energy overhead, but it does not prevent the proposed mechanism to largely reduce the total energy consumption over *Conv* and *Smooth* thanks to: i) aggressively undervolting the slice and ii) mitigating slice dynamic expenses when accessing to compressed registers. Compared to *Smooth*, only *Histog* and *RadixS* show a higher consumption. This is mainly due to these applications have a relatively low slice usage and data compression opportunities that prevent significant energy savings from dynamic slice expenses.

Notice too that the expenses of accessing the LDS spill partition are only noticeable in *BlackS*, and they are much smaller than those expenses due to accessing the slice, confirming that the number of accesses to spilled registers is much lower than the number of accesses to registers patched in the slice.

Overall, the total energy savings of DC-Patch under *common*, *clustered*, and *dispersed* are on average by 39, 43, and 47%, respectively, over the conventional scheme. Compared to *Smooth*, these percentages are 10, 14, and 21%, respectively.

VI. RELATED WORK

This section describes recent work that has exploited data compression in GPU register files, proposed fault patching techniques for both GPU and CPU architectures, orthogonal to this work, and other related research leveraging either data redundancy or reallocation approaches.

A. DATA COMPRESSION IN GPU REGISTER FILES

Zhang *et al*. implement a register file with spin-transfer torque magnetic RAM technology [50]. With the aim to reduce the dynamic energy overhead and long latencies of write operations, authors exploit the BDI algorithm to compress registers.

Warped-Compression saves static energy in register files by leveraging BDI compression [27]. For those registers identified as compressible, this approach maintains the compressed data in the least significant bits of these registers, whereas the remaining bits do not contain useful data and the associated cells are power gated.

The EREER approach removes duplicated adjacent components of a register, preserving the non-duplicated components and additional tag bits with the required metadata to decompress the register in the least significant bits of the register entry [48]. The unused components of a compressed register are power gated to reduce energy consumption.

Power gating helps not only to reduce static energy consumption but also to mitigate aging effects, particularly the Negative Bias Temperature Instability (NBTI) phenomenon [12]. Unlike Warped-Compression and EREER, RC+RAR is a technique that switches off entire registers by storing the compressed data in an auxiliary NBTI-free memory array [44]. Combining power gating with a register address rotation approach, all the register file cells can be powered off and age uniformly over time.

Mittal *et al*. combines the BDI compression algorithm with radiation-hardened circuits to tolerate transient faults [32]. Similarly to Warped-Compression and EREER, this approach stores the compressed data in the least significant bits of registers. To protect these data against particle strikes, authors propose to harden such bits. However, hardening methods either depend on advanced and costly semiconductor manufacturing processes and materials, which do not allow the dense integration scale of standard and much cheaper commercial semiconductor processes, or incur in larger area by using standard cells that include redundant transistors, guard rings, or modify the geometry of the transistors [19].

B. PATCHING TECHNIQUES

GR-Guard is a patching technique for GPU register files that leverages reliable dead entries containing useless registers to store useful registers from faulty entries, avoiding the use of such defective entries [42]. A register entry is identified as dead during the time period from the last read to the next write operation [33]. Since this information is unknown at run-time, GR-Guard makes use of the compiler and modifies the instruction set to identify dead register entries at run-time. Specifically, the instruction format includes an additional bit for each operand to distinguish if the associated register entry is dead or not. Contrary to this work, we do not rely on the compiler neither modify the instruction set. Our patching technique is driven by the key observation that register contents are easily compressible at run-time, which allows to treat faulty entries as patching locations themselves, increasing the patching opportunities over prior work.

iPatch tolerates faults in CPU L1 caches by exploiting the replication of both instruction and data cache contents in other pipeline structures, like the trace cache, the MSHR, and the store queue [36]. The proposal deals with faulty L1 lines by storing in such pipeline structures the correct data. This technique requires non-trivial modifications to the processor components, including the propagation of the memory consistency management to the pipeline structures used as a backup, which complicates the design and verification of the processor. Moreover, the fault coverage is limited to the L1 caches due to the relatively small size of the backup structures.

C. OTHER WORK

Xiang *et al*. exploit the storage of data copies combined with parity protection to detect and correct faults in GPU register files [47]. Other approaches focus on the GPU execution lanes by exploiting either data reallocation techniques to avoid the use faulty lanes and to reroute threads to idle lines [16], or data redundancy in adjacent lanes and the presence of idle lanes to detect and correct faults [2].

Duwe *et al*. focus on the tolerance of faults in a CPU cache-like structure [15]. With the aim to improve the ECC correction capability, this approach reorders the bits within a cache line in such a way that ECC is able to generate a correction code. The search process of a suitable bit reordering is exhaustive, since each possible combination should be tried. In terms of execution latency, this can be unbearable inside a processor pipeline. However, such an operation latency is more likely to withstand inside a cache. Finally, Jadidi *et al*. bypass permanent faults in CPU caches implemented with phase-change memory technology by leveraging data compression and shifting the compressed data within the cache lines [20].

VII. CONCLUSION

In GPUs, supply voltage reductions can bring substantial energy savings, especially in SRAM memory arrays occupying a large percentage of the on-die area. However, lowering the supply voltage compromises the reliability of such arrays as it induces the appearance of permanent faults due to

process variations, which in turn reduces the effective storage capacity of the memory.

The main goal of this work has been to operate GPU register files below the safe supply voltage limit for energy efficiency and to ensure the reliability of these circuits at such operating conditions. To do so, we have proposed a patching technique, DC-Patch, exploiting the inherent data redundancy of GPGPU applications to compress register entries at runtime. Such a redundancy appears in many data access patterns that programmers often write. With compression enabled, faulty register file entries become useful again because they can store multiple compressed registers in their reliable cells, effectively enhancing the register file storage capacity.

Experimental results have shown that DC-Patch ensures a reliable operation of a register file with 39% of faulty register entries. At this fault rate, energy consumption reduces by 47 and 21% compared to a fault-free register file operating at nominal supply voltage and at the safe supply voltage limit, respectively, whereas the impact on performance and area remains below 2 and 6%.

REFERENCES

[1] M. Abdel-Majeed and M. Annavaram, "Warped register file: A power efficient register file for GPGPUs," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2013, pp. 412–423.

[2] M. Abdel-Majeed, W. Dweik, H. Jeon, and M. Annavaram, "Warped-RE: Low-cost error detection and correction in GPUs," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2015, pp. 331–342.

[3] M. Abdel-Majeed, A. Shafaei, H. Jeon, M. Pedram, and M. Annavaram, "Pilot register file: Energy efficient partitioned register file for GPUs," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 589–600.

[4] *AMD Graphics Cores Next (GCN) Architecture*, AMD, Sunnyvale, CA, USA, 2012.

[5] *OpenCL Optimization Guide*, AMD, Sunnyvale, CA, USA, 2013.

[6] *AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK)*, AMD, Sunnyvale, CA, USA, 2015.

[7] *AMD Graphics Core Next Architecture, Generation 3, Reference Guide*, AMD, Sunnyvale, CA, USA, 2016.

[8] A. Limited. (2008). *ARM11 MPCore Processor. Revision: R2P0. Technical Reference Manual*. [Online]. Available: https://static.docs.arm.com/ddi0360/f/DDI0360F_arm11_mpcore_r2p0_trm.pdf

[9] H. Asghari Esfeden, F. Khorasani, H. Jeon, D. Wong, and N. Abu-Ghazaleh, "CORF: Coalescing operand register file for GPUs," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 701–714.

[10] 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. Develop.*, vol. 50, no. 4.5, pp. 433–449, Jul. 2006.

[11] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *Proc. 40th Conf. Design Autom. (DAC)*, 2003, pp. 338–342.

[12] A. Calimera, E. Macii, and M. Poncino, "Analysis of NBTI-induced SNM degradation in power-gated SRAM cells," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 2010, pp. 785–788.

[13] B. J. Campbell, "Integrated circuit with separate supply voltage for memory that is different from logic circuit supply voltage," U.S. Patent. 8 848 463 B2, Aug. 25, 2014.

[14] A. Chatzidimitriou, G. Panadimitriou, D. Gizopoulos, S. Ganapathy, and J. Kalamatianos, "Assessing the effects of low voltage in branch prediction units," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2019, pp. 127–136.

[15] H. Duwe, X. Jian, D. Petrisko, and R. Kumar, "Rescuing uncorrectable fault patterns in on-chip memories through error pattern transformation," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 634–644.

[16] W. Dweik, M. A. Majeed, and M. Annavaram, "Warped-shield: Tolerating hard faults in GPGPUs," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2014, pp. 431–442.

[17] S. Ganapathy, J. Kalamatianos, K. Kasprak, and S. Raasch, "On characterizing near-threshold SRAM failures in FinFET technology," in *Proc. 54th Annu. Design Autom. Conf.*, Jun. 2017, pp. 1–6.

[18] *Intel FPGA SDK for OpenCL*. Best Practices Guide, Intel Corporation, Mountain View, CA, USA, 2017.

[19] X. Iturbe, B. Venu, E. Ozer, J.-L. Poupat, G. Gimenez, and H.-U. Zurek, "The arm triple core lock-step (TCLS) processor," *ACM Trans. Comput. Syst.*, vol. 36, no. 3, pp. 1–30, Aug. 2019.

[20] A. Jadidi, M. Arjomand, M. K. Tavana, D. R. Kaeli, M. T. Kandemir, and C. R. Das, "Exploring the potential for collaborative data compression and hard-error tolerance in PCM memories," in *Proc. 47th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2017, pp. 85–96.

[21] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "GPU register file virtualization," in *Proc. 48th Int. Symp. Microarchitecture (MICRO)*, 2015, pp. 420–432.

[22] W. Jeon, J. H. Park, Y. Kim, G. Koo, and W. W. Ro, "Hi-end: Hierarchical, endurance-aware STT-MRAM-Based register file for energy-efficient GPUs," *IEEE Access*, vol. 8, pp. 127768–127780, 2020.

[23] U. R. Karpuzcu, K. B. Kolluru, N. S. Kim, and J. Torrellas, "VARIUS-NTV: A microarchitectural model to capture the increased sensitivity of manycores to process variations at near-threshold voltages," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2012, pp. 1–11.

[24] S. K. Khatamifard, M. Resch, N. S. Kim, and U. R. Karpuzcu, "VARIUS-TC: A modular architecture-level model of parametric variation for thin-channel switches," in *Proc. IEEE 34th Int. Conf. Comput. Design (ICCD)*, Oct. 2016, pp. 654–661.

[25] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. Hoe, "Multi-bit error tolerant caches using two-dimensional error coding," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2007, pp. 197–209.

[26] J. Kloosterman, J. Beaumont, D. A. Jamshidi, J. Bailey, T. Mudge, and S. Mahlke, "Regless: Just-in-time operand staging for GPUs," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2017, pp. 151–164.

[27] 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.*, vol. 66, no. 5, pp. 834–847, May 2017.

[28] J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, and V. J. Reddi, "Safe limits on voltage reduction efficiency in GPUs: A direct measurement approach," in *Proc. 48th Int. Symp. Microarchitecture (MICRO)*, 2015, pp. 294–307.

[29] J. Leng, Y. Zu, and V. J. Reddi, "GPU voltage noise: Characterization and hierarchical smoothing of spatial and temporal voltage noise interference in GPU architectures," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2015, pp. 161–173.

[30] 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," HP Lab., Palo Alto, CA, USA, Tech. Rep. HPL-2012-187, 2012.

[31] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar. 2008.

[32] S. Mittal, H. Wang, A. Jog, and J. S. Vetter, "Design and analysis of soft-error resilience mechanisms for GPU register file," in *Proc. 30th Int. Conf. VLSI Design 16th Int. Conf. Embedded Syst. (VLSID)*, Jan. 2017, pp. 409–414.

[33] T. Monreal, V. Vinals, J. Gonzalez, A. Gonzalez, and M. Valero, "Late allocation and early release of physical registers," *IEEE Trans. Comput.*, vol. 53, no. 10, pp. 1244–1259, Oct. 2004.

[34] *NVIDIA OpenCL Best Practices Guide*, Nvidia, Santa Clara, CA, USA, 2009.

[35] Nvidia Inc. (2017). *Inside Volta*. [Online]. Available: https://devblogs.nvidia.com/inside-volta/

[36] D. J. Palframan, N. S. Kim, and M. H. Lipasti, "IPatch: Intelligent fault patching to improve energy efficiency," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2015, pp. 428–438.

[37] 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. Conf. Parallel Architectures Compilation Techn. (PACT)*, 2012, pp. 377–388.

[38] M. Sadrosadati, A. Mirhosseini, S. B. Ehsani, H. Sarbazi-Azad, M. Drumond, B. Falsafi, R. Ausavarungnirun, and O. Mutlu, "LTRF: Enabling high-capacity register files for GPUs via Hardware/Software cooperative register prefetching," in *Proc. 23RD Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2018, pp. 489–502.
[39] B. Salami, O. S. Unsal, and A. C. Kestelman, "Comprehensive evaluation of supply voltage underscaling in FPGA on-chip memories," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 724–736.
[40] S. Schechter, G. H. Loh, K. Straus, and D. Burger, "Use ECP, not ECC, for hard failures in resistive memories," in *Proc. 37th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2010, pp. 141–152.
[41] J. Tan and X. Fu, "Mitigating the susceptibility of GPGPUs register file to process variations," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2015, pp. 969–978.
[42] J. Tan, S. L. Song, K. Yan, X. Fu, A. Marquez, and D. Kerbyson, "Combating the reliability challenge of GPU register file at low supply voltage," in *Proc. Int. Conf. Parallel Architectures Compilation*, Sep. 2016, pp. 3–15.
[43] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A simulation framework for CPU-GPU computing," in *Proc. 21st Int. Conf. Parallel Architectures Compilation Techn.*, 2012, pp. 335–344.
[44] A. Valero, F. Candel, D. Suárez-Gracia, S. Petit, and J. Sahuquillo, "An aging-aware GPU register file design based on data redundancy," *IEEE Trans. Comput.*, vol. 68, no. 1, pp. 4–20, Jan. 2019.
[45] X. Wei, H. Yue, and J. Tan, "LAD-ECC: Energy-efficient ECC mechanism for GPGPUs register file," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2020, pp. 1127–1132.
[46] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, "Trading off cache capacity for reliability to enable low voltage operation," in *Proc. Int. Symp. Comput. Archit.*, Jun. 2008, pp. 203–214.
[47] 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. ACM Conf. Int. Conf. Supercomputing (ICS)*, 2013, pp. 433–442.
[48] A. Yazdanpanah, S. Sajadimanesh, and S. Safari, "EREER: Energy-aware register file and execution unit using exploiting redundancy in GPGPUs," *Microprocessors Microsyst.*, vol. 77, Sep. 2020, Art. no. 103176.
[49] D. H. Yoon and M. Erez, "Memory mapped ECC: Low-cost error protection for last level caches," in *Proc. 36th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2009, pp. 116–127.
[50] H. Zhang, X. Chen, N. Xiao, and F. Liu, "Architecting energy-efficient STT-RAM based register file on GPGPUs via delta compression," in *Proc. 53rd Annu. Design Autom. Conf. (DAC)*, 2016, pp. 1–6.
[51] B. Zimmer, S. O. Toh, H. Vo, Y. Lee, O. Thomas, K. Asanovic, and B. Nikolic, "SRAM assist techniques for operation in a wide voltage range in 28-nm CMOS," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 59, no. 12, pp. 853–857, Dec. 2012.
[52] A. Zou, J. Leng, X. He, Y. Zu, C. D. Gill, V. J. Reddi, and X. Zhang, "Voltage-stacked GPUs: A control theory driven cross-layer solution for practical voltage stacking in GPUs," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 390–402.

ALEJANDRO VALERO received the Ph.D. 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, U.K. Since 2016, he has been an Assistant Professor with the Department of Computer and Systems Engineering, Universidad de Zaragoza, Spain. His current research interests include GPU architectures, memory hierarchy design, energy efficiency, and fault tolerance. He is a member of the Aragon Institute of Engineering Research (I3A) and the HiPEAC European NoE.

DARÍO SUÁREZ-GRACIA (Member, IEEE) received the Ph.D. degree in computer engineering from the Universidad de Zaragoza, Spain, in 2011. From 2012 to 2015, he was with Qualcomm Research Silicon Valley. He is currently an Associate Professor with the Universidad de Zaragoza. His research interests include parallel programming, heterogeneous computing, memory hierarchy design, energy-efficient multiprocessors, and fault-tolerance. He is a member of the Aragon Institute of Engineering Research (I3A), the IEEE Computer Society, ACM, and the HiPEAC European NoE.

RUBÉN GRAN-TEJERO graduated in computer science from the University of Zaragoza, Spain. He received the Ph.D. degree from the Polytechnic University of Catalonia (UPC), Spain, in 2010. Since 2010, he has been an Associate Professor with the Department of Computer Science and Systems Engineering, University of Zaragoza. His research interests include hard real-time systems, hardware for reducing worst-case execution time and energy consumption, efficient processor microarchitecture, and effective programming for parallel and heterogeneous systems.