Compiler-Directed and Architecture-Independent Mitigation of Read Disturbance Errors in STT-RAM

Fateme S. Hosseini
Department of Electrical and Computer Engineering
University of Delaware
Newark, Delaware, USA
fateme@udel.edu

Chengmo Yang
Department of Electrical and Computer Engineering
University of Delaware
Newark, Delaware, USA
chengmo@udel.edu

***Abstract*—High density, negligible leakage power, and fast read speed have made Spin-Transfer Torque Random Access Memory (STT-RAM) one of the most promising candidates for next generation on-chip memories. However, STT-RAM suffers from read-disturbance errors, that is, read operations might accidentally change the value of the accessed memory location. Although these errors could be mitigated by applying a restore-after-read operation, the energy overhead would be significant. This paper presents an architecture-independent framework to mitigate read disturbance errors while reducing the energy overhead, by selectively inserting restore operations under the guidance of the compiler. For that purpose, the vulnerability of load operations to read disturbance errors is evaluated using a specifically designed fault model; a code transformation technique is developed to reduce the number of vulnerable loads; and, an algorithm is proposed to selectively insert restore operations. The evaluation results show that the proposed technique can effectively reduce up to 97% of restore operations and 66% of the energy overhead while maintaining 99.8% coverage of read disturbance errors.**

I. Introduction

The ever-growing need for higher performance and lower power in modern computer systems demands large and scalable on-chip memories. However, SRAM, as the dominant technology for on-chip memories, cannot keep up with the pace due to its low density and high leakage power consumption. DRAM, as a viable alternative, offers higher density but suffers from even higher power consumption. Targeting the power consumption issue, several alternative nonvolatile memory technologies have been proposed. Among them, STT-RAM [6] is a promising candidate for on-chip memory, as it offers near-zero leakage power, high scalability, high cell density, low access latency, and high endurance [2], [6]. However, this technology has the downsides of high write energy and *read-disturbance-error*s.

STT-RAM's read and write operations are performed via injecting current into its cells. As the technology scales, the write current is reduced quickly but the read current remains almost the same. For sub-$32nm$ feature size, the amplitudes of read and write currents get so close that it is possible for a read operation to accidentally change the data stored in an STT-RAM cell [17], [19]. Read disturbance errors, once created, persist in the cell and affect all the subsequent reads until the cell is overwritten. Errors in multiple consecutive read operations can even accumulate up to a point that exceeds the capability of error correction codes.

This work is partially supported by NSF grant #1253733.

The basic solution to read disturbance errors is *restore-after-read* which writes the read data back to the STT-RAM cell after each read operation [15]. The problem with this solution is the large number of restore operations which induces high energy overhead and degrades performance. While some previous works have been proposed [7], [17], [19] to reduce such overhead, they usually target a specific component in the memory hierarchy, either a certain level of cache [17] or GPU register sets [19], and may require major modifications to the system architecture or the read/write circuits [7].

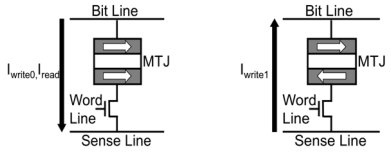
This paper proposes an architecture-independent and energy-efficient framework for mitigating read-disturbance errors, and is applicable to multiple types of STT-RAM memories. The framework utilizes a set of compile-time analysis and transformations to selectively place restore operations in the program. This is a pure application-level technique which is orthogonal to most of the architectural-level works, and thus can be combined with them if needed. The proposed framework is developed based on the observation that load instructions, even those accessing the same memory location, are not equally vulnerable to read disturbance errors. Such vulnerability is measured based on the application's data flow and its fine-grained behavior. In a nutshell, the major components of the framework are:

- Load vulnerability assessment, performed under a worst-case scenario fault model and an extended version of the program's data flow.
- Load instructions classification based on two criteria: whether they are vulnerable, and whether they need to be precise.
- An algorithm to selectively insert necessary restore operations after precise loads.
- Further reduction of restore overhead by exploiting free registers to hold vulnerable load values.

The rest of this paper is organized as follows: Section II briefly reviews STT-RAM design as well as the previous work. Section III describes the technical details of the proposed framework. Section IV presents the evaluation framework and experimental results, while Section V concludes the paper.

II. Background and Related Work

An STT-RAM cell consists of a Magnetic Tunnel Junction (MTJ) and an access transistor, shown in Fig. 1. A typical



(a) (b)

Fig. 1. STT-RAM cell storing a) logical '0' and b) logical '1'

MTJ structure is composed of two ferromagnetic layers (called reference and free layers) and one tunnel barrier layer. The magnetic direction of the reference layer is fixed, while the direction of the free layer represents the binary value stored in the cell, as shown in Fig. 1(a) and 1(b).

Write operations are performed by applying a large current between the sense line and bit line. The current direction determines the direction of the free layer and hence the binary value stored in the cell, see Fig. 1. Reading an STT-RAM cell is done by applying a small voltage between the bit line and sense line to sense the current flow. As the feature size shrinks, the write current is reduced and the gap between read and write currents becomes smaller. For sub-$32nm$ technologies, a read current can potentially flip the free layer direction and change the stored data, known as *read disturbance error*. At $32nm$, the line error rate of the 64B STT-RAM cache studied in [17] and the 1024-bit STT-RAM GPU register set studied in [19], after adopting error correcting codes, are respectively $1.6E-4$ and $6.12E-8$ which are higher than $2.5E-11$, the acceptable memory error rate of DRAM [13]. Both studies show that the error rates keep increasing in smaller feature sizes. As Fig. 1(a) shows, read current is in the same direction as writing '0' which makes read operations immune to unwanted reversal magnetization [8]. As a result, read disturbance errors are unidirectional $1 \rightarrow 0$ bit flips.

Some of the previous works address read disturbance errors at the circuit-level. [7] proposes a new sensing circuit to mitigate read disturbance errors by performing non-disruptive low current long latency reads. [16] proposes to use disruptive large current to speed up read operations and restore the data afterward. While effective, both works considerably degrade system performance.

A dual-mode fast-switching STT-RAM is presented in [15]. It switches between two read modes: *restore-after-read* in the reliable mode and *no-restore* in the low power mode. However, the large number of restores required in the reliable mode significantly reduces system performance and energy efficiency. To address this problem, multiple methods are proposed to perform restore operations selectively. [17] postpones restores in L2 STT-RAM caches until the cache-line is evicted from the L1 cache. Disturbed cells are selectively restored based on the status of the evicted line. [3] proposes a similar approach for multi-level cell (MLC) STT-RAM caches. [14] leverages error correcting codes to skip restore operations in STT-RAM memories. [3] decreases restore overhead of MLC STT-RAM caches by intentionally overwriting soft bit lines which are less likely to be read. [11] reduces the restore traffic in the last level STT-RAM caches by duplicating some data and using data compression techniques. [19] combines application-level and architecture-level techniques to reduce restore operations in STT-RAM GPU registers, based on the fact that most registers are usually referenced only once and after that are dead[1].

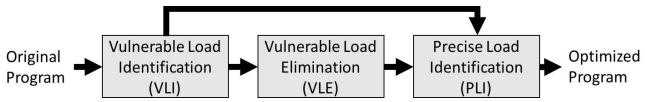


Fig. 2. Framework overview

The aforementioned works mostly target a specific component of the memory hierarchy. They typically require extra hardware in the design and sometimes data redundancy in the application. Their effectiveness heavily depends on the possibility of applying the desired modification to the system architecture or the underlying circuits. In comparison, this work proposes an application-level framework that does not require any extra hardware, data redundancy, or modification to the system at either architecture-level or circuit-level.

III. Proposed Framework

A. Overview

This proposed framework is developed based on the observation that loads are not equally vulnerable to read disturbance errors. The difference in the intrinsic vulnerability to faults among the application's data structures has been the basis for many approximation-based techniques [9], [12] which classify data structures and variables in high-level representations. This work moves one step further towards a finer-grained study of resilience, that is, even different loads following the same store are not equally vulnerable to read disturbance errors since they belong to different computational data flows. This motivates us to perform load vulnerability analysis and classification.

Fig. 2 shows an overview of the proposed framework composed of three parts. *Vulnerable load identification (VLI)* performs a fine-grained analysis on the vulnerability of memory locations which need protection against read disturbance errors, and classifies them as either *vulnerable* or *invulnerable* accordingly. Definitions of different load classes used throughout this paper are summarized in Table I.

After obtaining load vulnerability information, the *precise load identification (PLI)* step identifies *precise* and *imprecise* loads. A *precise* load requires a restore afterward to ensure that its operation is performed precisely, i.e., without inducing any error in STT-RAM. A load is marked *precise* if any of the succeeding loads to the same memory location are *vulnerable*. On the contrary, a load operation is *imprecise* and requires no restore if none of its succeeding loads are *vulnerable*.

In some applications, specifically the ones with lower inherent resilience, a high fraction of loads might be *vulnerable*, which in turn would result in most of the loads being classified as *precise* and requiring restore operations. To address this issue, the *vulnerable load elimination (VLE)* step transforms the original code to reduce the number of *vulnerable* loads before *PLI*.

One advantage of the proposed framework is that it does not rely on the availability of multiple load types or special restore

[1]At any program point, a register or memory location is dead if its subsequent access is a write.

TABLE I
LOAD CLASSES DEFINITIONS

Class	Definition
Vulnerable	Reading a disturbed value → unexpected program output
Invulnerable	Reading a disturbed value → expected program output
Precise	If disruptive → program output is likely to be unexpected
Imprecise	If disruptive → program output is likely to be expected

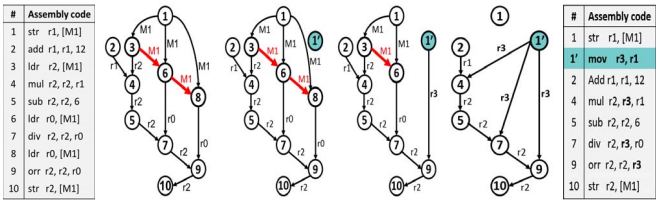


Fig. 3. (a) Original code, (b) Extended DDG, (c)(d)(e) *VLE* transformation steps, and (f) *VLE* transformed code

circuits. Loads classified by *PLI* can be encoded with two different opcodes if such an architectural platform is available at runtime. Otherwise, the framework is able to simulate the behavior of restore operations by inserting explicit store operations after *precise* loads in the code.

B. Vulnerable loads identification (VLI)

This step classifies program loads based on the analysis of their vulnerability to read disturbance errors. A *vulnerable* load satisfies two conditions: (I) its loaded value can be affected by a disturbance caused by a preceding load, and (II) a disturbed value loaded by this instruction affects the correctness of application's final outcome. If either condition is not met, the load is classified as *invulnerable*.

The first condition can be explained via a careful study of the program data-dependence-graph (DDG)[2]. A fundamental observation of this work is that read disturbance errors change the program's original data flow. Take the code snippet in Fig. 3(a) as an example. Its original data flow is presented as black edges in the DDG shown in Fig. 3(b). Each edge represents a data dependency between instructions, and is labeled with the register-operand (e.g., $r0$) or the memory-location (e.g., $M1$) creating that dependency. In the original DDG, loads are always consumers of the memory values and producers of register operands. However, in the presence of read disturbance errors, load instructions are able to produce memory values. For a load such as 6 in this example, its value producer is not the preceding store (instruction 1) but the preceding load (instruction 3). Red edges in Fig. 3(b) represent the extra dependencies under read-disturbance errors. The chain formed by these edges represents how one disturbed value is propagated to the subsequent loads and will be referred to as a **disturbance chain**.

The extended DDG gives a clear and useful structure for studying read disturbance errors and restore insertion. For example, in Fig. 3(b) the disturbance chain begins with load 3 and ends at 8. A value disturbed by instruction 8 does not affect the correctness of subsequent loads because after instruction 8 the value stored at $M1$ is dead and no restore is needed. Similarly, the first condition of *vulnerable* load can be translated to the load being reached by others through a disturbance chain. Among the three loads 3, 6, and 8 in this example, 3 can be immediately classified as *invulnerable* because it is the first node on the disturbance chain, and thus its loaded value cannot be disturbed by any preceding loads.

The second condition looks for any divergence in the program's behavior when faults are injected into a loaded value. Evaluating the program's behavior for all the loads under all the possible fault scenarios would be very time-consuming as fault injection needs to be performed independently for each load instruction. To obtain the vulnerability information precisely, for evaluating N load instructions, N faulty runs need to be performed. To mitigate such high overhead, this work performs two optimizations: (1) only the loads which satisfy the first condition are injected with a worst-case disturbance scenario; Since read disturbance errors cause unidirectional $1 \rightarrow 0$ bit-flips, the worst-case fault is designed to flip the leftmost '1' to '0', (2) multiple disturbed bits are simulated via injecting one worst-case fault once in the same instruction repeatedly every N iterations.

In the end, the results of faulty runs are compared with the expected results under the equality constraint of 100%[3] and load instructions are classified accordingly.

C. Vulnerable load elimination (VLE)

VLE aims to reduce the number of *vulnerable* loads via code transformations because to protect one *vulnerable* load from disturbance, all of its predecessors on the disturbance chain should be *precise*. Fig. 4(a) shows a disturbance chain where only the last load 5 is *vulnerable*. Although loads $1-4$ are *invulnerable*, they all should be paired with a restore operation to preclude any disturbance in 5.

To eliminate a *vulnerable* load from a disturbance chain, its memory access should be removed and the data stored in the memory location should be extracted using an alternative method. One possible approach is to pass this data among instructions using register-based operations. Replacing loads with *mov* instructions (i.e., load $ld\ V,[M]$ with the value producer $str\ r,[M]$ is replaced with $mov\ V,r$) seems to be a viable option. However, it does not guarantee the data flow correctness of the data flow because register r might be overwritten before the load is executed. Fig. 3(a) illustrates an example, where none of the load instructions can be directly replaced by such a *mov* instruction because instruction 2 overwrites store 1's value in $r1$. *VLE* tackles this challenge via utilizing free registers. Note that although in a global view of the program all the registers might be in use, it is still possible to find free registers in small regions of the code, such as loops. *VLE* inserts a *mov* instruction ($mov\ freeReg, r$) right after the load's value producer to transfer the stored value to the free register, such as instruction $1'$ in Fig. 3. Then, the subsequent loads can be eliminated and their values are provided by that free register to their dependent instructions.

[2]DDG is a graph in which nodes are instructions and an edge (u, v) represents a register/memory value produced (written) by u and consumed (read) by v.

[3]100% equality means the output of the faulty execution should exactly match the program's expected output. The equality constraint can be modified to relax the resilience constraint of the program.

(a) (b)

Fig. 4. Impact of *vulnerable* load location on the number of restores needed

Although removing all *vulnerable* loads is ideal, finding enough free registers for transforming all the vulnerable loads is unlikely. *VLE* addresses this issue by employing a heuristic to selectively eliminate as many of *vulnerable* loads as possible inside program's loops. Loops are targeted because their code sizes are small but the run-time benefit of reducing restores in them is significant. Eliminating loads with higher frequency the in loops is more likely to yield more reduction in restores, however, an efficient heuristic cannot only depend on the loads' frequencies. Fig. 4 illustrates the reason with an example. Assuming the two disturbance chains in the figure are inside the same loop with load frequencies of 800 and 1000 respectively, a frequency-based heuristic selects load 2 in Fig. 4(b) for elimination. However, eliminating 5 from Fig. 4(a) results in reducing four restore operations per loop iteration, and the overall reduction $(4 * 800)$ is higher than eliminating 2 in Fig. 4(b) $(1 * 1000)$. The example clearly shows that the location of a *vulnerable* load in the disturbance chain is an important factor in addition to the load's frequency. Therefore, *VLE*'s heuristic prioritizes loads based on their $weight$, defined as a product of their execution frequency ($freq$) and their total number of their predecessors (N_{pred}) on the disturbance chain. *VLE* also identifies a set of free registers ($freeSet$) in each loop. A register is considered free if and only if it is dead at the loop's entry and is not written inside it.

Algorithm 1 shows the *VLE* process performed for each loop L (L is the outermost loop when nested loops exist). In each iteration of the algorithm, the highest-weighted *vulnerable* load is eliminated using a free register (lines 1-8). A mov instruction is inserted right after each of the load's value producers located on different program paths, to move their values to the same free register (lines 3-4). The input operands of the load's consumers are renamed to the free register, i.e., they are fed by the inserted mov instruction, and then the load is eliminated (lines 5-6). Once a *vulnerable* load is transformed, other loads on the same disturbance chain, if all of their value producers are included in the transformed load's value producer set, can also be removed similarly (line 8). This way, *VLE* is able to eliminate the maximum number of loads and restores with one free register. The algorithm terminates when no free register or no *vulnerable* load instruction remains (line 9).

VLE's transformation maintains the program's original data flow consistency by: 1) using the same free register to transform all the load's value producers located on different program paths, 2) renaming the output of other instructions which provide the same register as this load's value to its consumers but on other paths, if any, to the same free register (line 7), 3) performing code transformation for a *vulnerable* load only if all the dependencies created by the free register in the DDG begin and end inside the loop boundary (lines 2), since the free register is only guaranteed to be free inside the loop body.

Algorithm 1 *VLE* Algorithm

Require:
loop L, load's $weight$, $freeSet$
1: Get the highest weighted vulnerable $load$ and find its value producer on each path;
2: If any of the $load$'s value producers, or any of its consumers, or any of its consumers' producers with the same output register are outside L, return to 1;
3: Get a $freeReg$ from $freeSet$;
4: Add $(mov\ freeReg, producer - val)$ after its value producer on each path;
5: Rename $load$'s consumer's input to $freeReg$;
6: Remove $load$;
7: Rename the output of $load$'s consumer's other producers with the same output to $freeReg$;
8: Repeat (4-7) for other loads on the disturbance chain if and only if the value of all of their value producers are already moved to the $freeReg$;
9: If any *vulnerable* load and free register remain, return to 1;

Algorithm 2 *PLI* Algorithm

Require:
$reverse - topological - ordered - list$
1: **for all** $load$ in the $reverse - topological - ordered - list$ **do**
2: **if** $load$ is $vulnerable$ **then**
3: mark all $load$'s predecessors as $precise$;
4: **end if**
5: **end for**
6: mark all the unclassified loads as $imprecise$;

For the example shown in Fig. 3(a), assume instructions $1-10$ are part of the loop body, $r3$ is a free register, and 6 and 8 are both *vulnerable* and have the same execution frequencies. *VLE* selects 8 as the highest-weighted load since it has two predecessors on the disturbance chain, while 6 has only one. Instructions 1 and 9 are identified respectively as the value producer and the value consumer of 8. Instruction 1 and all 8's producers of $r0$ are located inside the loop and the dependencies created by their transformation will not go beyond loop boundary. So, a mov instruction ($1'$) is inserted after 1 to move its value in $r1$ to $r3$, see Fig. 3(c). Then, $r3$ is fed to 9 and load 8 is removed, see Fig. 3(d). At this point, no further renaming is necessary since 8 does not have any other producers for $r0$. Finally, since 1 is the only value producer of loads 3 and 6 and its value has already been moved to $r3$, both 3 and 6 and their dependence chains are transformed as well, see Fig. 3(e). The transformed code in Fig. 3(f) confirms that *VLE* not only breaks the disturbance chains but also reduces the number of memory accesses caused by load instructions and can potentially increase performance.

D. Precise Load Identification (PLI)

PLI identifies whether a load needs to be paired with a restore afterward, based on the vulnerability of loads inside the transformed code.

The PLI algorithm, shown in Algorithm 2, processes loads in the reverse topological order of their disturbance chain starting from the bottom node(s) to protect a *vulnerable* load against all its preceding loads on the disturbance chain. If a processed load is *vulnerable*, all of its preceding loads are marked as *precise* (lines 2-3). The algorithm takes a conservative strategy to enhance the program reliability. Once a load is marked *precise* it will remain the same even if it can be classified *imprecise* via other paths on the disturbance chain. PLI's process continues until all the *vulnerable* loads are processed. In the end, all the unclassified loads are marked as *imprecise*.

IV. Evaluation

This section examines the proposed technique's impact on the number of restore operations, energy consumption, performance, and reliability of STT-RAM against read disturbance errors.

A. Evaluation Setup

All compile-time analysis and code transformations including *VLE* and *PLI* passes are implemented in LLVM [10] for ARM architecture. Runtime simulations are done in Gem5 [1] under *se* mode using *TimingSimpleCPU* model for the ARM architecture. Evaluations are performed for a 16MB STT-RAM cache by adopting the corresponding parameters from [20].

An in-house Gem5-based fault injection framework is developed to simulate read disturbance errors for 1) evaluating loads' vulnerability in *VLI* and 2) reliability assessments. The former utilizes the fault model presented in Section III-B and flips the leftmost '1' of a load value to '0'. The fault is repeatedly injected in the same load instruction every 100 iterations. The latter simulates read disturbance as a random $1 \rightarrow 0$ bit flip in a memory cell and propagates the disturbed value through the disturbance chain. The proposed technique is compared with *restore-after-read* [15] as the comparison **baseline** and *Red-shield* [19] which is originally developed for GPU registers and eliminates restores after the last reads of a register variables. This paper alters this technique to target memory operations by removing restore operations for loads after which the memory values are known to be dead at compile-time.

The proposed technique is evaluated for a diverse set of benchmarks, representing different program structures and resilience. Benchmarks are selected from three suites: Mibench [5] which includes widely used kernels for embedded systems, Axbench [18] for approximate computing applications, and MediabenchII [4] for multimedia applications. All benchmarks are cross-compiled for ARM.

B. Results

***Vulnerable* and *precise* loads**: Fig. 5 shows the ratio of *vulnerable* and *precise* loads classified respectively by the *VLI* and *PLI* in the original and transformed codes, as well as the ratio of loads eliminated by *VLE*. As shown, Mibench benchmarks show higher percentages of *vulnerable* and *precise* loads in general because many of them perform precise mathematical operations, and thus are less resilient to faults. On the other hand, MediabenchII and Axbench benchmarks perform image processing, machine learning, or video processing and are inherently more resilient. *Jmient* and *h.263*, respectively an image processing benchmark from Axbench and a video processing benchmark from MediaBenchII, have the lowest ratio of *vulnerable* loads (21%), while almost all loads in *basicmath* and *qsort* from Mibench are *vulnerable*. On average, 62% and 72% of loads in the benchmarks' original codes are classified as *vulnerable* and *precise*. According to the results, *VLE* successfully reduces *vulnerable* and *precise* loads, respectively by 11% and 12% on average, through the elimination of 12% of *vulnerable* loads only from the program loops.

Reduction in restore operations: Fig. 6 compares the ratio of restore operations reduced at run-time by the proposed framework

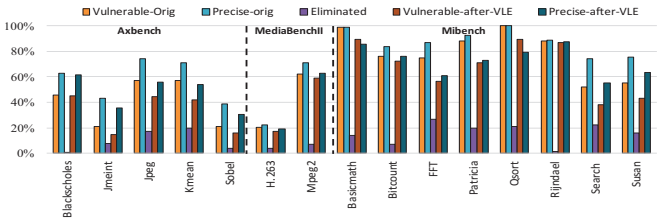


Fig. 5. Static load classification

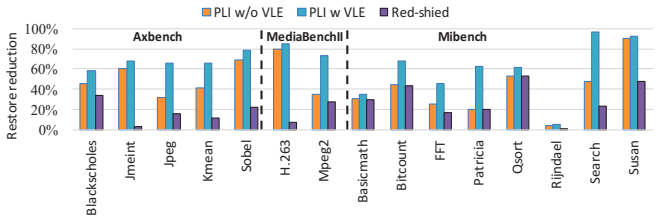


Fig. 6. Run-time restore reduction

and *Red-shield* with the baseline. As shown, *PLI* reduces 45% of restores on average by applying the proposed precise load classification algorithm on the original code. By eliminating 12% of static loads inside the loops, *VLE* is able to deliver an additional 19% reduction at run-time which results in a total reduction of 64%.

The run-time restore reduction results are in-line with static load classification results. Axbench and MediabenchII benchmarks show a consistently higher reduction in restores compared to most of the Mibench benchmarks, and *search* and *susan* have relatively higher reduction ratio in the Mibench suite. Having most of its *vulnerable* loads transformed in the main loops, *Search* shows the highest reduction of 97%. Comparison with Fig. 5 also reveals that although the ratio of static loads eliminated is not quite large, in many cases targeting the longest disturbance chain in loops leads to a significant reduction in restores at run-time, *mpeg2* is an example of this case. As shown, the total reduction achieved by the proposed framework is on average 24% higher than *Red-shield*. In general, the more non-dead loads transformed or marked as *imprecise*, the more the proposed technique outperforms *Red-shield*. *Search* is an example of this case, while *Rijndael* is a counter-example.

Energy and Performance efficiency: In this section, the energy overhead and performance of the proposed technique and *Red-shield* are calculated and compared with the baseline. Fig. 7 shows the energy overhead reduction of STT-RAM accesses. The energy consumption of read and write operations are respectively set to $0.205nJ$ and $1.620nJ$ according to [20]. As shown, the proposed technique reduces the baseline energy overhead by 38% on average which is 25% higher than *Red-shield*. The amount of energy reduction is determined by the fraction of total memory accesses reduced due to removing restore operations and load instructions. For example, *patricia* and *qsort* have more-or-less the same restore reduction ratios but the energy saving of *patricia* is higher, because the eliminated restores form a bigger fraction of total memory accesses.

Fig. 8 compares the performance of the proposed technique and *Red-Shield* to the baseline in terms of the number of clock cycles.

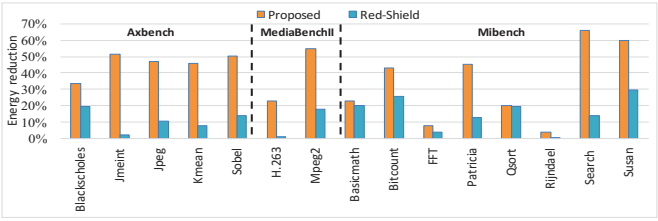


Fig. 7. Energy reduction of STT-RAM accesses

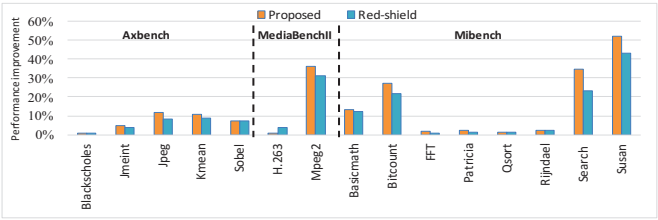


Fig. 8. Performance improvement

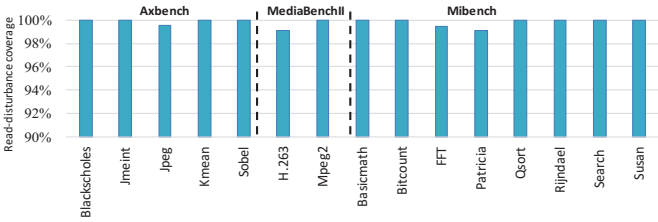


Fig. 9. Read disturbance fault coverage

The proposed scheme achieves an average of 14% improvement in performance compared to the baseline by reducing the restore operations and applying code transformation. This number is 3% better than *Red-shield*. Note that the performance improvement achieved by the framework is proportional not to the ratio of loads and restores reduced but to the ratio of total number of instructions reduced. Take *sobel* as an example. Its restore reduction ratio is 6% higher than *mpeg2* but its performance improvement is 11% lower because in *sobel* the proposed technique reduces a smaller fraction of the total number of instructions.

Reliability: In this part, the reliability of the proposed technique against read disturbance errors is evaluated. The baseline technique covers 100% of read disturbance errors, by immediately writing the loaded values back to the memory. The proposed technique, however, eliminates some of these potential recovery actions and might lose some fault coverage as a result.

In order to evaluate the fault coverage of the proposed technique, all program loads which are left unprotected, i.e. their restores are eliminated, are injected with faults. To mimic the behavior of read disturbance errors, for each unprotected load, a random '1' bit of the load's accessed memory location is flipped to '0' after the load is executed. This disturbed value persists in the memory and propagates through all the subsequent loads until being overwritten. The outcome of these experiments is compared with the expected results under equality threshold of 100%. Fig. 9 depicts the fault coverage of the proposed technique. As shown, for 11 out of 15 benchmarks, 100% of read disturbance errors are covered. The reliability of evaluated benchmarks is 99.8% on average which is a trivial reduction (0.2%) in the fault coverage.

V. Conclusions

This paper proposes an application-level framework which aims to reduce the number of restore operations required for mitigating STT-RAM's read disturbance errors. It performs compile-time analysis and transformations to identify loads after which restores can be eliminated, based on the program's extended data flow graph and a set of fine-grained behavior analysis. The experiment results show that the framework successfully reduces the number of restore operations, improves the energy efficiency and performance of the application while imposes a trivial reduction on the fault coverage.

References

[1] N. Binkert et al. The Gem5 Simulator, 1–7. *SIGARCH Comput. Archit. News*, 39(2), 2011.

[2] M. T. Chang, P. Rosenfeld, S. L. Lu, and B. Jacob. Technology comparison for large last-level caches (L3Cs): Low-leakage SRAM, low write-energy STT-RAM, and refresh-optimized eDRAM, 143–154. In *HPCA*, 2013.

[3] X. Chen, N. Khoshavi, R. F. DeMara, J. Wang, D. Huang, W. Wen, and Y. Chen. Energy-Aware Adaptive Restore Schemes for MLC STT-RAM Cache, 786–798. *IEEE Trans. Comput.*, 66(5), 2017.

[4] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf. MediaBench II Video: Expediting the Next Generation of Video Systems Research, 301–318. *Microprocess. Microsyst.*, 33(4), 2009.

[5] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite,, 3–14. In *WCC-4*, 2001.

[6] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram, 459–462. In *IEDM*, 2005.

[7] W. Kang, W. Zhao, J. O. Klein, Y. Zhang, C. Chappert, and D. Ravelosona. High reliability sensing circuit for deep submicron spin transfer torque magnetic random access memory, 1283–1285. *Electronics Letters*, 49(20), 2013.

[8] T. Kawahara et al. 2Mb Spin-Transfer Torque RAM (SPRAM) with Bit-by-Bit Bidirectional Current Write and Parallelizing-Direction Current Read, 480–617. In *ISSCC*, 2007.

[9] D. S. Khudia and S. Mahlke. Harnessing Soft Computations for Low-Budget Fault Tolerance, 319–330. In *MICRO*, 2014.

[10] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation, 75–86. In *CGO*, 2004.

[11] S. Mittal, J. S. Vetter, and L. Jiang. Addressing Read-Disturbance Issue in STT-RAM by Data Compression and Selective Duplication, 94–98. *IEEE Comput. Archit. Letters*, 16(2), 2017.

[12] A. Ranjan, S. Venkataramani, Z. Pajouhi, R. Venkatesan, K. Roy, and A. Raghunathan. STAxCache: An approximate, energy efficient STT-MRAM cache , 356–361. In *DATE*, 2017.

[13] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM Errors in the Wild: A Large-scale Field Study, 193–204. *SIGMETRICS Perform. Eval. Rev.*, 37(1), 2009.

[14] S. M. Seyedzadeh, R. Maddah, A. Jones, and R. Melhem. Leveraging ECC to Mitigate Read Disturbance, False Reads and Write Faults in STT-RAM, 215–226. In *DSN*, 2016.

[15] Z. Sun, H. Li, and W. Wu. A Dual-mode Architecture for Fast-switching STT-RAM, 45–50. In *ISLPED*, 2012.

[16] R. Takemura, T. Kawahara, K. Ono, K. Miura, H. Matsuoka, and H. Ohno. Highly-scalable disruptive reading scheme for Gb-scale SPRAM and beyond, 1–2. In *IEEE Intl. Memory Workshop*, 2010.

[17] R. Wang, L. Jiang, Y. Zhang, L. Wang, and J. Yang. Selective restore: An energy efficient read disturbance mitigation scheme for future STT-MRAM, 1–6. In *DAC*, 2015.

[18] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. AxBench: A Multiplatform Benchmark Suite for Approximate Computing, 60–68. *IEEE Design Test*, 34(2), 2017.

[19] H. Zhang, X. Chen, N. Xiao, F. Liu, and Z. Chen. Red-shield: Shielding read disturbance for STT-RAM based register files on GPUs, 389–392. In *GLSVLSI*, 2016.

[20] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. Energy reduction for STT-RAM using early write termination, 264–268. In *ICCAD*, 2009.