LARS: Logically Adaptable Retention Time STT-RAM Cache for Embedded Systems

Kyle Kuan and Tosiron Adegbija
Department of Electrical & Computer Engineering
University of Arizona, Tucson, AZ, USA
Email: {ckkuan, tosiron}@email.arizona.edu

***Abstract*—STT-RAMs have been studied as a promising alternative to SRAMs in embedded systems' caches and main memories. STT-RAMs are attractive due to their low leakage power and high density; STT-RAMs, however, also have drawbacks of long write latency and high dynamic write energy. A popular solution to this drawback relaxes the retention time to lower both write latency and energy, and uses a *dynamic refresh scheme* that refreshes data blocks to prevent them from prematurely expiring. However, the refreshes can incur overheads, thus limiting optimization potential. In addition, this solution only provides a single retention time, and cannot adapt to applications' variable retention time requirements. In this paper, we propose *LARS (Logically Adaptable Retention Time STT-RAM)* cache as a viable alternative for reducing the write energy and latency. LARS cache comprises of multiple STT-RAM units with different retention times, with only one unit on at a given time. LARS dynamically determines which STT-RAM unit to power on during runtime, based on executing applications' needs. Our experiments show that LARS cache is low-overhead, and can reduce the average energy and latency by 35.8% and 13.2%, respectively, as compared to the dynamic refresh scheme.**

***Index Terms*—Spin-Transfer Torque RAM (STT-RAM) cache, configurable memory, low-power embedded systems, adaptable hardware, retention time.**

I. Introduction

Much research has focused on optimizing caches' performance and energy efficiency, due to the caches' non-trivial overall impact on embedded systems [1]. One of such emerging optimizations involves replacing the traditional SRAM cache with the non-volatile Spin-Torque Transfer RAM (STT-RAM). Apart from its non-volatility, the STT-RAM offers higher storage density than SRAM, low leakage power, and compatibility with CMOS technology [2], [3], [4]. However, dynamic operations in STT-RAM caches accrue significant overheads, compared to SRAM caches, due to long write latency and high dynamic write energy [4]. This paper aims to mitigate these overheads associated with STT-RAM caches. Specifically, we target the L1 cache, since it typically has the highest number of dynamic operations in the cache hierarchy.

The STT-RAM was originally developed to preserve data for up to ten years in the absence of an external power source [5]. This duration is known as the *retention time*. Prior work [6] has shown that longer retention times translate to increased write latency and energy, especially in resource-constrained embedded systems. In addition, such long retention times are usually unnecessary, since data is typically only needed in the cache for no longer than one second [2]. To reduce the write latency and energy, the retention time can be reduced, such that it is just sufficient to hold cached data.

Much prior work has explored the benefits of significantly relaxing the retention time [7], [4], [2], [8], [9]. Sometimes, the retention time is shorter than the duration for which data blocks must remain in the cache. To prevent the premature eviction of data blocks, the *dynamic refresh scheme (DRS)* [7], [4], [2] continuously refreshes the blocks that must remain in the cache beyond the retention time. However, the refreshes—involving multiple read/write operations—accrue runtime overheads and limit the optimization potential [8].

To mitigate these overheads, a few techniques (e.g., [8], [10]) have been developed to reduce refreshes. These techniques, however, typically rely on compiler-based data rearrangement, and the associated overheads, including compilation time and the costs of extra physical circuits to implement the techniques [8], [9]. In addition, these techniques typically feature a single retention time throughout the cache's lifetime [8], [10], even though different applications may require different retention times based on the lifetimes of the applications' data blocks.

Our work is motivated by three key observations: 1) different applications may require different retention times; 2) rather than performing multiple refreshes on cache blocks with longer lifetimes than the static retention time, more energy can be saved by using a longer retention time; and 3) a lower retention time than the static retention time can reduce the write energy and latency, if the reduced retention time does not result in excessive cache misses.

Given that cache blocks' lifetimes depend on different applications' execution characteristics, we propose that the STT-RAM cache's access energy and latency can be substantially reduced by dynamically adapting the retention time to the applications' individual needs. The retention time, however, is an inherent physical characteristic of STT-RAMs [5], which precludes physical runtime adaptability. Therefore, we explore a technique for logically adapting the retention time to executing applications' needs.

In this paper, we propose ***LARS: Logically Adaptable Retention time STT-RAM*** as a viable option for achieving dynamically adaptable retention times in STT-RAMs. Since STT-RAM cache units are much smaller in area than SRAM caches, we propose a LARS cache that comprises of multiple

STT-RAM cache units, wherein only one unit is on at any given time based on executing applications' retention time requirements. Our major contributions are summarized as follows:

- To the best of our knowledge, this paper is the first to propose dynamically adaptable retention time to reduce STT-RAM's write energy and latency. To this end, we explore our idea of logical adaptation and its potentials for reducing energy and latency.
- We explore and evaluate simple and easy-to-implement algorithms to dynamically determine the best retention times during runtime.
- We compare LARS to both the SRAM and DRS to investigate its potentials. Experiments reveal that, compared to the state-of-the-art DRS, LARS can reduce the average STT-RAM cache energy and latency by 35.8% and 13.2%, respectively.

II. Background and Related Work

The STT-RAM's basic structure, comprising of magnetic tunnel junction (MTJ) cells, and characteristics have been detailed in prior work [2], [11]. In this section, we present a brief overview of prior work on volatile STT-RAMs that provides the background for LARS.

A. Refresh Schemes on Volatile STT-RAM Cache

Prior work has shown that reducing the STT-RAM's retention time (i.e., volatile STT-RAMs) can significantly reduce the write energy and latency [7], [4], [2]. To prevent data loss in STT-RAMs with reduced retention times, Sun et al. [4] proposed the *dynamic refresh scheme* (DRS), which uses a counter to monitor how long each cache block is in the cache, in order to prevent premature expiry. Once the counter reaches its maximum value, the cache controller continuously refreshes the cache block until its lifetime expires (e.g., through eviction).

The dynamic refresh scheme incurs energy overheads due to the large number of refreshes. To reduce this overhead, Jog et al. [2] proposed the *cache revive* scheme, a flavor of DRS. Rather than refreshing all the cache blocks as proposed in [4], the cache revive scheme uses a small buffer to temporarily hold cache blocks that have prematurely expired due to the retention time. The most recently used cache blocks are then copied back into the cache and refreshed.

More recent works used compiler-based techniques—such as code optimization [8] and loop scheduling [10]—to reduce refreshes. However, these works preclude runtime optimization and incur overheads, since they typically rely on dedicated hardware to deal with the data loss in volatile STT-RAM cache.

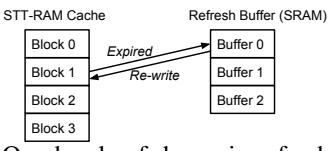


Fig. 1: Overheads of dynamic refresh scheme

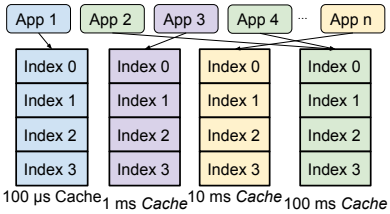


Fig. 2: STT-RAM retention time adapts to applications.

B. Cost of Refresh Schemes and Motivation for LARS

Fig. 1 illustrates some of the overheads incurred by the dynamic refresh scheme. Consider an STT-RAM cache with four blocks. Assuming that the retention time has elapsed, but Block 1 has not expired, the data is refreshed by copying the cache block into an SRAM refresh buffer and written back into the STT-RAM cache. Each refresh costs read and write operations to transfer the block between the STT-RAM and SRAM caches. The energy overheads of these operations can be prohibitive [8], especially in embedded systems.

Jog et al. [2] studied applications' cache block lifetimes to revealed that different applications have different cache block lifetimes and retention time requirements. We performed further studies to reveal that, based on the variable block lifetimes in different applications, retention times can be adapted to the applications to reduce the access energy and latency.

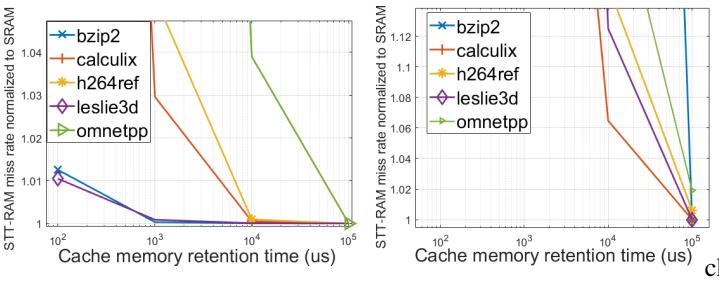
III. Logically Adaptable Retention Time STT-RAM (LARS) Cache

Fig. 2 illustrates how LARS works. Given a constrained design space of retention times, assume that $App1$'s best retention time is 100 μs, $App2$'s and $App4$'s best retention time is 100 ms, and so on. LARS allows the different applications to execute on cache units with retention times that match the applications' needs, in order to reduce the energy and latency.

A. Retention Time Analysis

To motivate our work, we analyzed how retention times affect applications' cache miss rates. Fig. 3 illustrates the relationship between cache miss rates and retention times for different applications in the SPEC 2006 [12] benchmark suite. The miss rates for the different STT-RAM retention times are normalized to the applications' SRAM miss rates with the same cache configuration. For clarity, Fig. 3 only shows five applications; we note, however, that the trend was similar for all the applications. Since a higher retention time implies higher energy and latency, our goal was to find the lowest (best) retention times that maintained comparable cache miss rates to the SRAM.

In general, the miss rates decreased as the retention times increased for all the applications. However, we observed different behaviors between the data and instruction caches. For the data cache (Fig. 3a), the retention times that achieved low cache miss rates varied for the different applications. For example, *bzip2*'s and *leslie3d*'s best retention time were 100μs, *calculix*'s and *h264ref*'s were 10ms, and *omnetpp*'s was 100ms. For the instruction cache (Fig. 3b), however, 100ms was consistently the best retention time for all applications.



(a) Data cache (b) Instruction cache

Fig. 3: STT-RAM cache miss rate changes under different retention times (normalized to SRAM, baseline of 1)

Since embedded systems applications are typically unknown at design time, our observations suggest that for optimal energy and latency in STT-RAM data caches, the retention times must be dynamically adapted to match changing application needs. Considering the consistency of the best retention time for the instruction cache (Fig. 3b)—instructions typically have less runtime variability than data—we decided to keep the instruction cache's retention time at 100ms, and focused the rest of our work on the data cache.

B. LARS Architecture

The STT-RAM is 3 to 9 times denser than the SRAM [4], [2]. However, for speed considerations, L1 cache sizes are typically limited (e.g., 16 – 32KB), especially in embedded systems devices. Thus, for the same memory size, an STT-RAM would take up a much smaller physical area than the SRAM. Based on this physical characteristic of the STT-RAM, we propose a LARS cache that comprises of four STT-RAM units; these four STT-RAM units will take up approximately the same area as one SRAM cache of the same size.

Fig. 4 depicts the proposed LARS architecture. The LARS cache comprises of four STT-RAM units with four different data memory retention times. The cache also comprises of four tag memory units, a status array with each element containing a valid bit, dirty bit—we assume a write-back cache—and *monitor counter* bits. To save energy, only one unit is on at a time, depending on the retention time needs of executing applications. Each element of the array indicates the status of one cache block. Although an expiring cache block is not refreshed in the LARS cache, we use the monitor counter to determine when to eliminate an expiring cache block (through invalidation, for example). We adopt the monitor counter from [4], and assume a monitor clock whose period is N times smaller than the retention time. Therefore, when a block's monitor counter goes from 0 to N, the block has reached its maximum retention time and should be invalidated.

Fig. 5 shows the state machine of the n-bit monitor counter for each cache block. The state machine comprises of states S_0 to S_N, which advance on the monitor clock's rising edge. In each state, the monitor counter resets to S_0 once the cache block receives a write or invalidate command. When the monitor counter's state is S_N, the counter begins the expiring process and sets the E signal. The E signal triggers LARS to

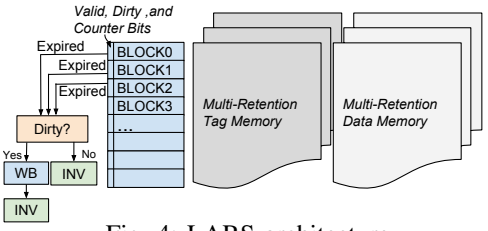


Fig. 4: LARS architecture

check the block's dirty bit. If the block is dirty, LARS will flush the block to the next memory level. Otherwise, LARS will only invalidate the cache block. Note that LARS requires minimal modifications to the cache controller, since these processes (e.g., writing back/invalidating a cache block) are implemented in state-of-the-art cache controllers. We discuss the monitor counter's overhead in Section III-D.

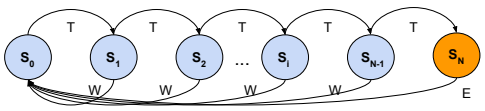
C. Determining the Best Retention Time

We assume that the cache controller [4], [7], [2] orchestrates the powering on/off of the appropriate STT-RAM units. To enable a non-intrusive process of determining the best retention time, we designed a low-overhead hardware *LARS tuner* to implement the algorithms described herein (the tuner overheads are described in Section III-D). The choice of retention time determines which LARS STT-RAM unit is powered on for different executing applications. Thus, we explored different techniques for dynamically determining an application's best retention time.

1) Sampling Technique: First, we considered a simple sampling technique that determines the *best* retention time by exhaustively sampling every available retention time. The application is executed on each STT-RAM unit for a *tuning interval*—we used intervals of 100million instructions—during which the energy consumption is measured. After sampling all retention times, the best retention time (lowest energy) is then selected and stored in a low-overhead data structure for subsequent use. The tuning overheads (time and energy) from this sampling technique are not prohibitive, since there are only four retention time options in the proposed LARS cache. With a tuning interval of 100 million instructions, tuning takes place during execution of the first 400million instructions, after which the optimal retention time is determined. Since applications can run for trillions of instructions [12], the tuning overheads are rapidly amortized during execution.

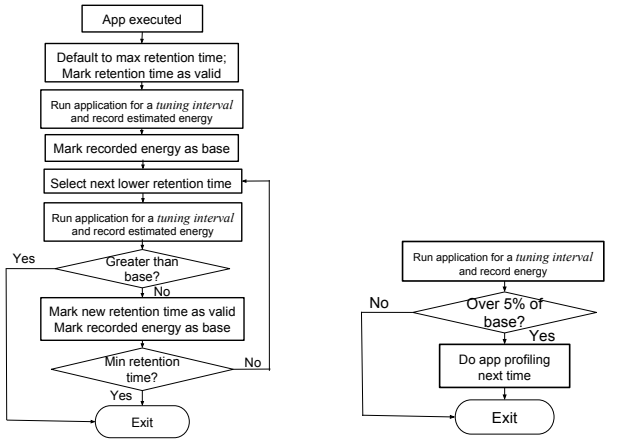
We used energy as the evaluation metric because we observed that the energy-based approach provided an optimal balance between energy and latency optimization, as compared to a latency-based approach (details in Section V-A).

2) LARS Tuning Algorithm: For easy practical implementation, we also designed a simple tuning algorithm—*LARS-Optimal*—to determine the best retention times during runtime. The algorithm determines the best retention time for energy,



T:Counter Pulse Width, W:Write/Invalidate, E:Expired

Fig. 5: Monitor counter state machine for each cache block



(a) LARS-Optimal tuning algorithm (b) Checking process

Fig. 6: LARS-Optimal Tuning Algorithm

using a cache energy model [13] based on the number of cache accesses, writebacks, misses, and the associated latencies.

Fig. 6a illustrates the *LARS-Optimal* tuning algorithm, which runs during an application's first execution. When the application begins, LARS defaults to the maximum retention time. The application is then executed for a *tuning interval*, during which the execution statistics are collected from hardware performance counters [14] and the energy consumed is calculated using the energy model. For our experiments, we used a tuning interval of 100 million instructions to provide a balanced tradeoff between tuning overhead and accuracy; this interval, however, can be adjusted based on specific system tradeoffs [15].

Fig. 7 illustrates our datapath, which implements the energy model for calculating the energy. The datapath uses a multiply-accumulate (MAC) unit, comprising of a multiplier, *intermediate register*, and adder, to calculate the current energy. The circled numbers in Fig. 7 represent the order in which the controller state machine selects data items for the MAC. The calculated *current energy* is stored as the *base energy* for comparison during the tuning process.

As shown in Fig. 6a, LARS iterates through the retention times in descending order for one tuning interval per retention time, and compares the current energy value to the base energy. If the calculated energy with the current retention time is less than or equal to the previous energy, the current retention time is stored and the base energy is updated to

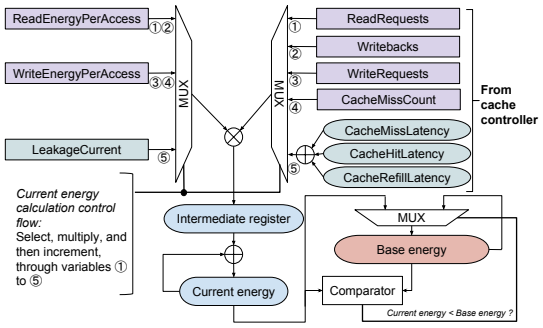


Fig. 7: Datapath for the energy model

the current energy. Otherwise, the previous retention time is retained for the application, after which the tuning process exits. The retention time and energy values are stored in a small low-overhead hardware data structure (Section III-D) for non-intrusive functioning.

LARS-Optimal also features a *checking process*, shown in Fig. 6b, that acts as a feedback on subsequent executions of the application. On each subsequent execution, the energy consumed during the first interval is calculated and compared to the stored value. If the calculated energy value deviates from the stored value by more than 5%, LARS returns to the tuning process for that application. Otherwise, the current retention time is used to run the application. We added the checking process to enable LARS to react to any runtime changes, such as new data inputs.

D. LARS Overheads

The main LARS overheads result from the *LARS hardware*, *runtime tuning*, and the *switching overheads*. We estimated the overheads using Verilog implementations, synthesized with Synopsys Design Compiler [16].

LARS' hardware overheads result from the monitor counters (Section III-B) and the LARS tuner. The tuner implements the LARS-Optimal algorithm (Section III-C), datapath for calculating the energy (Fig. 7), and data storage for retention time and energy histories (Section III-C). The number of bits, n, required for the monitor counter is equal to $log_2 N$, where N is the number of monitor clock periods within the retention time. For example, given a 100μs retention time and a monitor clock period of 10μs, $N = 10$, and $n = 4$. A 32KB cache with 64B lines has 512 monitor counters for each STT-RAM unit; each monitor counter requires 4 bits. For this cache, LARS constitutes an area overhead of 0.78%.

We synthesized the tuner with SAED_EDK90 Synopsys standard cell library. The estimated area overhead was 0.0145 mm^2, dynamic power was 28.04 mW, and leakage power was 422.68 μW. With respect to the ARM Cortex-A15 [17], the tuner's overhead is negligible (around 0.095%).

The switching overhead is the latency incurred while migrating the cache state (tag and data) from one STT-RAM unit to another during the LARS tuning process. In the worst case, we estimated that the migrations took approximately 4608 cycles and 57.34nJ energy.

IV. Experimental Setup

To evaluate and quantify the benefits of LARS, we modified the GEM5 simulator [18] to model LARS. We added a new retention parameter to GEM5 in order to model the variable retention time behavior. We also implemented DRS [4]—the most related work to ours—in GEM5 to enable comparison with the state-of-the-art[1].

To model a state-of-the-art embedded system microprocessor, we used configurations similar to the ARM Cortex A15 [14]. The microprocessor features a 2GHz clock frequency,

[1]The modified GEM5 version can be found at www.ece.arizona.edu/tosiron/downloads.php

TABLE I: Cache parameters of SRAM and STT-RAM with different retention times

Cache Configuration	32KB, 64B line size, 4-way				
Memory Device	SRAM	STT-RAM-100μs	STT-RAM-1ms	STT-RAM-10ms	STT-RAM-100ms
Write Energy (per access)	0.033nJ	0.040nJ	0.056nJ	0.076nJ	0.101nJ
Read Energy (per access)	0.033nJ	0.012nJ	0.012nJ	0.011nJ	0.011nJ
Leakage Power	38.021mW	1.753mW			
Hit Latency (cycles)	2	2	2	2	2
Write Latency (cycles)	2	3	4	5	7

separate 32KB L1 instruction and data caches, and an 8GB main memory. We used a base retention time of 10ms for DRS, similar to prior work [2], [4]. Our LARS cache comprises of four STT-RAM units with 100μs, 1ms, 10ms, and 100ms retention times. We chose the retention times to be as low as possible without excessively increasing the cache miss rates with respect to the SRAM, while covering a range of application requirements. Note that the cache can be designed with more STT-RAM units (and different retention times), depending on the design objectives.

We used the MTJ cell modeling technique proposed in [11] to obtain essential parameters, such as the write pulse, write current, and MTJ resistance value R_{AP}. We then applied these parameters to the circuit modeling tool, NVSim [19], in order to construct the STT-RAM cache for different retention times, as shown in Table I. To compare LARS with the SRAM cache, we modeled the SRAM using CACTI [20].

To represent a variety of workloads, we used twelve benchmarks from the SPEC 2006 benchmark suite compiled for the ARM instruction set architecture, using the reference input sets. Prior work has shown that these benchmarks are suitable for embedded systems research due to the compute and memory intensity of emerging embedded systems applications [21].

V. Results

To illustrate LARS' effectiveness, we evaluated and analyzed the L1 data cache's energy and the memory access latency achieved by LARS compared to the SRAM and DRS, representing prior work. In this section, we first summarize the results from the LARS sampling technique, and thereafter present results for LARS-Optimal.

A. LARS Sampling Technique

We evaluated the sampling technique based on different metrics—energy-, latency-, and EDP-based approaches—to determine which metric to use for the LARS tuning algorithm. Fig. 8 illustrates the energy consumed by the different sampling approaches normalized to DRS. Both the EDP- and energy-based approaches reduced the average energy by 35.8%, while the latency-based approach reduced the energy by 30.1%.

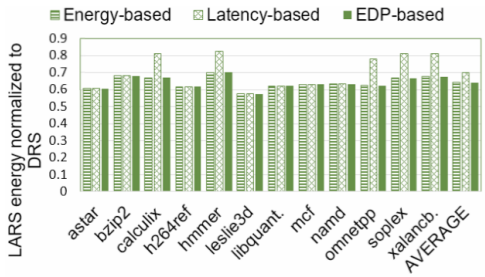


Fig. 8: Energy of sampling approaches normalized to DRS

Similarly, on average compared to DRS, the EDP- and energy-based approaches reduced the latency by 11.1%, while the latency-based approach reduced the latency by 13.2%. Finally, the EDP- and energy-based approaches reduced the EDP by 46.3%, while the latency-based approach reduced the EDP by 45.3%. The energy-based approach performed best in all instances, except for latency reduction, where the latency-based approach only improved over the energy-based approach by a marginal 2.1%. We have omitted the graphs for brevity.

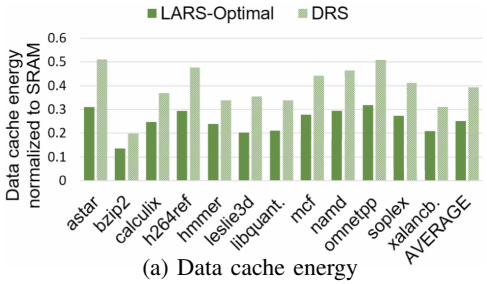
B. LARS-Optimal Compared to the SRAM and DRS

We observed that LARS significantly reduced the energy consumption as compared to the SRAM. Fig. 9 depicts the cache energy and latency achieved by both LARS-Optimal and DRS normalized to the SRAM. Fig. 9a shows that, on average across all the applications, LARS-Optimal reduced the energy by 75% as compared to the SRAM, with energy savings as high as 86.4% for *bzip2*. We note that part of this energy reduction was accounted for by the significantly reduced leakage power that the STT-RAM offers as compared to the SRAM (Table I). Thus, both LARS-Optimal and DRS significantly reduced the energy compared to the SRAM.

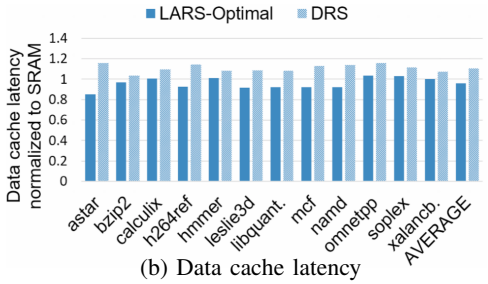
Fig. 9b shows that on average, LARS-Optimal reduced the latency by 4% as compared to the SRAM, with reductions as high as 15.1% for *astar*. As shown in Fig. 9b, while the latency improvement over the SRAM was not significant, LARS did not degrade the latency, whereas DRS degraded the latency by 8.5%.

Compared to DRS, Fig. 9a shows that LARS reduced the average energy by 35.8%, with energy savings as high as 42.6% for $leslie3d$. This energy reduction was directly tied to the reduction in the dynamic energy resulting from the dynamic refreshes featured in DRS. Fig. 9b shows that LARS reduced the average latency by 13.2%, as compared to DRS, with latency reductions as high as 26.9% for $astar$.

In general, LARS reduced the write energy and latency by adapting the retention time to different applications' needs, and by using lower retention times when appropriate. In addition, LARS eliminated the need for dynamic refreshes, which was a source of overhead in DRS. We also observed that LARS performed best for applications that maintained low cache misses despite a low retention time. For example, as seen in Fig. 3a, some applications, like $leslie3d$ and $bzip2$, maintained low cache misses as the retention time reduced.



(a) Data cache energy



(b) Data cache latency

Fig. 9: LARS and DRS data cache energy and latency normalized to SRAM

Concomitantly, LARS' improvements over DRS were higher than average for these applications—42.6% and 31.8% energy savings, respectively.

C. Exploring a Synergy Between LARS and DRS

We were further interested in exploring the synergy of LARS with DRS to achieve additional energy and latency improvements. Thus, we also implemented a theoretical scheme that combined LARS and DRS, featuring the best retention time (equivalent to LARS-Optimal) and a refresh mechanism to prevent premature data evictions (equivalent to DRS). We observed only marginal improvements from this synergistic scheme as compared to LARS-Optimal. The synergistic scheme improved the average energy and latency by 1.5% and 2.8%, respectively (graphs omitted for brevity).

VI. Conclusions and Future Research

In this paper, we propose *LARS: Logically Adaptable Retention Time STT-RAM* cache. LARS cache logically adapts the STT-RAM's retention time to different applications' runtime requirements, in order to reduce the write energy and latency. LARS comprises of multiple STT-RAM units with different retention times; only one unit is powered on at a time, depending on an application's needs. Experiments show that LARS can reduce the average energy and latency by up to 35.8% and 13.2%, respectively, as compared to a dynamic refresh scheme. For future work, we plan to explore LARS' impact for dynamic phase-based application characteristics. We will also extend LARS to dynamically tune other cache configurations, such as cache size, line size, and associativity in order to fully satisfy executing applications' resource requirements.

References

[1] S. Mittal, "A survey of architectural techniques for improving cache power efficiency," *Sustainable Computing: Informatics and Systems*, vol. 4, no. 1, pp. 33–43, 2014.

[2] A. Jog, A. K. Mishra, C. Xu, Y. Xie, V. Narayanan, R. Iyer, and C. R. Das, "Cache revive: Architecting volatile stt-ram caches for enhanced performance in cmps," in *DAC Design Automation Conference 2012*, June 2012, pp. 243–252.

[3] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen, "Circuit and microarchitecture evaluation of 3d stacking magnetic ram (mram) as a universal memory replacement," in *Proceedings of the 45th Annual Design Automation Conference*, ser. DAC '08. New York, NY, USA: ACM, 2008, pp. 554–559. [Online]. Available: http://doi.acm.org/10.1145/1391469.1391610

[4] Z. Sun, X. Bi, H. Li, W. F. Wong, Z. L. Ong, X. Zhu, and W. Wu, "Multi retention level stt-ram cache designs with a dynamic refresh scheme," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2011, pp. 329–338.

[5] Z. Diao, Z. Li, S. Wang, Y. Ding, A. Panchula, E. Chen, L.-C. Wang, and Y. Huai, "Spin-transfer torque switching in magnetic tunnel junctions and spin-transfer torque random access memory," *Journal of Physics: Condensed Matter*, vol. 19, no. 16, p. 165209, 2007. [Online]. Available: http://stacks.iop.org/0953-8984/19/i=16/a=165209

[6] C. Xu, D. Niu, X. Zhu, S. H. Kang, M. Nowak, and Y. Xie, "Device-architecture co-optimization of stt-ram based memory for low power embedded systems," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2011, pp. 463–470.

[7] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan, "Relaxing non-volatility for fast and energy-efficient stt-ram caches," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, Feb 2011, pp. 50–61.

[8] Q. Li, J. Li, L. Shi, C. J. Xue, Y. Chen, and Y. He, "Compiler-assisted refresh minimization for volatile stt-ram cache," in *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2013, pp. 273–278.

[9] G. Rodríguez, J. Touriño, and M. T. Kandemir, "Volatile stt-ram scratchpad design and data allocation for low energy," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 38:1–38:26, Dec. 2014. [Online]. Available: http://doi.acm.org/10.1145/2669556

[10] K. Qiu, J. Luo, Z. Gong, W. Zhang, J. Wang, Y. Xu, T. Li, and C. J. Xue, "Refresh-aware loop scheduling for high performance low power volatile stt-ram," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, Oct 2016, pp. 209–216.

[11] K. C. Chun, H. Zhao, J. D. Harms, T. H. Kim, J. P. Wang, and C. H. Kim, "A scaling roadmap and performance evaluation of in-plane and perpendicular mtj based stt-mrams for high-density cache memory," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 2, pp. 598–610, Feb 2013.

[12] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[13] T. Adegbija and A. Gordon-Ross, "Phase-based cache locking for embedded systems," in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, ser. GLSVLSI '15. New York, NY, USA: ACM, 2015, pp. 115–120. [Online]. Available: http://doi.acm.org/10.1145/2742060.2742076

[14] "Cortex-A15 Processor." [Online]. Available: https://www.arm.com/products/processors/cortex-a/cortex-a15.php

[15] A. Gordon-Ross and F. Vahid, "A self-tuning configurable cache," in *Proceedings of the 44th annual Design Automation Conference*. ACM, 2007, pp. 234–237.

[16] D. Compiler, "Synopsys inc," 2000.

[17] F. A. Endo, D. Couroussé, and H.-P. Charles, "Micro-architectural simulation of embedded core heterogeneity with gem5 and mcpat," in *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, ser. RAPIDO '15. New York, NY, USA: ACM, 2015, pp. 7:1–7:6. [Online]. Available: http://doi.acm.org/10.1145/2693433.2693440

[18] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[19] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 31, no. 7, pp. 994–1007, Jul. 2012. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2012.2185930

[20] "HP Labs: CACTI." [Online]. Available: http://www.hpl.hp.com/research/cacti/

[21] M. Domeika, *Software development for embedded multi-core systems: a practical guide using embedded Intel architecture*. Newnes, 2011.