HALLS: An Energy-Efficient Highly Adaptable Last Level STT-RAM Cache for Multicore Systems

Kyle Kuan, *Student Member, IEEE* and Tosiron Adegbija, *Member, IEEE*

Abstract—Spin-Transfer Torque RAM (STT-RAM) is widely considered a promising alternative to SRAM in the memory hierarchy due to STT-RAM's non-volatility, low leakage power, high density, and fast read speed. The STT-RAM's small feature size is particularly desirable for the last-level cache (LLC), which typically consumes a large area of silicon die. However, long write latency and high write energy still remain challenges of implementing STT-RAMs in the CPU cache. An increasingly popular method for addressing this challenge involves trading off the non-volatility for reduced write speed and write energy by relaxing the STT-RAM's data retention time. However, in order to maximize energy saving potential, the cache configurations, including STT-RAM's retention time, must be dynamically adapted to executing applications' variable memory needs. In this paper, we propose a highly adaptable last level STT-RAM cache (HALLS) that allows the LLC configurations and retention time to be adapted to applications' runtime execution requirements. We also propose low-overhead runtime tuning algorithms to dynamically determine the best (lowest energy) cache configurations and retention times for executing applications. Compared to prior work, HALLS reduced the average energy consumption by 60.57% in a quad-core system, while introducing marginal latency overhead.

Index Terms—Spin-Transfer Torque RAM (STT-RAM) cache, configurable memory, low-power systems, adaptable hardware, retention time, last level cache, multicore systems, computer architecture, energy-efficient, runtime adaptable systems.

1 Introduction

Multicore architectures have become mainstream due to the growing demand of compute- and memory-intensive applications. Consequently, to bridge the processor-memory performance gap, much effort is being placed on designing more efficient memory hierarchies, especially for resource-constrained systems. Such designs typically involve provisioning the system with a larger last level cache (LLC) to enable higher computing throughput and alleviate challenges associated with limited main memory bandwidth [1]. For example, the ARM Cortex A15 [2] allows implementations that feature four cores with a shared 1MB L2 cache. However, the LLC, which is typically implemented using conventional SRAM, imposes significant overheads with respect to leakage power and silicon area; these overheads could be prohibitive for resource-constrained systems.

To address some of these challenges, non-volatile memory (NVM) technologies have emerged as a viable alternative to SRAMs for implementing LLCs [3]. Among several emerging NVM technologies, the *Spin-Transfer Torque RAM (STT-RAM)* is considered to be one of the most promising candidates to replace the SRAM [4]. STT-RAM, apart from its non-volatility, has other advantages, including high storage density, low leakage current, and compatibility with CMOS technology [5]. However, implementing caches using STT-RAMs is still challenging due to the overheads imposed by STT-RAM's long write latency and high dynamic write energy [6]. Prior studies have revealed that the STT-RAM consumes 6-14 times more energy per write access than the SRAM [7].

A popular approach for reducing STT-RAM's write latency and energy involves reducing the STT-RAM cell's data retention time. Smullen at al. [8] showed that relaxing the retention time—the duration for which data blocks remain in memory in the absence of power—can substantially reduce both latency and energy. Furthermore, Jog et al. [9] observed that several benchmarks did not require data retention time of more than one second. Consequently, the STT-RAM's intrinsic retention time, which could be up to ten years, is unnecessary, and even undesirable in terms of energy and latency. However, the reduced retention time can sometimes be shorter than the cache blocks' requirements. To maintain data correctness, prior works propose the *dynamic refresh scheme (DRS)* [10], [8], [6], [9], which refreshes data blocks to prevent premature expiry. The refreshes incur additional overhead, which are especially worse in the LLC. Compared to the level one (L1) cache, the LLC typically requires longer cache block lifetime for data reuse. In addition, a larger number of cache blocks—as is the case in the LLC—increase the minimum number of refreshes required to hold cached data, which can drastically impact the scalability of the STT-RAM cache [11].

Our work aims to mitigate the overheads imposed by STT-RAM's write energy, especially in resource-constrained systems. To this end, the work proposed herein is inspired by two important observations: 1) due to the variability in data block lifetimes, different applications may require different retention times, and these requirements may change

The authors are with the Department of Electrical and Computer Engineering, The University of Arizona, USA, e-mail: {ckkuan, tosiron}@email.arizona.edu.

during runtime and 2) different applications, and combination of applications (i.e., workloads) may require different STT-RAM cache configurations (i.e., cache size, line size, and associativity) in order to minimize the energy consumption.

In this paper, we propose and explore a *highly adaptable last level STT-RAM cache (HALLS)* as a viable alternative for reducing STT-RAM's write energy for LLC implementation. HALLS exploits the synergies of cache configurability [12] and retention time adaptability. HALLS features a multi-banked cache that enables cache configuration through bank shutdown (to configure the cache size), bank concatenation (to configure the associativity), and multi-line fetch (to configure the line size). The different cache banks are provisioned with different retention times that can satisfy a variety of applications' requirements. Based on runtime profiling, data blocks are opportunistically placed in cache banks that offer the right-provisioned retention time for energy minimization, without substantially degrading the latency.

Our contributions are summarized as follows:

- We propose to design the STT-RAM LLC with different retention times in the different cache banks, and at finer granularity than prior work [6]. Furthermore, our work leverages the synergy of adapting both the cache configuration and retention time to different application requirements.
- We explore and evaluate simple and easy-to-implement algorithms to determine the best cache configurations and retention time for each executing application.
- We show, through extensive analysis, that compared to prior work in a quad-core system, HALLS can reduce the energy by an average of 60.57%, while introducing minimal hardware overhead and a latency overhead of 1.47%. Compared to SRAM, HALLS achieved average energy savings of 70.12%, with a latency overhead of 5.16%.

2 Background, Related Work, and Motivation

From the system-level perspective, two popular approaches exist for mitigating STT-RAM's write latency and energy overheads in caches. The first involves removing as many unnecessary writes as possible. For example, dead write prediction (e.g., DASCA [11]) identifies dead writes—blocks that are written to the cache, but not reused thereafter—and bypasses cache accesses for those blocks. Wang et al. [13] also used dead write prediction to guide block placement in a hybrid SRAM/STT-RAM bank LLC. Flip-N-Write (FNW) [14] uses bit-wise comparisons to detect the difference between a block to be replaced, and the new block. The replaced block's contents are then updated by flipping bits that are different, in order to minimize unnecessary bit-writes, thereby reducing the write energy consumption as well as the latency. Similarly, the Encoded Content-Aware cache Replacement (ECAR) scheme [15] features a block replacement policy that reduces the number of switching bits by replacing the block whose contents are most similar to the missed block. While these prior works do not enable runtime adaptability, we consider them to be orthogonal and complementary to the work presented herein.

The second approach, on which we focus the related work discussion herein, involves substantially relaxing the retention time and incorporating the dynamic refresh scheme to ensure data correctness after the retention time elapses [8], [6], [9], [10], [16].

In this section, we present a brief background and overview of related work in order to motivate our approach. For brevity, we omit details of the circuit-level design [4], since that is not the focus of our work.

2.1 Refresh Schemes in Volatile STT-RAM Cache

STT-RAM stores data bits using a magnetic tunnel junction (MTJ) cell [4], [17]. Prior work [8] has shown that decreasing the MTJ cell's thermal stability can substantially reduce the STT-RAM's write latency and write energy. In effect, the MTJ cell only retains data for a limited time period—the retention time—beyond which the data would become unstable and lose correctness. To maintain data correctness, prior work [6], [9], [10] proposed techniques that dynamically refresh the cache blocks after the retention time has elapsed. Sun et al. [6] used a global clock to track all valid blocks and refreshed the blocks as needed. Jog et al. [9] refreshed only the first eight most recently used (MRU) blocks. The authors used a write buffer to handle the surge of refresh requests, given STT-RAM's long write time. Other techniques used compiler-assisted techniques to optimize data object organization in order to make refreshes more efficient [10], [16]. However, compiler-oriented techniques, which are typically static, are not amenable to dynamic runtime changes in application requirements.

2.2 Overhead of DRS

One of the key drawbacks of the dynamic refresh scheme (DRS) is the overhead accrued as a result of the refresh operations [9]. Typically, DRS requires a buffer that holds data blocks during the refresh operations. In [9], the authors used a 121.6 KB STT-RAM buffer that had 1900 slots. Assuming a 128 KB direct mapped buffer with 16B line sizes and a 10ms retention time, this buffer can consume up to 141.425mW of leakage power. Ideally, the buffer should have high associativity and larger block sizes, thus consuming even more power. Furthermore, each refresh operation consists of four physical operations: 1) STT-RAM cache read, 2) buffer write, 3) buffer read, and 3) STT-RAM cache write. Given a 1MB L2 STT-RAM cache with $100ms$ retention time, for example, our analysis revealed that each refresh operation would accrue about 1.311nJ in energy.

Similarly, the refresh scheme used in [6] accrued overheads due to the refresh buffer and its peripheral arbitration circuits. As such, leakage power was accrued during the refresh process, especially since every single L2 cache write took 3ns to 4ns. Also, even though the DASCA technique, for example, got rid of the need for refreshes by directly predicting and removing dead writes, the technique still accrued a 6.44 KB overhead for the SRAM buffer that was used to store the prediction table for a 1MB STT-RAM cache.

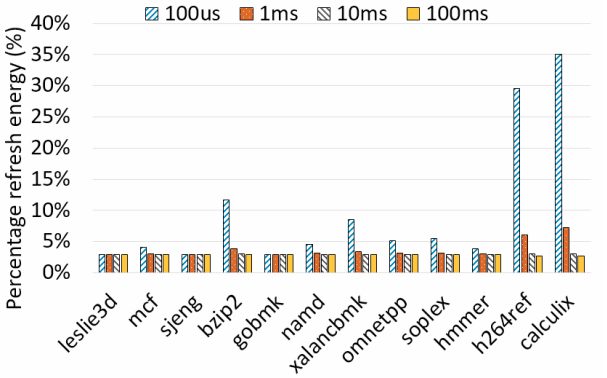


Fig. 1: Refresh energy percentage for different retention times

To further illustrate the impacts of refreshes on overall energy, we analyzed the refresh energy for different retention times while running a subset of SPEC CPU2006 benchmarks. Fig. 1 depicts the percentage of the overall energy that comprises of refresh energy (details of our experimental setup are in Section 4).

We observed that for some memory-intensive benchmarks like *leslie3d* and *bzip2*, the total number of refreshes, and thus, refresh energy, was very low. These benchmarks had several blocks that exhibited short lifetimes and were not needed in the cache beyond the retention times. However, the refresh mechanism still constituted about 3% of the total energy as a result of the buffer's leakage power. On the other hand, compute-intensive benchmarks like *h264ref* and *calculix* had blocks with longer lifetimes and higher number of refreshes, resulting in higher refresh energy. For example, *calculix*'s refresh energies were 35.06%, 7.25%, 3.08%, and 2.69% of the total energy for the 100μs, 1ms, 10ms, and 100ms retention times, respectively. Based on these observations, we sought to develop techniques to minimize—if possible, eliminate—the need for refreshing the cache blocks.

2.3 Adaptable STT-RAM caches

Prior works [18], [19], [13] have discussed the benefits of resource specialization in the STT-RAM cache. Most of these works focused on leveraging a hybrid SRAM/STT-RAM cache to gain the benefits of both SRAM's low access latency and STT-RAM's low leakage power. In general, these prior works proposed data placement techniques to determine when data blocks should be placed in either SRAM or STT-RAM array. For example, Chen et al. [19] proposed techniques to monitor the access counts of a set and predicted the set's future usage. Similarly, Wang et al. [13] used execution characteristics on the CPU, such as prefetch and current program counter (PC), to determine block placement. Imani et al. [18] proposed a cache swap policy to intentionally place majority-zero data in STT-RAM and majority-one in SRAM, thus reducing the number of writes of '1' in order to reduce the STT-RAM's energy consumption.

Another category of techniques for enabling STT-RAM cache adaptability focuses on profiling running applications and adapting the cache configurations based on the applications' cache requirements [12], [20]. These works focus on optimizing the L1 cache due to its impact on overall processor performance and energy consumption. Our work focuses on optimization opportunities for the shared last level caches since multicore systems are becoming increasingly ubiquitous in resource-constrained systems [21]. However, we note that the prior techniques discussed herein are orthogonal to our work, and can be complementary to our proposed approach.

3 Highly Adaptable Last Level STT-RAM (HALLS) Cache

3.1 Access Pattern and Retention Time Analysis in LLC

Unlike the L1 cache, which is usually separated into instruction and data caches, L2 caches—we use the L2 cache to represent the LLC in this work—are usually unified. That is, L2 caches do not distinguish between instruction and data cache blocks. Since instruction blocks would not be updated or written into from CPU, they simply rely on the memory cell to hold the block's value. Thus, intuitively, instruction blocks would require longer retention times than data blocks. Data cache blocks, on the other hand, may be updated frequently by the CPU, and would require a new *retention period* every time they are updated by a higher level cache (i.e., the L1 cache in our work).

Based on these observations, we hypothesized that LLC cache blocks can be sorted into different *retention time groups* depending on an executing application's code and data object behaviors. To test this hypothesis, we performed experiments using a quad-core processor featuring a shared 4-way, 128KB L2 cache with 64B blocks. We grouped ten multi-programmed workloads' data blocks into way-sized chunks of 32KB each, performed an exhaustive design space exploration of four retention times for each chunk, and then calculated the energy for each run. We empirically selected the retention times to satisfy a variety of applications' requirements, but note that these retention times may change for a different set of benchmarks.

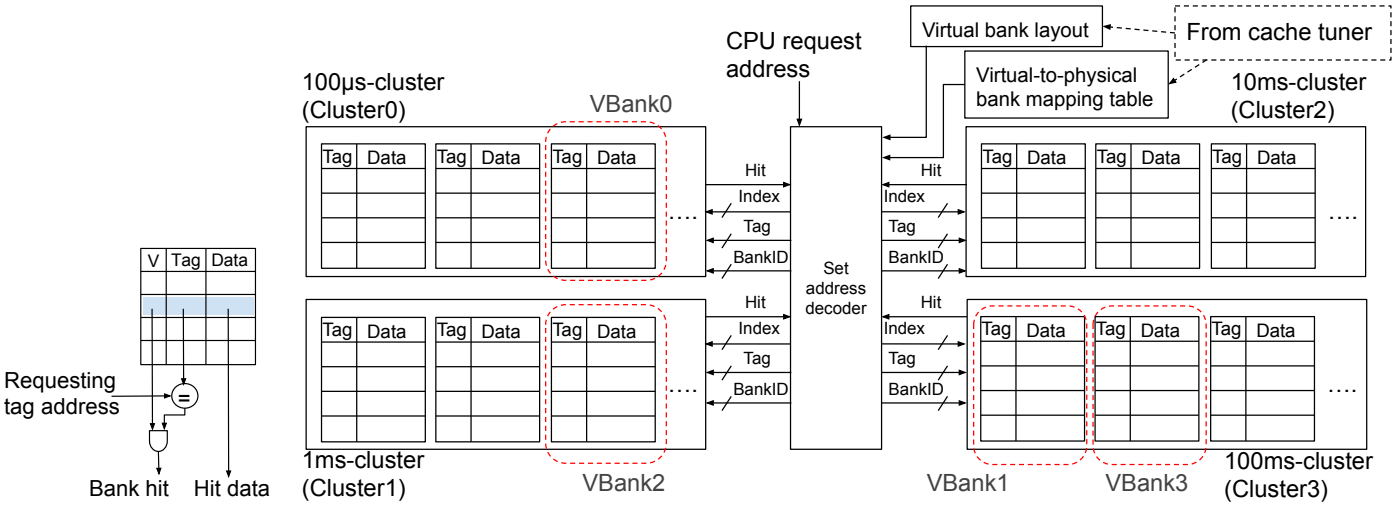
Fig. 2 illustrates the combination of retention times that achieved the lowest energy for a workload (for brevity, only four workloads are shown, but the analysis and insights applied to all the workloads). Our first observation was that for each workload, the placement of instructions and data blocks dictated the retention time requirement. For example, in Fig. 2a, *way0* was a hot region for data block write activities and short block lifetimes. Hence, the 100μs retention time consumed the lowest energy for that way. *Way1* and *way3* frequently stored instructions or read-only tables; thus, the 100ms retention time achieved minimum energy for those ways. Across the different workloads, we observed that the retention time requirements were indicative of the applications' execution behavior. These observations implied that energy savings can be achieved by provisioning the L2 cache with different retention time values to satisfy a variety of runtime retention needs.

3.2 HALLS Architecture

Fig. 3 illustrates the HALLS architecture. We assume a physical 1MB, 16-way STT-RAM L2 cache, since this size is found in state-of-the-art architectures [22], but note that the idea can be extended to any arbitrary size L2 (or L3) cache. Since the retention time is a physical characteristic of

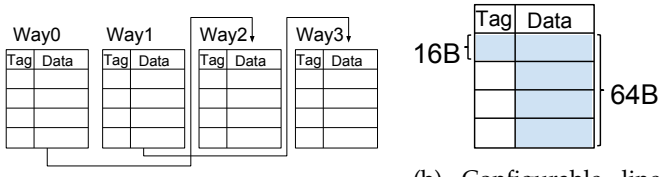
(a) Workload1 (b) Workload2 (c) Workload3 (d) Workload4

Fig. 2: Best energy retention time selection for different workloads in a 128KB, 64B line size, 4-way L2 cache. Each multi-programmed workload comprises of four benchmarks run in a quad-core system



(a) Detail of a cache bank (b) HALLS architecture

Fig. 3: HALLS architecture



(a) Configurable associativity (b) Configurable line size

Fig. 4: HALLS's capability of configurable associativity and line size

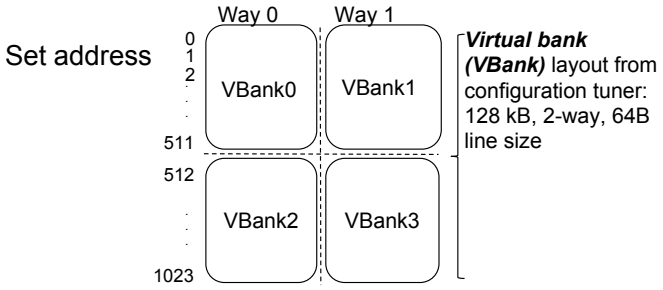


Fig. 5: An example of virtual bank layout

the STT-RAM [4], it cannot be easily dynamically adapted, unlike other cache configurations such as the cache size, line size, and associativity. Thus, we use a *logical adaptation* of the retention time [20], wherein different cache banks are designed with different physical retention times. During runtime, cache blocks can be written into the cache banks that most closely match the blocks' retention time requirements.

As illustrated in Fig. 3, the HALLS cache architecture is designed using 32KB banks, i.e., 32 banks in the 1MB STT-RAM cache. Runtime adaptability is achieved using similar mechanisms to SRAM-based configurable caches [23], which have been widely studied and analyzed in prior work. The cache size can be configured by shutting down cache banks, e.g., the 1MB L2 cache can be configured into a 512KB cache by shutting down 16 banks or into a 128KB cache by shutting down 28 banks. The associativity can be configured by concatenating different banks as shown in Fig. 4a. Finally, given a physical line size of 16B, multiple lines can be fetched to logically configure the line size, from 16B to 64B, as shown in Fig. 4b. As detailed in prior work [23], augmenting the cache for this adaptability is low-overhead and does not adversely impact the cache's critical path.

Fig. 3a shows details of a cache bank. Each cache bank contains a tag and a data array, along with a valid bit checker and a tag comparator. As such, each bank can operate independently, even when other banks are shut down to save power[12], [19]. Thus, apart from enabling the adaptability proposed herein, the 32-bank structure also lends itself to higher memory bandwidth in the L2 cache. The 32 banks are organized in 8-bank clusters, with each cluster designed with a different retention time. We used a total of four retention times—100μs, 1ms, 10ms, and 100ms—to satisfy a range of application requirements based on empirical analysis. However, more (or different) retention times can be used depending on the executing applications. We use $ClusterID$ to indicate these four retention time clusters. Cluster0, Cluster1, Cluster2, and Cluster3 represent 100μs,

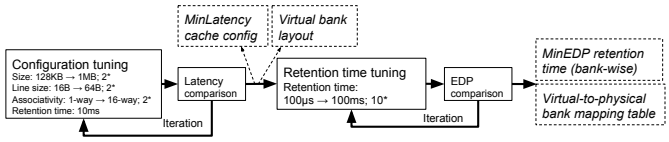


Fig. 6: Block diagram of HALLS flow

$1ms$, $10ms$, and $100ms$ retention time clusters, respectively.

To complete the tag look up for different configurations and different retention time clusters, we propose a modified set address decoder. As shown in Fig. 3b, the set address decoder receives the request address from CPU. Based on the cache configuration and retention time (as determined by the tuner) and the requested address, the set address decoder dispatches $Index$, Tag, and the corresponding $BankID$ to the appropriate retention time clusters and monitors hit bits. We define *virtual bank* ($VBank$) as a specific location of cache ways and set address from the point of view of the CPU request address. Based on the outcome of cache tuning, VBank represents the mapping of requested data blocks to the physical cache banks with the appropriate retention times. Fig. 5 shows a sample layout of virtual banks, given a best cache configuration output (from the tuner) of 128 KB, 2-way, and 64B line size. VBank0 represents the cache blocks located in set address 0-511 and way 0, VBank1 represents blocks in set address 0-511 and way 1, and so on.

The dotted boxes in Fig. 3b illustrate an example of virtual-to-physical bank mapping. When a request address has set address between 0 and 511, the set address decoder sends the index, tag, and the corresponding BankID to the 100μs-cluster (Cluster0) for VBank0 and 100ms-cluster (Cluster3) for VBank1. The BankID allows HALLS to perform bank-to-bank mapping between virtual and physical banks if multiple banks must be accessed in the same cluster. For example in Fig. 3b, 100ms-cluster (Cluster3) is able to distinguish VBank1's index and tag from VBank3's using the BankID.

The HALLS architecture also features a low-overhead hardware *tuner* that orchestrates the process of determining which cache configuration and retention time to use for an executing application. We decided to use a hardware tuner, as opposed to a software tuner, in order to make the tuner's operations non-intrusive to executing applications' behaviors (e.g., cache accesses). When a new application is run, the tuner begins sampling the application based on the tuning algorithm (Section 3.3) for a *tuning interval* to determine the best (i.e. minimum energy) cache configuration. After configuration tuning, HALLS has determined the cache configuration (cache size, line size, and associativity), and then constructs a *virtual bank layout* based on the determined configuration. For example, in Fig. 5, the virtual bank layout defines a 128 KB cache with 2-way set associativity and 64B line size. Thereafter, HALLS performs the retention time tuning and establishes the mapping of the virtual banks to physical banks. Tuning details are described in Section 3.3.

Even though HALLS eliminates the need for a refresh mechanism, we still incorporate a per-block counter to keep track of the cache blocks' lifetimes [9]. HALLS uses the counter to detect the expiration of a cache block and evicts the block when the retention time has elapsed. Dirty blocks are first written to the main memory before eviction. We implemented the counter as a state machine, with a clock period defined as the retention time divided by N, where N defines the granularity of block eviction. When a block is written to the cache, the counter's state advances from the initial state until it reaches the maximum state. The block is then evicted, and the counter is reset to the initial state whenever a new write operation occurs for the block. HALLS overheads are described in Section 5.

Algorithm 1: HALLS configuration tuning algorithm

Data: Size $S = \{s_{min}, ..., s_{max}\}$
Data: Line size $L = \{l_{min}, ..., l_{max}\}$
Data: Associativity $A = \{a_{min}, ..., a_{max}\}$
Result: Best cache size, line size, associativity

```
1  CurConfig.size ← s_max;
2  CurConfig.linesize ← l_max;
3  CurConfig.ways ← a_max;
4  Config ← CurConfig;
5  MinLatency ← Latency_max;
6  foreach p ∈ [size, linesize, ways] do
7      for (p_i = p_max; p_i >= p_min; p_i = p_i/2) do
8          CurConfig.p ← p_i;
9          CurLatency ← samplingLatency(CurConfig);
10         if CurLatency < MinLatency then
11             MinLatency ← CurLatency;
12             Config ← CurConfig;
13         end
14         else
15             break;
16         end
17     end
18 end
19 return Config,MinLatency;
```

3.3 Determining the best configuration

Fig. 6 depicts the high level flow of HALLS. HALLS first determines the best cache configuration using the configuration tuning algorithm (Algorithm 1). This algorithm yields the best cache configuration and the virtual bank layout. We used the latency as the algorithm's objective function based on our observation that tuning the configurations for latency improved the energy consumption, whereas tuning for energy substantially degraded the latency. We tested the algorithm using energy as the objective function and found that much smaller cache sizes were favored, thereby reducing the dynamic and leakage power. However, the resulting caches were severely under-provisioned for the applications' access requirements. On the other hand, tuning for latency allowed HALLS to tradeoff achieving optimal energy savings for reduced latency degradation.

After determining the best cache configuration, HALLS determines the best retention time using the retention time tuning algorithm (Algorithm 2). The retention time tuning algorithm uses the energy delay product (EDP) as the objective function in order to achieve energy savings without substantial latency degradation, and outputs the virtual-to-physical bank mapping that satisfies the executing application's requirements. In this work, we used application-based tuning, but plan to explore phase-based tuning in future work.

Algorithm 2: HALLS retention time tuning algorithm

Data: Retention time tuning sets $T = \{Set0, Set1, Set2, Set3\}$
Data: Requested virtual banks $V = \{VBank0, VBank1, ..., VBank31(as\ the\ highest)\}$
Result: Virtual-to-physical banks mapping table

```
1  foreach t ∈ T do
2  |   AllPhysicalBanksEDP ← samplingEDP(t);
3  |   foreach v ∈ V do
4  |   |   ClusterID,BankID ← findPhysicalBank(t, v);
5  |   |   BankEDP ← findPhysicalBankEDP(ClusterID, BankID, AllPhysicalBanksEDP);
6  |   |   v.EDP[ClusterID ] ← BankEDP;
7  |   end
8  end
9  foreach v ∈ V do
10 |   MinEDPCluster ← findMin(v.EDP);
11 |   ClusterID,BankID ← popAvailablePhysicalBank(MinEDPCluster);
12 |   MappingTable.push(v, ClusterID, BankID);
13 end
14 return MappingTable;
```

3.3.1 Configuration tuning algorithm

Algorithm 1 depicts the HALLS configuration tuning algorithm. The inputs to the algorithm are the cache design space, and the outputs are the best cache size, line size, and associativity. When a new application is executed, the algorithm defaults to the maximum configuration (i.e., 1MB size, 16-way set associative, and 64B line size in our HALLS architecture) (lines 1-3). HALLS then runs the application using each configuration for one tuning interval—we assumed an interval of 10M instructions in our experiments—while iterating through the configurations in descending order of the cache parameters' energy impact (cache size, followed by line size, followed by associativity). For each parameter, the configurations are explored as long as reducing the parameter value also reduces the latency (lines 6-18). The algorithm stops tuning if a configuration change increases the latency as compared to the current minimum latency (lines 14-16). The algorithm only needs to store the minimum-latency configuration, with which the current configuration's latency is compared.

3.3.2 Retention time tuning algorithm

To minimize implementation overhead for dynamically determining the best retention time (Fig. 6), we use a simple algorithm that samples all four retention times for a tuning interval. After each sampling period, HALLS collects the cache statistics from hardware performance counters and combines the statistics (e.g., read requests, write requests, writebacks, etc.) with predefined STT-RAM cache access parameters to estimate the energy consumption.

Since the HALLS cache features retention time clusters featuring different retention times in each cluster (8 banks/cluster in our case), it is not possible to set a single uniform retention time for the cache during tuning. Thus, we defined *retention time tuning sets* comprising of different retention time settings to guide the tuning process. The retention time tuning sets represent a mapping of *virtual banks* (i.e., bank-sized chunks of data blocks) to physical banks to enable sampling of the applications' data with the different retention times.

Algorithm 2 depicts the HALLS retention time tuning algorithm, which takes as input the retention time sets and the virtual banks, and outputs the virtual-to-physical bank mapping table. For clarity, we illustrate the retention time tuning process using Table 1, which shows the retention time tuning for the four virtual banks depicted in Fig. 3b. Assume that the best cache configuration—determined by the cache configuration tuning algorithm—is 128KB (i.e., requiring four banks), 2-way associativity, and 64B line size. The first iteration starts from *Set 0*, where *VBank 0,1,2,3* are allocated to Cluster0, Cluster1, Cluster2, and Cluster3, respectively. HALLS then samples the cache statistics for a tuning interval and collects the execution statistics (line 2).

Thereafter, HALLS shifts the clusters by one as shown in Table 1 and collects statistics after every iteration. In every iteration, HALLS stores bank-wise EDP of physical banks into the VBank object. For each VBank, `findPhysicalBank` (line 4) extracts the ClusterID and the BankID of the physical bank being sampled in the current iteration. Based on the collected statistics, HALLS calculates the EDP of the physical bank and stores it in VBank's EDP array (line 5-6). After the last iteration (*Set 3*), HALLS then selects the least EDP retention time as the best for each VBank. We note that there may be a limited availability of retention times, and the best retention time may not be available if multiple VBanks select a particular retention time (e.g., 10ms for 12 banks when there are only 8 banks with 10ms). In this case, `findMin` (line 10) searches the VBank's EDP array, checks if the physical banks in a cluster are available, and selects the best-performing available cluster. With the available ClusterID, HALLS then returns the ClusterID and the allocated BankID (line 11-12). Although we illustrate this process using only four VBanks (one for each retention time cluster), eight VBanks per cluster can be used, when necessary, to complete the tuning for the 1MB cache within four sampling intervals.

4 Experimental Setup

We evaluated and quantified the benefits of HALLS through extensive simulations using an in-house modified version of the GEM5 simulator [24]. We modified GEM5[1] to implement

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

TABLE 1: Retention time tuning example

Retention time tuning set	VBank-Cluster mapping	Retention time tuning set	VBank-Cluster mapping
Set 0	VBank0: Cluster0 VBank1: Cluster1 VBank2: Cluster2 VBank3: Cluster3	Set 2	VBank0: Cluster2 VBank1: Cluster3 VBank2: Cluster0 VBank3: Cluster1
Set 1	VBank0: Cluster1 VBank1: Cluster2 VBank2: Cluster3 VBank3: Cluster0	Set 3	VBank0: Cluster3 VBank1: Cluster0 VBank2: Cluster1 VBank3: Cluster2

both HALLS and DRS to represent prior work as described in [9], [6]. To enable a stringent comparison to our approach, we modeled DRS as a "perfect" refresh scenario, meaning that there were no extra misses caused by failed refreshes, there were no unnecessary refreshes, and there were no refresh-related latency overheads. We used the modeling technique proposed in [25] to estimate MTJ characteristics for different retention times. Based on the technique, we calculated write pulse, write current, and MTJ resistance value R_{AP} and R_P. With these parameters, we used NVSim [26] to construct the STT-RAM cache for the different retention times. We set the technology process to 22 nm to comply with the proper memory cell size that can exhibit a retention time as low as 100μs [25].

To model modern-day resource-constrained processors, we simulated dual and quad-core systems with configurations similar to processors such as the ARM Cortex A15, as shown in Table 4. We used retention times from 100μs to 100ms, which we empirically determined to satisfy a range of application requirements. We note that more retention times can be used at the expense of tuning complexity.

Table 2 depicts the cache parameters for the base SRAM and STT-RAM configurations. Table 3 depicts the leakage power and dynamic energy for different configurations to illustrate how these characteristics change with different STT-RAM retention times and with respect to SRAM. The configurations are denoted as xK-yW-zB, where x, y, and z represent the cache size (in KB or MB), associativity (in ways), and line size (in B), respectively. Note that these statistics change for the different cache configurations in the design space, but for brevity, we only show the numbers for the configurations selected by our algorithm. We also only show the energy statistics, since the write latency was constant for different retention times as shown in Table 2, and hit latency was 1 cycle across the different selected configurations. The STT-RAM leakage power values resulted from the peripheral/decode circuits and optimization for read latency in our simulations.

We observed that in the 512K-16W-64B cache, 100μs exhibited higher leakage power and dynamic energy than other higher retention times. This observation was an artifact of NVSim that we attribute to its bank organization for that configuration [26]. A smaller bank size was used for that cache size; as such, additional leakage was incurred from the peripheral circuits. For comparisons with HALLS, we also modeled the SRAM using NVsim [26] with 22 nm technology.

To represent a variety of workloads, we used twelve benchmarks from the SPEC CPU2006 benchmark suite compiled for the ARM instruction set architecture, using the reference input sets. We created ten multi-programmed workload comprising of two and four randomly selected benchmarks for the dual- and quad-core experiments, respectively—with one benchmark running on each core—ensuring that all twelve benchmarks used were represented in the workloads. The workload composition is shown in Table 5.

5 Results and Comparison to Prior Work

In both dual-core and quad-core scenarios, we evaluated HALLS's effectiveness by analyzing the L2 cache's energy as achieved by a HALLS cache compared to the SRAM and DRS with the base configuration shown in Table 2. We used a retention time of $10ms$ for DRS, since it was considered the average best in prior work [9]. Prior work has shown that an SRAM LLC can consume up to 24% of a processor's power [27]; thus, replacing SRAM with STT-RAM can substantially improve the energy efficiency, especially in resource-constrained systems. In general, due to a substantial reduction in leakage power, both HALLS and DRS significantly reduced the total energy as compared to the SRAM. Apart from adapting the retention time to applications' runtime needs, HALLS also provides the important advantage of enabling a right-provisioned cache for executing applications, thereby achieving substantial energy savings as compared to DRS.

5.1 Energy and Latency Analysis

5.1.1 Dual-core system

Fig. 7a and 7b depict the energy and latency of HALLS and DRS normalized to the SRAM in a dual-core system. On average across all the workloads, HALLS reduced the energy by 60.53% and 50.28% compared to SRAM and DRS, respectively. We observed substantial energy reductions specifically for $workload3$, $workload4$ and $workload10$, which contributed over 70% and 65% energy savings from SRAM and DRS, respectively. These workloads include $gobmk$, $namd$, $hmmer$, $bzip2$, and $sjeng$, which are compute-intensive benchmarks [28] that exhibited short block lifetimes. As such, HALLS allocated data blocks to clusters with smaller retention time (i.e. 100μs and 1ms) and achieved smaller latency and dynamic energy.

Conversely, HALLS performed worst in energy savings for $workload5$ and $workload9$. The energy savings for $worklod5$ were 20.13% and 9.65% compared to SRAM and DRS, respectively. Similarly HALLS decreased the energy for $workload9$ by 12.82% compared to SRAM and *increased* by 9.59% compared to DRS. We attribute these reduced energy savings to the fact that $workload5$ and $workload9$ include write-intensive benchmarks such as $soplex$, $h264ref$, $hmmer$, and $omnetpp$ [9], [11]. These benchmarks increased the dynamic energy due to the write operations, and also incurred additional leakage energy due to the long write latencies.

Fig. 7b compares the latency achieved by HALLS with DRS and SRAM. On average across all the workloads, HALLS *increased* the latency by 1.90% compared to SRAM,

TABLE 2: Cache parameters of SRAM and STT-RAM for the base cache configurations

Cache Configuration	1MB, 64B line size, 16-way				
Memory Device	SRAM	STT-RAM-100μs	STT-RAM-1ms	STT-RAM-10ms	STT-RAM-100ms
Write Energy (per access)	0.338nJ	0.392nJ	0.404nJ	0.419nJ	0.438nJ
Cache Hit Energy (per access)	5.318nJ	5.794nJ	5.794nJ	5.794nJ	5.794nJ
Leakage Power	3234.916mW	2200.032mW			
Hit Latency (cycles)	2	2	2	2	2
Write Latency (cycles)	2	3	4	6	7

TABLE 3: Cache parameters of SRAM and STT-RAM for HALLS cache configuration results

Memory Device		SRAM	STT-RAM-100μs	STT-RAM-1ms	STT-RAM-10ms	STT-RAM-100ms
128K-1W-16B	Write Energy (per access)	0.033nJ	0.033nJ	0.037nJ	0.041nJ	0.047nJ
	Cache Hit Energy (per access)	0.035nJ	0.028nJ			
	Leakage Power	277.744mW	141.139mW	141.282mW	141.425mW	141.568mW
128K-1W-32B	Write Energy (per access)	0.059nJ	0.059nJ	0.066nJ	0.074nJ	0.084nJ
	Cache Hit Energy (per access)	0.061nJ	0.051nJ			
	Leakage Power	288.864mW	186.218mW	186.49mW	186.761mW	187.033mW
128K-2W-32B	Write Energy (per access)	0.058nJ	0.056nJ	0.062nJ	0.07nJ	0.08nJ
	Cache Hit Energy (per access)	0.117nJ	0.092nJ			
	Leakage Power	346.743mW	185.298mW			
128K-1W-64B	Write Energy (per access)	0.112nJ	0.108nJ	0.12nJ	0.135nJ	0.153nJ
	Cache Hit Energy (per access)	0.113nJ	0.09nJ			
	Leakage Power	325.697mW	196.05mW			
128K-4W-64B	Write Energy (per access)	0.130nJ	0.150nJ	0.162nJ	0.177nJ	0.196nJ
	Cache Hit Energy (per access)	0.519nJ	0.519nJ	0.519nJ	0.519nJ	0.520nJ
	Leakage Power	507.852mW	363.607mW			
256K-8W-64B	Write Energy (per access)	0.193nJ	0.212nJ	0.224nJ	0.24nJ	0.258nJ
	Cache Hit Energy (per access)	1.526nJ	1.532nJ			
	Leakage Power	1181.176mW	858.677mW			
512K-16W-64B	Write Energy (per access)	0.309nJ	0.375nJ	0.351nJ	0.367nJ	0.385nJ
	Cache Hit Energy (per access)	4.871nJ	5.577nJ	4.953nJ	4.953nJ	4.953nJ
	Leakage Power	2268.544mW	1880.816mW	1566.491mW	1566.491mW	1566.491mW
1M-1W-32B	Write Energy (per access)	0.179nJ	0.128nJ	0.135nJ	0.143nJ	0.153nJ
	Cache Hit Energy (per access)	0.188nJ	0.12nJ	0.121nJ	0.121nJ	0.122nJ
	Leakage Power	1745.328mW	762.778mW	763.444mW	764.109mW	764.775mW
1M-1W-64B	Write Energy (per access)	0.335nJ	0.244nJ	0.257nJ	0.273nJ	0.291nJ
	Cache Hit Energy (per access)	0.344nJ	0.228nJ	0.229nJ	0.229nJ	0.23nJ
	Leakage Power	1866.193mW	982.701mW	983.986mW	985.271mW	986.555mW
1M-2W-64B	Write Energy (per access)	0.329nJ	0.25nJ	0.263nJ	0.278nJ	0.292nJ
	Cache Hit Energy (per access)	0.663nJ	0.464nJ	0.466nJ	0.467nJ	0.458nJ
	Leakage Power	2276.707mW	1472.512mW	1475.038mW	1477.564mW	1456.263mW
1M-4W-64B	Write Energy (per access)	0.354nJ	0.278nJ	0.29nJ	0.305nJ	0.323nJ
	Cache Hit Energy (per access)	1.415nJ	1.035nJ			
	Leakage Power	3228.278mW	2767.573mW			
1M-8W-64B	Write Energy (per access)	0.285nJ	0.256nJ	0.268nJ	0.283nJ	0.301nJ
	Cache Hit Energy (per access)	2.261nJ	1.841nJ			
	Leakage Power	2839.156mW	1432.367mW			

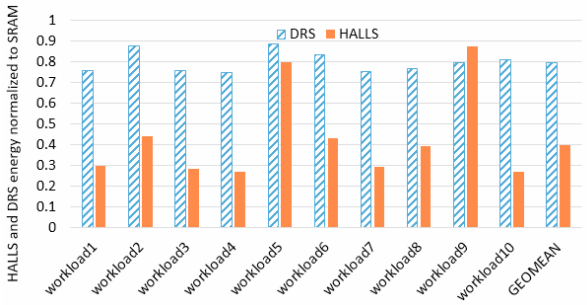
TABLE 4: System configurations

System unit	Experimental setup
CPU	Modeled after ARM Cortex A15 @ 2GHz
L1 SRAM I/D-Cache	32KB, 64 line size, 4-way, LRU, MOESI
L2 STT-RAM Cache	Bank size: 32KB Size: 128KB $\longrightarrow$ 1MB; 2* Line size: 16B $\longrightarrow$ 64B; 2* Associativity: 1-way $\longrightarrow$ 16-way Retention time:100μs, 100ms, 10ms, 1ms Random replacement
Main memory	8GB DRAM

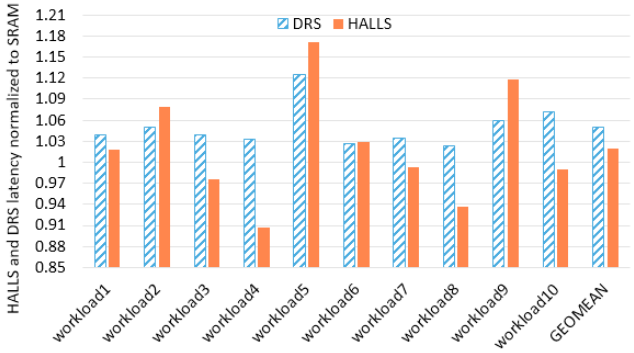
TABLE 5: Experimental workloads

Workload	Dual-core	Quad-core
Workload1	calculix-leslie3d	calculix-h264ref-bzip2-leslie3d
Workload2	omnetpp-xalancbmk	omnetpp-sjeng-gobmk-xalancbmk
Workload3	sjeng-hmmer	sjeng-hmmer-mcf-namd
Workload4	gobmk-bzip2	gobmk-bzip2-leslie3d-soplex
Workload5	soplex-h264ref	soplex-h264ref-hmmer-omnetpp
Workload6	mcf-xalancbmk	sjeng-mcf-calculix-xalancbmk
Workload7	gobmk-h264ref	gobmk-h264ref-calculix-mcf
Workload8	bzip2-soplex	bzip2-soplex-namd-leslie3d
Workload9	hmmer-omnetpp	hmmer-omnetpp-h264ref-xalancbmk
Workload10	namd-hmmer	namd-hmmer-calculix-gobmk

but decreased the latency by 2.94% compared to DRS. HALLS decreased the latency by up to 9.23% and 12.09% compared to SRAM and DRS for $workload4$, which contains $gobmk$ and $bzip2$, both of which are read-intensive applications and exhibit short block lifetimes. Due to the STT-

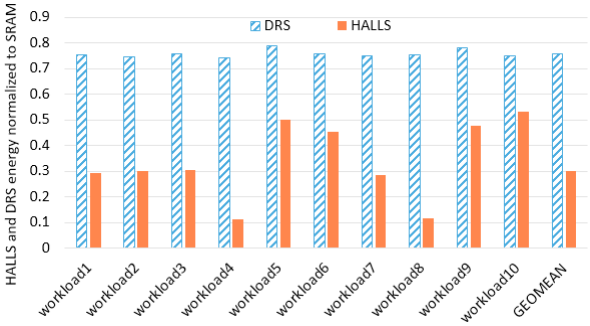


(a) HALLS and DRS energy normalized to SRAM

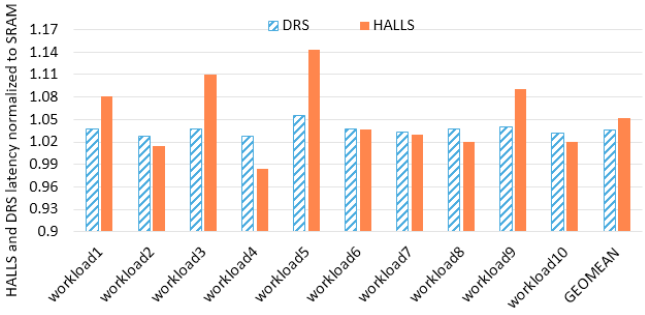


(b) HALLS and DRS latency normalized to SRAM

Fig. 7: HALLS and DRS comparison in energy and latency normalized to SRAM in a dual-core system



(a) HALLS and DRS energy normalized to SRAM



(b) HALLS and DRS latency normalized to SRAM

Fig. 8: HALLS and DRS comparison in energy and latency normalized to SRAM in a quad-core system

RAM's short hit latency and HALLS's ability to adapt the configurations to the executing workloads' requirements, HALLS reduced the latency for five of ten workloads compared to SRAM. However, for $workload5$—a write-intensive workload—HALLS increased the latency by up to 17.10% and 4.07% compared to SRAM and DRS, representing a substantial latency tradeoff in favor of the aforementioned energy savings.

5.1.2 Quad-core system

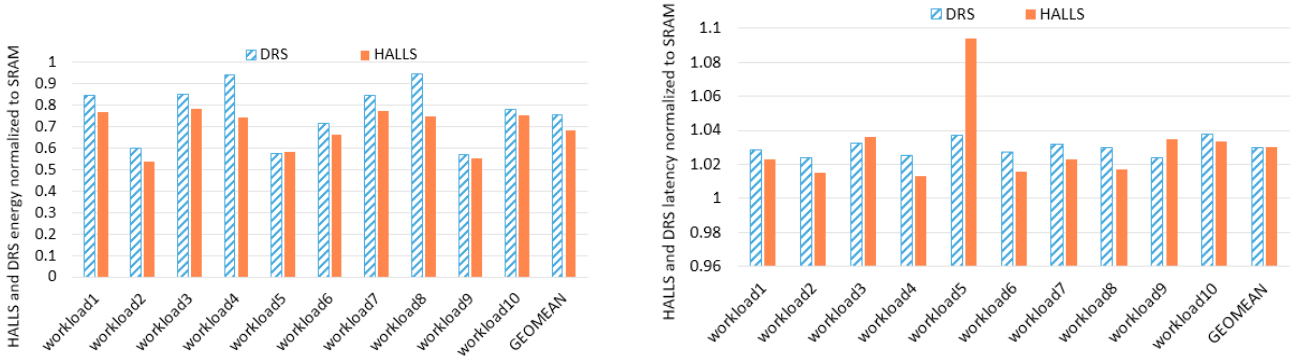
Fig. 8a and 8b depict the energy and latency of both HALLS and DRS normalized to SRAM in a quad-core system. On average across the workloads, Fig. 8a shows that HALLS reduced the average energy by 60.57% and 70.12%, as compared to DRS and SRAM, respectively. Compared to DRS, energy savings were over 85% for $workload4$ and $workload8$. This improvement was possible due to the short block lifetimes of the benchmarks featured in this workload. Both of these workloads featured $bzip2$ and $leslie3d$ (Table 5), both of which featured blocks that exhibited short lifetimes. Unlike most other benchmarks, both $bzip2$ and $leslie3d$ also had low increase in miss rates for low retention times compared to the higher retention times. As such, HALLS was able to use a short retention time, while also adapting the cache configurations to the benchmarks' requirements. In most cases, HALLS's adaptability reduced the energy by more than 50%. These results illustrate HALLS's ability to adapt STT-RAM configurations to the variety of execution requirements exhibited by applications in multicore systems.

We note that HALLS's substantial energy reduction was at the expense of some latency overhead. Fig. 8b shows that HALLS increased the latency by 5.16% and 1.47%, as compared to SRAM and DRS, respectively. Latency overheads were up to 14.25% for $workload5$ as compared to SRAM. These latency overheads occurred because HALLS resulted in additional misses for several data blocks whose lifetimes exceeded the available retention time. We also observed that another important factor that caused the latency overhead was the workloads' reactions to the STT-RAM's long write latency characteristic. We observed that the most latency degradation occurred with workloads that featured write-intensive applications. For instance, $workload5$ contained $soplex$, $h264ref$, and $omnetpp$, which are characterized as write-intensive benchmarks [9], [11]. Similarly, $workload3$ also featured three write-intensive benchmarks: $sjeng$, $hmmer$, and mcf.

Compared to DRS, HALLS's latency overheads were down to 6.97% and 8.22% for $workload3$ and $workload5$, respectively. We reiterate that considering our target of resource-constrained systems, the average 1.47% latency overhead can be considered an acceptable tradeoff for the substantial 60.57% energy savings compared to DRS.

5.1.3 Summary of HALLS's latency behaviors

We observed that workloads exhibited similar latency behavior in HALLS for both dual-core and quad-core systems.



(a) HALLS and DRS energy normalized to SRAM

(b) HALLS and DRS latency normalized to SRAM

Fig. 9: Energy and latency comparison of HALLS, DRS, and SRAM with adaptable cache configurations in the quad-core system. To illustrate the benefits of retention time adaptability, we assume adaptable cache configurations for all three techniques, but HALLS features adaptable retention time in addition.

We observed a strong correlation between HALLS's performance (compared to SRAM and DRS) and the write intensities and cache block lifetimes of the benchmarks featured in executing workloads. To illustrate these observations, Table 6 depicts how HALLS's average latency compared with SRAM and DRS for workloads featuring read vs write intensive benchmarks, and short vs. long block lifetimes. We defined a benchmark as having short block lifetimes if the benchmark's data blocks were not required in the cache (i.e., time between successive references) beyond 1ms on average, while benchmarks with long block lifetimes were required in the cache for more than 1ms.

While SRAM outperformed HALLS for write-intensive workloads, SRAM's superiority over HALLS for latency was less visible for workloads that had short block lifetimes. This behavior is exemplified in the dual-core system by $workload5$ and $workload9$ (Table 5). All four benchmarks in these two workloads are write-intensive benchmarks. However, $hmmer$ (in $workload9$) exhibits shorter cache block lifetimes than $omnetpp, soplex$, and $h264ref$. Thus, compared to SRAM, HALLS increased the latency for $workload9$ by a smaller amount (11.83%) than $workload5$ (17.10%).

We also observed this behavior between the dual- and quad-core systems. For $workload5$, for example, HALLS increased the latency compared to SRAM by 17.10% and 14.25% in the dual- and quad-core systems, respectively. In the dual-core system, $workload5$ comprises of two write-intensive benchmarks ($soplex$ and $h264ref$) with long block lifetimes. The introduction of $hmmer$, which has short block lifetimes, to the mix for the quad-core system caused a latency reduction, even though the other three benchmarks had longer cache block lifetimes. We also observed that HALLS performed best with respect to latency for read-intensive benchmarks with shorter cache block lifetime. For instance, workloads featuring $bzip2$, $gobmk$, or $xalancbmk$ ($workload2, 4, 6, 7, 8$) exhibited small latency overheads in both the dual-core (1.28% on average) and quad-core systems (1.66% on average) (Fig. 7b and 8b). As shown in Table 2, in the same cache configuration, write latency in STT-RAM is generally higher than SRAM, and the latency grows as the retention time increases. As such, read-intensive benchmarks that issue fewer write requests would suffer less latency overheads as compared to SRAM. If the benchmark also exhibits shorter cache block lifetime, HALLS can adapt to a shorter retention time requirement without substantial overheads from cache misses, thereby taking advantage of shorter write latency per access.

5.1.4 Benefits of retention time adaptability

To explore the benefits of exclusively adapting the retention time using our approach in a shared L2 cache, we also implemented and analyzed DRS and SRAM with adaptable cache configurations as determined by the HALLS configuration tuning algorithm. That is, DRS's retention time was kept constant, while its (and SRAM's) cache configurations were adapted to the different applications' requirements similar to HALLS.

Fig. 9a depicts the energy and latency comparison for HALLS and DRS, normalized to SRAM, given adaptable cache configurations for all three techniques. On average across all the workloads, HALLS reduced the energy by 31.83% and 9.34% as compared to SRAM and DRS, respectively. We observed energy savings (compared to DRS) as high as 21.01% and 20.84% for $workload4$ and $workload8$, respectively. As described earlier, we attribute these energy savings to the benchmarks' short block lifetimes. HALLS took advantage of the energy benefits of smaller retention times that more closely match the applications' needs.

With the same configuration across HALLS, SRAM, and DRS, HALLS (with variable retention times) incurred smaller latency overhead than when compared to static SRAM and DRS configurations. As shown in Fig. 9b, on average, HALLS increased the latency by 3.02% as compared

TABLE 6: Summary of latency behaviors in HALLS

Benchmarks	Intensity	Cache block lifetime	HALLS vs SRAM	HALLS vs DRS
hmmer,leslie3d, sjeng	write	short	+4.29%	+0.45%
soplex,h264ref, mcf,omnetpp	write	long	+13.03%	+5.67%
bzip2,xalancbmk, gobmk	read	short	-5.44%	-8.18%
calculix,namd	read	long	+2.68%	-1.67%

to SRAM, and exhibited nearly the same latency on average (with a marginal 0.03% improvement) as compared to DRS. Compared to DRS, HALLS marginally *decreased* the latency for seven out of ten workloads with the highest reduction at 1.28% for $workload8$. The highest latency overhead was 5.48% for $workload5$. Apart from $workload5$, other workloads' latency overheads were below 1%.

5.2 HALLS Overheads:

HALLS overheads result from: 1) hardware overhead: the tuner—including the datapath for energy estimation—and the counter (Section 3.2); 2) time overhead, including the runtime tuning overhead, which includes the time it takes to determine the best configuration, and context switching overheads when swapping cache data during reconfiguration.

We estimated the tuner overhead using Verilog and analyzed using Synopsys Design Compiler. The estimated area overhead was 0.0145 mm^2, and the dynamic and leakage power were 28.04 mW and 422.68 μW, respectively. Compared to an ARM Cortex-A15 processor [2], the tuner's overhead is negligible (approximately 0.095%). The counter required 4 bits per 64B block, resulting in a 0.78% overhead. We assume the counter is stored in the STT-RAM along with other meta-data (e.g., tags), in order to further reduce area and power overheads.

We also evaluated the number of tuning intervals required to determine the best configuration for the different applications. Overall, the highest number of intervals required was seven to determine the best cache configuration. Retention time tuning took a constant of four tuning intervals for all applications, since all the retention times were sampled. Given our tuning interval of 10M instructions, the tuning overheads amortize rapidly over the rest of the application's execution. In the worse case, context switching incurred a latency of 114688 cycles and an energy overhead of 14.844 μJ.

6 Conclusions and Future Research

In this paper, we propose a *highly adaptable last level STT-RAM cache (HALLS)* as a viable option for mitigating the overheads of implementing the STT-RAM in last level caches (LLC). HALLS allows the LLC's configurations to be dynamically adapted to executing applications' cache configuration and retention time requirements. We designed HALLS as a 1MB L2 cache organized as 32 physical banks. The 32 banks are organized in 8-bank clusters, with each cluster featuring a different retention time. During runtime, data blocks are placed in the physical banks that best suits the applications' retention time requirements. Furthermore, the cache configuration can be adapted to suit executing applications' needs. Experiments reveal that in a quad-core system, HALLS reduced the average energy by 60.57% compared to prior work, while introducing 1.47% of latency overhead. For future research, we plan to explore the design space of retention times for multithreaded applications and also explore techniques for reducing runtime overheads using history-based prediction of the best cache configurations and retention times.

References

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[2] 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

[3] S. A. Wolf, J. Lu, M. R. Stan, E. Chen, and D. M. Treger, "The promise of nanomagnetics and spintronics for future logic and universal memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2155–2168, Dec 2010.

[4] 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

[5] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi, "Spin-transfer torque magnetic random access memory (STT-MRAM)," *J. Emerg. Technol. Comput. Syst.*, vol. 9, no. 2, pp. 13:1–13:35, May 2013. [Online]. Available: http://doi.acm.org/10.1145/2463585.2463589

[6] 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.

[7] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A novel architecture of the 3d stacked MRAM l2 cache for CMPs," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, Feb 2009, pp. 239–249.

[8] 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.

[9] 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.

[10] 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.

[11] J. Ahn, S. Yoo, and K. Choi, "DASCA: dead write prediction assisted STT-RAM cache architecture," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 25–36.

[12] T. Adegbija, "Exploring configurable non-volatile memory-based caches for energy-efficient embedded systems," in *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, May 2016, pp. 157–162.

[13] Z. Wang, D. A. Jimnez, C. Xu, G. Sun, and Y. Xie, "Adaptive placement and migration policy for an STT-RAM-based hybrid cache," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 13–24.

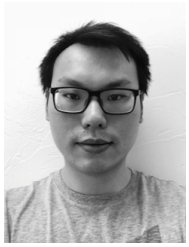
[14] S. Cho and H. Lee, "Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2009, pp. 347–357.

[15] Q. Zeng and J. K. Peir, "Content-aware non-volatile cache replacement," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 92–101.

[16] 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.

[17] 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

[18] M. Imani, S. Patil, and T. Rosing, "Low power data-aware STT-RAM based hybrid cache architecture," in *2016 17th International Symposium on Quality Electronic Design (ISQED)*, March 2016, pp. 88–94.
[19] Y. Chen, J. Cong, H. Huang, B. Liu, C. Liu, M. Potkonjak, and G. Reinman, "Dynamically reconfigurable hybrid cache: An energy-efficient last-level cache design," in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2012, pp. 45–50.
[20] K. Kuan and T. Adegbija, "LARS: logically adaptable retention time stt-ram cache for embedded systems," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 461–466.
[21] "Qualcomm Announces Snapdragon 845 Mobile Platform: Tocks Next-Gen CPU Cores, GPU, AI, & More." [Online]. Available: https://www.anandtech.com/show/12114/qualcomm-announces-snapdragon-845-soc
[22] "Cortex-A15 Technical Reference Manual." [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0438c/CHDDDHFD.html
[23] C. Zhang, F. Vahid, and W. Najjar, "A highly configurable cache for low energy embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 2, pp. 363–387, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1067915.1067921
[24] 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
[25] 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.
[26] 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
[27] 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.
[28] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 412–423. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250713



Kyle Kuan (M'16) is a Ph.D. student in the Department of Electrical and Computer Engineering at the University of Arizona. He received his M.S. in Electrical Engineering from National Taiwan University in 2008 and B.S. in Mechanical Engineering from National Chiao Tung University in 2006. His research interests include cache design for energy efficient systems, non-volatile memories, and right-provisioned micro architectures for IoT devices.

Tosiron Adegbija (M'11) received his M.S and Ph.D in Electrical and Computer Engineering from the University of Florida in 2011 and 2015, respectively and his B.Eng in Electrical Engineering from the University of Ilorin, Nigeria in 2005.

He is currently an Assistant Professor of Electrical and Computer Engineering at the University of Arizona, USA. His research interests are in computer architecture, with emphasis on adaptable computing, low-power embedded systems design and optimization methodologies, and microprocessor optimizations for the Internet of Things (IoT).

Dr. Adegbija was a recipient of the CAREER Award from the National Science Foundation in 2019 and the Best Paper Award at the Ph.D forum of IEEE Computer Society Annual Symposium on VLSI (ISVLSI) in 2014.