SCART: Predicting STT-RAM Cache Retention Times Using Machine Learning

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

***Abstract*—Prior studies have shown that the retention time of the non-volatile spin-transfer torque RAM (STT-RAM) can be relaxed in order to reduce STT-RAM's write energy and latency. However, since different applications may require different retention times, STT-RAM retention times must be critically explored to satisfy various applications' needs. This process can be challenging due to exploration overhead, and exacerbated by the fact that STT-RAM caches are emerging and are not readily available for design time exploration. This paper explores using known and easily obtainable statistics (e.g., SRAM statistics) to predict the appropriate STT-RAM retention times, in order to minimize exploration overhead. We propose an STT-RAM Cache Retention Time (SCART) model, which utilizes machine learning to enable design time or runtime prediction of right-provisioned STT-RAM retention times for latency or energy optimization. Experimental results show that, on average, SCART can reduce the latency and energy by 20.34% and 29.12%, respectively, compared to a homogeneous retention time while reducing the exploration overheads by 52.58% compared to prior work.**

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

I. Introduction

Spin-transfer torque RAM (STT-RAM) has emerged as a popular alternative to SRAM for implementing caches. STT-RAMs offer several benefits, such as high density, low leakage power, compatibility with CMOS, high endurance, etc. However, STT-RAMs suffer from high write latency and write energy, which may impede their widespread adoption in state-of-the-art resource-constrained systems. A promising optimization involves relaxing STT-RAM's *retention time*—the duration for which data is retained in the absence of power—from the intrinsic duration, which could be up to 10 years [1]. Reducing the retention time offers much promise for latency and energy improvements because the long write latency and high write dynamic energy directly result from the long retention times of a non-volatile STT-RAM [1]. Thus, prior works [2], [1], [3], [4] have studied the benefits of reducing/relaxing the retention times, especially in caches since cache data blocks are usually only needed in the cache for short periods of time (typically less than 1 second).

Given a relaxed retention STT-RAM cache (hereafter referred to simply as STT-RAM cache), prior work has shown that different applications may require different retention times. An application's retention time requirements are dictated by its *cache block lifetimes*, i.e., how long the blocks must remain in the cache. To yield maximal benefits from STT-RAM caches, the retention time must be specialized to the needs of the executing applications or application domains. If the retention times are not specialized, they may be over-provisioned, thus wasting energy/latency, or under-provisioned, thus requiring additional schemes (e.g., the dynamic refresh scheme [3]) to maintain data integrity after the retention time elapses. Both cases accrue overheads that may substantially limit optimization potential [4, 2].

To enable right-provisioned retention times for STT-RAM caches, the retention times must be critically explored for different applications and metrics (e.g., energy, latency). An exhaustive exploration of retention times is a challenging task, given that a wide variety of applications, application characteristics (e.g., read/write behaviors, cache block characteristics), and objective functions (e.g., energy, latency, energy delay product, user experience) must be considered. Furthermore, in systems with adaptable retention times, such as the logically adaptable retention STT-RAM (LARS) cache proposed in [2], an exhaustive exploration can incur substantial runtime overheads, including hardware, switching, time, and energy, especially in complex systems.

In this paper, we propose an approach—*STT-RAM* ***Ca****che* ***R****etention* ***T****ime (SCART) Model*—that utilizes machine learning to predict right-provisioned retention times for a variety of systems, applications, and metrics. Since SRAM caches are widely available and accessible to researchers and designers, whereas STT-RAM caches are still nascent, we explore using SRAM characteristics that can easily be obtained via simulations as input labels to enable the prediction of right-provisioned retention times for STT-RAM caches for target applications or application domains. During runtime in a system with multiple retention time units (e.g., [2]), based on execution statistics from one cache unit (SRAM, in a hybrid design [5] or STT-RAM), our approach can directly predict the best unit on which to run the application, without the need for overhead-prone design space exploration.

Our contributions are summarized as follows:

- We show, for the first time (to our knowledge), that right-provisioned retention times for STT-RAM caches can be

978-1-7281-5416-9/19/$31.00 ©2019 IEEE

predicted using easily obtainable SRAM characteristics.
- We compare several machine learning classifiers, and propose a machine learning-based model (*SCART*) that enables fast runtime retention time prediction. SCART can be implemented with low overhead for runtime prediction in a system with multiple retention times or in a hybrid system.
- Using extensive simulations with three benchmark suites (SPEC CPU2006 [6], MiBench [7], and GAP [8]), to represent different kinds of applications, we show that our model reduces exploration time by 52.58%. Furthermore, in a runtime implementation, our approach achieves average latency and energy savings of 20.34% and 29.12%, respectively, compared to a homogeneous system.

II. Related Work

The STT-RAM bit cell's basic structure comprises of a transistor and a magnetic tunnel junction (MTJ). STT-RAM's characteristics and operations of the STT-RAM have been discussed in the prior work [9]. Smullen et al. [1] showed that for implementation in caches, STT-RAM's retention time can be substantially reduced (e.g., by reducing the planar area) in order to mitigate the attendant write latency and energy overheads of non-volatile STT-RAMs. In this section, we summarize a few related prior works that leverage reduced retention STT-RAMs and briefly overview prior work on cross-architectural prediction to motivate our work.

A. *Multi-retention and Hybrid STT-RAM Caches*

Sun et.al. [3] proposed to use a hybrid STT-RAM L2 cache with multiple retention times in order to more closely match the needs of executing applications. The authors used a coarse-grained approach, featuring a long retention time for read-intensive applications and a short retention time for write-intensive applications. Cache blocks that needed to remain in the cache beyond the retention time were refreshed via a DRAM-style dynamic refresh scheme to maintain data correctness. To reduce the overheads introduced by the need to refresh cache blocks, Kuan et.al [2] further analyzed application cache block characteristics and showed that the refresh overheads could be mitigated by more closely matching the applications' runtime execution requirements. The authors proposed a logically adaptable retention STT-RAM (LARS) L1 cache featuring multiple retention time units, and used a sampling-based algorithm to dynamically determine applications' right-provisioned retention times.

Since STT-RAM is generally more prone to overheads when running write-intensive applications, due to the high write latency, hybrid (SRAM+STT-RAM) caches have been proposed. To minimize overheads, the STT-RAM is used to run read-intensive workloads and the SRAM is used for write-intensive workloads. While multiple hybrid (SRAM+STT-RAM) caches [5] have been proposed, they typically only feature a single retention time. We anticipate that hybrid caches featuring multiple retention times will be explored in the near future. In all these systems, an important existing challenge, which our work addresses, is how to rapidly explore the right-provisioned retention times with which to design the systems, or how to rapidly select the best retention time during runtime, in order to maximize the energy or latency benefits of reduced retention STT-RAM caches.

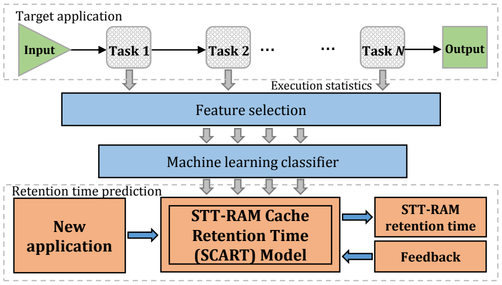


Fig. 1: High-level overview of predictive model

B. *Cross-Architectural Prediction*

The work proposed herein is along the lines of prior work where a known architecture is used to predict the behavior of an unknown architecture. For instance, Ardalani et.al. [10] presented cross-architecture performance prediction using CPU implementation to predict the performance of GPUs. Yang et.al. [11] presented techniques for predicting the performance of parallel applications using partial execution. Guo et.al. [12] presented a model to provide inter-architecture performance prediction for sparse matrix vector multiplication to help researchers choose the appropriate GPU architecture for the application. Similarly, Zheng et.al. [13] presented a phase level cross-platform prediction for performance and power for CPU architectures. These works are orthogonal to ours, but illustrate the viability of the approach proposed herein.

III. STT-RAM Cache Retention Time Prediction (SCART)

Unlike SRAM caches, where easily observable statistics from performance counters (e.g., cache miss rates) can be used to directly determine the best cache configurations, the correlations between miss rates and retention times are not that direct in STT-RAM caches. Therefore, in this work, we focus on using machine learning to predict the best retention times for STT-RAM L1 data cache energy and latency minimization based on hardware performance statistics. We chose to focus on the data cache since our experiments showed that the instruction cache blocks exhibit low variability in the retention time needs of the considered applications. A static retention time of $10ms$ sufficed for the applications considered.

SCART incorporates a low-overhead machine learning classifier for design time or runtime fast and accurate prediction of retention times. For a design time exploration scenario, we assume that the target applications are first profiled on an SRAM cache with any arbitrary configurations. These statistics can be obtained via simulators (e.g., GEM5 [14]) or by running the application on an actual computer. The execution statistics are then provided as input labels to SCART, which then outputs the best STT-RAM retention time for the target applications and specified objective function. This scenario is suitable for designing STT-RAM caches for an application-specific processor or provisioning a processor with a range of retention times in order to satisfy a variety of runtime retention time requirements [3, 2]. For a runtime scenario, the application can be run for a brief interval on one cache unit, and SCART uses the execution statistics to directly predict the best unit on which to run the rest of the application. SCART will substantially reduce the runtime complexity and migration costs for three system scenarios: 1) Multi-retention time cache designs (similar to [2]) for which the best cache unit must be determined during runtime; 2) hybrid caches to determine which unit to execute the application on; and 3) a multi-core system with a combination of SRAM and/or heterogeneous retention time STT-RAM caches [15].

A. SCART Model Architecture

Figure 1 presents a high level overview of our machine learning-based model. We model executing applications as task graphs, wherein each task may have one or more implementations, called *task options* (e.g., different algorithmic implementations). These tasks are equivalent to application phases in our work. The different tasks and task options may have different execution characteristics, which also affect the target objective functions (energy or latency). Furthermore, each task may have different data configurations (e.g., data size, bit-width, etc.) that may change based on the inputs.

The training data points are composed of execution statistics obtained from hardware performance counters. To generate the training data, we used GEM5 to gather the execution statistics of the different phases of a random subset of SPEC 2006, MiBench, and GAP benchmarks. We observed that 1 million instructions was sufficient to obtain stable statistics for predicting full phase behaviors. Thus, we used an interval size of 1 million instructions. As such, our model can predict retention times after executing an application or application phase for only 1 million instructions.

Based on the SRAM characteristics of the training data, we performed feature selection to determine the most relevant features (i.e., hardware characteristics) for the STT-RAM retention time. We explored 59 features[1] based on SRAM performance characteristics. These features can be either directly obtained from hardware performance counters or calculated from performance counter statistics. Some of the most important features included L1 and L2 cache miss rates, number of branches, cache read and write statistics while some less important features included the DRAM read and write bursts, number of integer and floating point instructions etc.

To enable extensive testing, our initial training label size was 256 and the test label size was 64 (representing all the application phases). Our training label also consisted of six retention times: $10\mu s, 26.5\mu s, 50\mu s, 75\mu s, 100\mu s,$ and $1ms$. We empirically found that longer retention times were not beneficial for any of the considered applications. Given the selected features, we then fed the labels into a machine learning classifier (Section III-B) to develop SCART for predicting the best retention time for a new application.

To prevent substantial energy or latency degradation in runtime execution, the model also features a feedback mechanism that monitors the statistics of the predicted retention time. If the predicted retention time degrades the energy or latency compared to the base, the configuration is reverted to the base. To prevent data corruption resulting from the reduced retention time, we incorporate a low-overhead *monitor counter*, similar to prior work [2, 3], to keep track of each cache block's lifetime and invalidate the block (or write back to lower level memory if dirty) before the retention time expires. The counter can be implemented as an N-state finite state machine, which begins at the initial state when a block is written into the cache, counts up until the retention time is about to expire, and raises a flag to evict the block or write back to a lower memory level. We assumed $N = 4$ in our work, resulting in a hardware overhead of only two bits per block.

B. Machine Learning Classifier Comparison and Selection

SCART features a machine learning classifier that comprises of two stages: the *training stage* and the *prediction stage*. In the training stage, the model learns the patterns in the input data (benchmarks and execution characteristics) and their correlations to the different retention time labels. In the prediction stage, the model takes as input new benchmarks and their characteristics, and outputs the predicted retention time labels for the new benchmarks.

To select the best classifier, we considered several different classifiers and evaluated their accuracy. The classifiers we explored included: *linear SVC, radial basis function SVC, decision tree, random forest classifiers, decision trees-based bagging, adaptive boosting, gradient adaptive boosting [16], extra-tree classifiers based ensemble technique [17],* and *K-nearest neighbor (KNN) classifiers [18].* For brevity, we omit detailed descriptions of these classifiers, since they are described in prior work.

Table I presents the different classifiers' F-scores [19]. The F-score is an evaluation metric that considers both precision and recall, and is a measure of a classifier's accuracy. The classifiers with the highest F-score were KNN and extra trees. However, we chose KNN classifier for use in our model due to its simplicity and lower prediction time (which makes it suitable for runtime predictions). Furthermore, KNN offers other advantages, such as lack of generalization (resulting in rapid training), and its non-parametric qualities. That is, KNN does not make any assumptions on the underlying data distribution. Thus, our model is amenable to applications that may not obey the typical theoretical assumptions (e.g., Gaussian mixtures, linearly separable, etc.). In general, KNN operates based on feature similarity; it determines how closely out-of-sample features resemble a training set, and classifies a given data point based on the similarity. Additional low level details of the KNN classifier can be found in [18].

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

TABLE I: Classifier F-scores

	Linear SVC	Decision Tree	Extra Trees	Random Forest	KNN	RBF-SVC	Bagging	Adaboost	Gradient Boost
F-score (0-1)	0.54713	0.70989	0.78098	0.73437	0.78203	0.66875	0.75625	0.67552	0.76692

C. *KNN Classifier Tuning*

We observed that predicting the best retention times for latency vs. energy required different sets of features and KNN classifier characteristics. This observation was due to the conflicting nature of latency and energy with respect to retention time requirements. Thus, we tuned the KNN classifier and number of features to enable high accuracy for predicting retention times for latency or energy settings. Furthermore, to ascertain the robustness of our model, we randomly shuffled the data and performed five-fold cross-validation to ensure the validity of the classifier for a wide variety of applications.

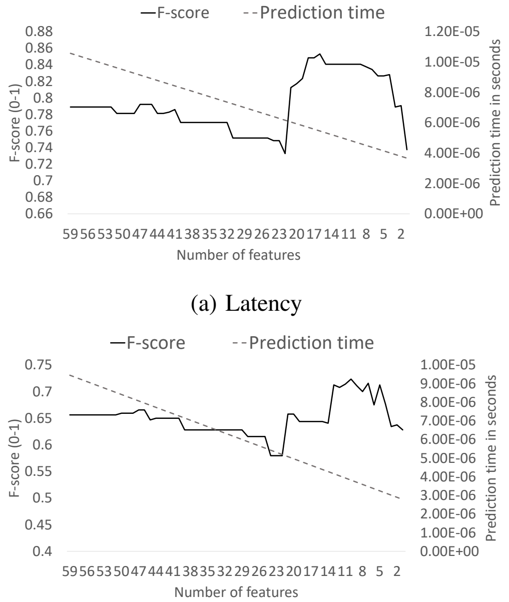
We empirically determined that the KNN classifier with three nearest neighbors and uniform weights achieved the highest F-score for latency and energy optimization. To select the appropriate features, we determined the features' importance values (that is, the features' impacts on prediction accuracy) using the *scikit-learn* tools [20], ordered the features in order of their importance, and eliminated the least important features for both latency and energy.

Figure 2 illustrates our selection of the optimal number of features for energy and latency. We compared the F-score and prediction time while iteratively eliminating the least important or redundant features in every run. The goal of iteratively eliminating the least important features was to find the optimal number of features that enabled the classifier to achieve the highest F-score. That is, we selected the fewest number of features, while eliminating features that did not change the F-score, since fewer features also reduce the prediction time.

From Figure 2a, we observe that the highest F-score for latency optimization was obtained using 9 to 15 features. Thus, we used 9 features for latency in order to achieve a fast prediction time. For energy, as depicted in Figure 2b, 10 features achieved the highest F-score. Therefore, for both latency and energy, we eliminated the least important features until 9 and 10 features, respectively, remained. We also observed from our experiments that even though the highest accuracy was approximately 75%, the false predictions still resulted in near-optimal retention times. As a result, SCART was able to achieve substantial latency and energy savings despite the error rate (Section V-B).

IV. Experimental Setup

We modified the GEM5 simulator [14] to model accurate STT-RAM behavior for different retention times and to capture the L1 and L2 cache statistics. We simulated single and quad-core processors with configurations similar to the ARM Cortex A-15 processor, with a 2GHz clock frequency. Each core had private instruction and data STT-RAM L1 caches, and a shared SRAM L2 cache (in the quad-core processor). Table II depicts the cache parameters for both the SRAM and STT-RAM caches.



(a) Latency

(b) Energy

Fig. 2: Selection of optimal number of features for latency and energy optimization. Tuning began with 59 features, and features were iteratively removed to maximize F-score and minimize prediction time.

To represent a variety of workloads, we used 34 benchmarks in total (for both training and testing—see Section III-A); 22 from SPEC CPU2006 [6] (high performance benchmarks), 6 from MiBench [7] (embedded systems benchmarks) and 6 from GAP [8] (graph algorithms). We ran simulations for a maximum of one billion instructions for all the benchmarks, using the *reference* and *large* input sets for SPEC and MiBench, respectively, and 2048 nodes for the GAP benchmarks. We used Simpoint [21] to obtain the program phases for all the benchmarks, with intervals of 1 million instructions. We used execution statistics gathered after 1 million instructions for prediction.

For a thorough analysis, we initially considered nine retention times: 10μs, 26.5μs, 50μs, 75μs, 100μs, 1ms, 10ms, 100ms, and 1s. However, we found that the best latency or energy retention times for different applications were, for the most part, in the range of 10μs to 1ms. Thus, we eliminated 10ms to 1s from our modeling and analysis. To model the different retention times, we used the MTJ modeling technique proposed in [9] to compute the write pulse, write current and MTJ resistance value R$_{AP}$. We then applied the values to NVSim [22] and integrated with statistics obtained from GEM5 [14] to calculate the cache latency and energy. To model the SRAM cache in the hybrid cache, we used NVSim's SRAM settings. Table II shows different latency and energy specifications for SRAM and STT-RAM used in our

TABLE II: SRAM and STT-RAM cache parameters

L1 cache configuration	32KB, 64B line size, 4-way						
L2 cache configuration	1MB SRAM, 64B line size, 16-way						
Memory device	SRAM	STT-RAM					
Retention times	–	10μs	26.5μs	50μs	75μs	100μs	1ms
Hit latency	0.486ns	0.464ns	0.454ns	0.448ns	0.445ns	0.443ns	0.438ns
Write latency	0.350ns	0.601ns	0.769ns	0.894ns	0.981ns	1.045ns	1.647ns
Read energy (per access)	0.0076nJ	0.003nJ	0.003nJ	0.003nJ	0.003nJ	0.003nJ	0.003nJ
Write energy (per access)	0.0066nJ	0.026nJ	0.030nJ	0.033nJ	0.035nJ	0.036nJ	0.051nJ
Leakage power	34.265mW	4.659mW					

experiments. For stringent comparison, we used a hit cycle of 1 for both SRAM and STT-RAM, unlike prior work that used higher hit cycles for SRAM (e.g., [3]), thus resulting in lower optimization compared to SRAMs. To implement the machine learning algorithms, we used Python's *scikit learn (Sklearn)* library [20].

V. RESULTS

In this section, we first evaluate SCART in the context of a single-core processor, in comparison to a base retention time and exhaustive search. Thereafter, we evaluate SCART in the context of a quad-core processor running multi-programmed workloads, and finally compare SCART to prior work.

A. Comparison to the Base Retention Time

To evaluate SCART's effectiveness, we compared the latency and energy savings achieved by our model with a base retention time. We selected the base retention time as $1ms$ to be conservatively large enough to satisfy the cache block lifetimes of the considered applications, in order to prevent the need to refresh any blocks. Thus, the base configuration eliminates the additional overheads from refreshing data blocks [3]. For each benchmark, we report the overall results as the weighted combination of the phase results, as is the common practice in phase-based optimization [21].

Figure 3 depicts the latency and energy improvements achieved using SCART as compared to the base. On average across all the benchmarks, SCART improved the latency by 20.34%, with improvements of up to 35.19% for bfs (breadth-first search algorithm). We observed different trends for different benchmark suites. For instance, SCART achieved substantial improvements over the base for the GAP benchmarks, since the base retention time was over-provisioned for the benchmarks. Most of the cache blocks needed to remain in the cache for much less than $1ms$. On the other hand, SCART did not achieve substantial latency improvements for some SPEC and MiBench benchmarks, such as $patricia$, for which there was no improvement, and $hmmer$, for which SCART reverted to the base retention time in order to prevent a latency degradation. For $patricia$, the base $1ms$ retention time was sufficient for its cache block lifetimes, while $hmmer$'s cache blocks required more than $1ms$ retention time to prevent premature eviction. A closer look at $hmmer$'s cache blocks revealed that while several of the blocks required less than $1ms$, there were also several blocks that required closer to $10ms$ to prevent premature expiry. However, using a $10ms$ base retention time would have incurred overall overheads for our mix of benchmarks.

Similar to latency, SCART improved the energy, compared to the base, by an average of 29.12%, with savings of up to 34.54% for $libquantum$. The energy trends varied for the different benchmark suites, and we also observed that the retention time that was best for energy was not necessarily best for latency. For example, when SCART was set to optimize for energy, there was a latency overhead of 15.45%; when it was set to optimize for latency, there was an energy overhead of 10.81%. For a few benchmarks (e.g., $hmmer$), however, similar retention times sufficed for both latency and energy optimization. In general, SCART was able to trade off the non-optimized metric for the specified metric, as necessary.

B. Comparison to Exhaustive Search

To further evaluate SCART, we compared the results obtained to exhaustive search of the retention time design space. Note that while the retention time design space will typically not be expansive (six options, in our case), the design time overhead from exhaustive search comes into play when several applications or application domains must be explored. Thus, SCART must be able to rapidly determine retention times that are close to the optimal.

Figure 4 depicts the comparison of the latency and energy achieved by SCART and exhaustive search (i.e., optimal) to the base. For brevity, we only show the geometric means for each benchmark suite considered. As seen in the figure, SCART's results were very close to exhaustive search for the different benchmark suites. For the GAP benchmarks, using SPEC benchmarks as training data, SCART achieved identical savings to exhaustive search for latency, and achieved energy savings within 0.07% of the optimal. Similary, using SPEC benchmarks as training data for the MiBench workloads, SCART achieved latency and energy savings that were 0.4% and 1.9%, respectively, *less* than exhaustive search. The degradation with respect to exhaustive search resulted from false prediction penalty of the labels. However, the penalty was low, since SCART predicted retention times that were close to the optimal, further illustrating SCART's effectiveness.

To further evaluate SCART's robustness, we also performed experiments to predict the retention times for GAP and SPEC benchmarks using training data from MiBench benchmarks (MiBench → GAP). We indicate the summary of the results for MiBench → GAP and MiBench → SPEC predictions in Figure 4 with an asterix (*). SCART achieved similar results to exhaustive search for MiBench → GAP predictions with average latency and energy savings of 34.71% and 39.11% over the base. However, while MiBench → SPEC yielded average latency and energy improvements of 10.3% and 20.71%, respectively, these results were farther from the optimal by

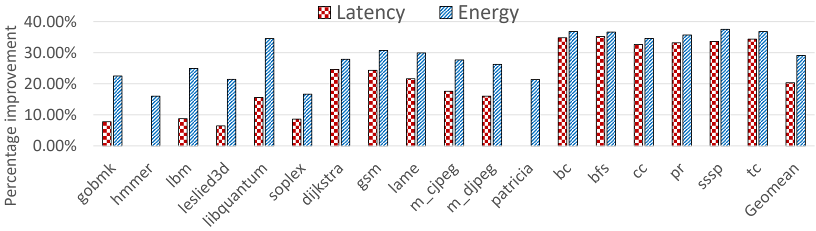


Fig. 3: Percentage latency and energy improvements using SCART model compared to the base retention time of 1ms.

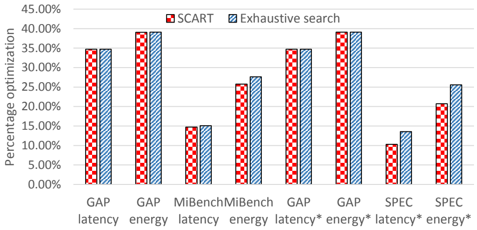


Fig. 4: SCART vs exhaustive search latency and energy improvements compared to the base ($1ms$) retention time. Geometric means of the results are presented.

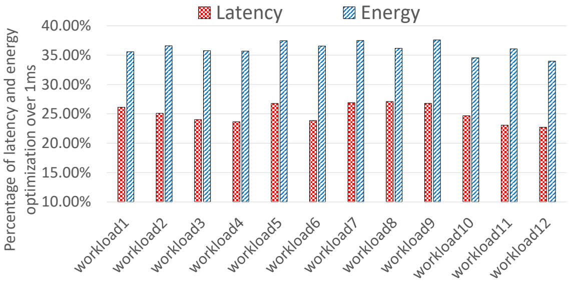


Fig. 5: SCART latency and energy savings in a multi-programmed scenario

3.26% and 4.87%, respectively. We attribute this to the fact that the SPEC benchmarks' labels featured much higher variation than MiBench. As a result, a MiBench → SPEC prediction afforded less coverage in predicted characteristics than the SPEC → MiBench prediction. Overall, these results further illustrate SCART's ability to effectively predict latency and energy-saving retention times.

C. SCART Execution in a Multi-Programmed Scenario

To further evaluate our model, we tested SCART in a multi-programmed execution scenario featuring a quad-core processor with a shared 1MB L2 cache. The experiments performed herein enable us to evaluate SCART's scalability in a more complex system, since resource sharing in the L2 cache can impact the L1 cache behavior of the applications running on each core [23]. We assume that each core features multiple retention time units as in [2], and SCART predicts the best retention time unit for each application on each core.

For the multi-programmed workloads, we created twelve workloads featuring a random combination of four benchmarks per workload, wherein each core runs one benchmark. The workloads used are shown in Table III. For the experiments in this subsection, we used the SPEC benchmarks (66% of the total benchmarks) as training data and MiBench and GAP benchmarks (33%) as testing data.

Figure 5 summarizes the percentage latency and energy optimizations achieved by SCART in the multi-programmed scenario compared to a base retention time of $1ms$. On average across all the workloads, SCART achieved latency and energy savings of 25.07% and 36.13%, respectively. As seen in Figure 5, the latency and energy savings were relatively consistent across the different workloads, demonstrating SCART's effectiveness in various execution scenarios.

D. Comparison to Prior Work and Implementation Overhead

To further evaluate the effectiveness of our approach, we compared the exploration time to prior work [2] that proposed different retention time units within each STT-RAM cache. We chose this prior work, called *LARS*, since it is the most related to ours and determined the optimal latency and energy configurations during runtime using exhaustive sampling. However, unlike LARS, which had four retention times, our implementation featured six retention times. In our implementation, each benchmark was first run on the base STT-RAM unit ($1ms$) for 1 million instructions, and the data was then used by SCART to predict the best retention time unit on which to run the rest of the application. Overall, SCART achieved similar results to exhaustive search (Section V-B).

Given SCART's similar performance to exhaustive search, we also evaluated SCART's benefit for reducing the exploration/tuning time. In LARS, the applications were sampled on each STT-RAM cache unit. Thus, LARS required six migrations between cache units for each tuning decision, with each migration taking 4608 cycles, which translates to 2.304μs at a 2GHz frequency. In total, the migration overhead was 13.824μs. SCART, for most of the cases, required only one migration if a different retention time than the base was determined to be the best. Therefore, SCART's average overhead (prediction + migration) was 6.554μs, reducing the exploration overhead by 52.58% compared to LARS, while achieving similar latency and energy savings. Furthermore, unlike LARS, which runs the application on potentially sub-optimal retention times before arriving at the best, SCART directly predicts the best without exploring sub-optimal retention times.

We assume that SCART is implemented in software (e.g., in the operating system). As such, SCART does not incur any hardware overhead other than the monitor counter described in Section III-A. However, SCART incurs some memory overhead. We used memory profiling to observe the memory consumed by SCART, and found that SCART consumes 0.156 MB of memory during the training stage and 2.5 KB of memory for the runtime prediction stage.

TABLE III: Multi-programmed workload distribution

#	Workload1	Workload2	Workload3	Workload4	Workload5	Workload6	Workload7	Workload8	Workload9	Workload10	Workload11	Workload12
1	bc_20	dijkstra	m_djpeg	cc_20	pr_20	gsm	tc_20	m_cjpeg	patricia	bfs	sssp_20	lame
2	patricia	sssp_20	lame	gsm	sssp_20	pr_20	bc_20	bfs_20	m_cjpeg	tc_20	m_djpeg	dijkstra
3	gsm	sssp_20	tc_20	bc_20	pr_20	cc_20	patricia	bfs_20	lame	m_cjpeg	m_djpeg	dijkstra
4	sssp_20	gsm	tc_20	dijkstra	patricia	pr_20	m_cjpeg	lame	bc_20	cc_20	bfs_20	m_cjpeg

VI. Conclusion and Future Work

In this paper we proposed an STT-RAM Cache Retention Time (SCART) model that uses a KNN classifier to predict the best retention time for an STT-RAM L1 cache. SCART uses execution statistics obtained from hardware performance counters. In a runtime single-core scenario, SCART predicted retention times that achieved average latency and energy savings of 20.34% and 29.12%, respectively, compared to a base $1ms$ retention time. In a quad-core scenario with multi-programmed workloads, SCART achieved average latency and energy savings of 25.07% and 36.13%, respectively, compared to a base $1ms$ retention time. Compared to prior work, SCART reduced the exploration time by 52.58%, while achieving similar latency and energy savings. Future work involves exploring a hardware implementation of SCART, extending SCART to predict other architecture parameters, and reducing the number of required labels in order to reduce the memory overhead, without sacrificing prediction accuracy.

Acknowledgement

This work was supported in part by the National Science Foundation under grant CNS-1844952. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] J. Li, C. J. Xue, and Y. Xu, "Stt-ram based energy-efficiency hybrid cache for cmps," in *2011 IEEE/IFIP 19th International Conference on VLSI and System-on-Chip*, Oct 2011, pp. 31–36.

[6] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[7] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, Dec 2001, pp. 3–14.

[8] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[9] 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.

[10] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 725–737.

[11] L. T. Yang, X. Ma, and F. Mueller, "Cross-platform performance prediction of parallel applications using partial execution," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2005, p. 40.

[12] P. Guo and L. Wang, "Accurate cross–architecture performance modeling for sparse matrix–vector multiplication (spmv) on gpus," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 13, pp. 3281–3294, 2015.

[13] X. Zheng, L. K. John, and A. Gerstlauer, "Accurate phase-level cross-platform power and performance estimation," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016, pp. 1–6.

[14] 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.

[15] D. Gajaria and T. Adegbija, "Arc: Dvfs-aware asymmetric-retention stt-ram caches for energy-efficient multicore processors," in *The International Symposium on Memory Systems (MEMSYS)*. ACM, 2019.

[16] G. Bonaccorso, *Machine Learning Algorithms: Popular algorithms for data science and machine learning*. Packt Publishing Ltd, 2018.

[17] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine Learning*, vol. 63, no. 1, pp. 3–42, Apr 2006.

[18] A. Mucherino, P. J. Papajorgji, and P. M. Pardalos, "K-nearest neighbor classification," in *Data mining in agriculture*. Springer, 2009, pp. 83–106.

[19] Y. Sasaki *et al.*, "The truth of the f-measure," *Teach Tutor mater*, vol. 1, no. 5, pp. 1–5, 2007.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[21] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[22] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging non-volatile memory," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 31, no. 7, pp. 994–1007, Jul. 2012.

[23] A. Abel, F. Benz, J. Doerfert, B. Dörr, S. Hahn, F. Haupenthal, M. Jacobs, A. H. Moin, J. Reineke, B. Schommer *et al.*, "Impact of resource sharing on performance and performance prediction: A survey," in *International Conference on Concurrency Theory*. Springer, 2013, pp. 25–43.