ARC: DVFS-Aware Asymmetric-Retention STT-RAM Caches for Energy-Efficient Multicore Processors

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

ABSTRACT

Relaxed retention (or volatile) spin-transfer torque RAM (STT-RAM) has been widely studied as a way to reduce STT-RAM's write energy and latency overheads. Given a relaxed retention time STT-RAM level one (L1) cache, we analyze the impacts of dynamic voltage and frequency scaling (DVFS)—a common optimization in modern processors—on STT-RAM L1 cache design. Our analysis reveals that, apart from the fact that different applications may require different retention times, the clock frequency, which is typically ignored in most STT-RAM studies, may also significantly impact applications' retention time needs. Based on our findings, we propose an asymmetric-retention core (ARC) design for multicore architectures. ARC features retention time heterogeneity to specialize STT-RAM retention times to applications' needs. We also propose a runtime prediction model to determine the best core on which to run an application, based on the applications' characteristics, their retention time requirements, and available DVFS settings. Results reveal that the proposed approach can reduce the average cache energy by 20.19% and overall processor energy by 7.66%, compared to a homogeneous STT-RAM cache design.

CCS CONCEPTS

• **Computer systems organization → Processors and memory architectures**;

KEYWORDS

Spin-Transfer Torque RAM (STTRAM); cache; retention time; nonvolatile memory; energy efficient systems; write energy; write latency; DVFS

ACM Reference format:
Dhruv Gajaria and Tosiron Adegbija. 2019. ARC: DVFS-Aware Asymmetric-Retention STT-RAM Caches for Energy-Efficient Multicore Processors. In *Proceedings of the International Symposium on Memory Systems, Washington, DC, USA, September 30-October 3, 2019 (MEMSYS '19),* 12 pages.
https://doi.org/10.1145/3357526.3357553

MEMSYS '19, September 30-October 3, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-7206-0/19/09…$15.00
https://doi.org/10.1145/3357526.3357553

1 INTRODUCTION

Caches remain very important components of modern-day and emerging processors due to their impact on both performance and power. While caches bridge the well-known processor-memory performance gap, on-chip caches can also consume a substantial amount (over 40%) of processors' power [37]. Thus, caches have been the focus of much research for power/energy and performance optimization. One such optimization involves replacing traditional SRAMs with the non-volatile spin-transfer torque RAM (STT-RAM) in emerging caches. STT-RAMs have substantially lower leakage power and higher density than SRAMs [24], [13], [53]. However, STT-RAMs have significant dynamic overheads due to long write latency and high dynamic write energy [53].

In the absence of a power source, STT-RAMs intrinsically preserve data for up to 10 years. This duration is called the *retention time*. Prior work [50] showed that the long write latency and high dynamic write energy directly result from this long retention time. As such, prior works [28], [50], [53], [24] have studied the benefits of reducing/relaxing the retention times. Relaxing the retention time, especially in caches, is beneficial since data blocks are usually only needed in the cache for a short duration (typically less than one second).

However, due to the reduced retention time, cache blocks may sometimes expire before they are evicted (or invalidated). This premature expiry results in a type of cache misses that we refer to as *expiration misses*—misses that occur because a referenced block prematurely expired due to elapsed retention. To prevent premature expiry, Dynamic Refresh-Scheme (DRS) [53] was proposed to refresh the cache block before the data expires. However this technique accrues additional hardware costs and needs multiple read/write operations for each refresh; this limits the optimization potential [33]. Various techniques have been proposed [33],[24] to reduce the number of refreshes in order to mitigate the overheads. These techniques use either compiler optimization techniques or physical circuits, which result in additional design, power and area overheads, especially in complex multicore systems.

Another increasingly popular optimization for emerging caches involves combining both STT-RAM and SRAM in a *hybrid architecture* to take advantage of STT-RAM's low read energy and SRAM's low write energy. For instance, Li et al. [32] explored hybrid caches comprising of both SRAM and STT-RAM, wherein write-intensive cache blocks were serviced by the SRAM cache and read-intensive blocks were serviced by the STT-RAM cache. Applications were scheduled to the cache that best satisfied the applications' execution needs. Donyanavard et al. [15] proposed to deploy a combination 'fast cores' using SRAM caches and 'slow cores' using STT-RAM caches for multicore processors.

In this paper, we focus on exploiting the energy benefits of reduced retention STT-RAM caches in multicore processors. We explore the interplay of variable retention times and expiration misses in level one (L1) STT-RAM caches, considering a system without any refresh mechanisms. Importantly, we perform this study within the context of variable clock frequencies, which have been ignored in prior studies of STT-RAM caches. Variable clock frequencies, otherwise known as dynamic frequency scaling (DFS) or dynamic voltage frequency scaling (DVFS) [4], is widely used in modern processors. In DVFS-enabled systems, the clock frequency is changed according to workload requirements in order to manage the power consumption, temperature, or performance [49], [5].

Prior work [31] found diminishing returns in using DVFS in modern computer systems featuring SRAM caches. However, we observed that the trends were different in systems with STT-RAM caches—DVFS still offers energy saving benefits if the interplay of the clock frequency and retention time is carefully considered. The analysis presented herein is motivated by two important observations. First, due to the reduction in clock period concomitant to frequency increases, the number of cache access cycles will increase as the frequency increases. Since STT-RAMs have a higher cache write access latency than SRAMs, we observe a higher change in cache write access cycles as the frequency increases. Secondly, for STT-RAM cache, whereas reducing the retention time will typically increase the expiration misses—a major performance indicator for STT-RAM caches—increasing the clock frequency may result in the eviction of data blocks since the frequency of cache accesses also increases, thereby mitigating the expiration misses.

These observations motivate us to perform a thorough analysis of retention times for a variety of workload characteristics within the context of DVFS. Based on our analysis, we explore the appropriate retention times for different execution scenarios and objective functions (energy and latency). We investigate the interplay of retention time specialization (via *asymmetric-retention cores*) and DVFS for energy savings in multicore STT-RAM caches.

Our major contributions are summarized as follows:

- For the first time, to our best knowledge, we analyze the impact of DVFS on reduced retention STT-RAM caches from the perspective of the cache and the overall processor. We explore the impact of frequency on STT-RAM cache misses and how it impacts the selection of the appropriate retention times for energy savings or performance improvement.
- For runtime retention time specialization to applications' requirements, we propose *asymmetric-retention cores (ARC)*. ARC equips cores in a multicore system with different retention times and frequency ranges, such that applications are executed on the core that best satisfies their execution needs. Furthermore, we propose a low-overhead machine learning-based prediction algorithm that maps applications to the appropriate cores during runtime.
- Using extensive simulations with a total of 30 applications (for training and testing), we show that our proposed work reduces the cache energy in a multicore system by 20.19%, on average, compared to a homogeneous STT-RAM system. Furthermore, our approach reduces the overall processor energy by 7.66% and 13.66%, on average, compared to a homogeneous multicore STT-RAM and SRAM system, respectively.
- Finally, we augment our approach to cater to a system with different execution performance constraints, and study the impact of the different constraints on our approach. We show that our prediction algorithm also works under various performance constraints.

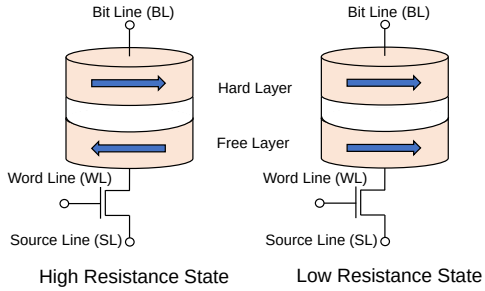


Figure 1: STT-RAM cell structure. High resistance state is in anti-parallel state and low resistance state is parallel state

2 BACKGROUND AND RELATED WORK

To provide a background for our work, this section first presents an overview of an STT-RAM cell. Thereafter, to provide context for our work, we describe prior related work on STT-RAM caches, dynamic voltage and frequency scaling, and heterogeneous multicore architectures.

2.1 Overview of STT-RAM Caches

Figure 1 illustrates the STT-RAM bit cell structure, which consists of a transistor and a magnetic tunnel junction (MTJ). In summary, MTJ comprises of two ferromagnetic layers separated by an oxide layer. The magnetization direction of the hard layer is fixed while the magnetization direction of free layer, which stores the memory bit, can be set by passing current through it. Details of the characteristics and functioning of the STT-RAM are provided in the prior work [9].

STT-RAM is a promising candidate for the cache hierarchy due to several characteristics, such as good scalability, high endurance (compared to other NVM technologies [50, 52]), near-zero leakage power, and radiation hardness. STT-RAM cells require a cell area from 1/9 to 1/3 that of SRAM, which enables a larger cache with the same die footprint [52]. However, deploying STT-RAM technology in caches is challenging because of high write latency and high dynamic write energy [50]. Smullen et al. [50] proposed relaxing the retention time of STT-RAM and observed that relaxing the retention time of STT-RAM cell reduces the latency and energy for write operations. STT-RAM can be designed with the desired retention time based on the understanding that the retention time is exponentially proportional to the magnetization stability energy, Δ, which can be expressed as:

$$\Delta \propto \frac{V.H_{\mathrm{k}}.M_{\mathrm{s}}}{T}$$

where V is activation volume for STT-RAM writing current, H_k is in-plane anisotropy field, M_s is saturation magnetization and T is absolute temperature in Kelvin [50].

We note that STT-RAM also has other issues, such as reliability and process variation issues, that still impede its widespread adoption in modern processors. While these issues are outside the scope of this paper, there is much ongoing research to address them. For instance, Emre et.al [16] studied the reliability of STT-RAM cell and showed that process variations and variations in device edge geometry affects their failure rate. They used error correction codes and circuit-level parameter tuning to obtain a block failure rate (defined as a binomial distribution of uniform errors, of 10^{-9}. Similarly, Alkabani et.al [27] used a combination of pulse width scaling and error correction coding schemes for STT-RAMs to reduce the write energy consumption. The proposed approach reduced the average write energy by 46% with uncorrectable bit error rate (UBER) of 10^{-13}.

2.2 Mitigating the Overheads of Relaxed Retention Time STT-RAM

Prior work [53] showed that relaxing the retention time of an STT-RAM cache could improve the IPC and energy by up to 80% and 84%, respectively, compared to non-volatile STT-RAM caches. However, relaxed retention STT-RAMs can still result in energy or time overheads resulting from premature expiration of data blocks. To mitigate the overheads of premature expiration, prior work also proposed various dynamic refresh schemes (DRS) [53], [24]. DRS refreshes data blocks if they must remain in the cache after the retention time elapses. These approaches typically use counters to monitor the block's lifetime in order to determine when to refresh the block. However, these techniques incur additional overheads, resulting from the refresh buffers and refresh operations. As such, optimization potential may be severely limited and other studies have been done to explore techniques for reducing the cost of refresh operations [33].

Kuan et al. [28] found that different applications require different retention times at a finer granularity than previously shown. They focused on mapping different applications with their best retention times, leading to latency and energy savings of up to 13.2% and 35.8%, respectively, compared to DRS. However, the proposed approach required four STT-RAM units in each core, resulting in hardware overhead, which our work mitigates. Furthermore, the impacts of variable clock frequency (and frequency scaling) on applications' retention time needs have not been explored by prior studies.

2.3 Dynamic Voltage Frequency Scaling

Dynamic voltage and frequency scaling (DVFS) is a popular optimization technique that is implemented in several mainstream processors. DVFS adjusts the CPU voltage and frequency according to runtime task requirements. There have been several studies on the impact on DVFS on system performance and energy [8, 10, 35]; we focus herein on the impact of DVFS on the cache.

Prior works have studied the impacts of voltage and frequency scaling on SRAM architectures [38, 45]. For instance, Wang et al. [54] used a combination of dynamic cache reconfiguration and DVS to achieve energy savings compared to a DVS-only system. To reduce energy, Saito et al. [43] suggested switching between high and low speed SRAM L1 caches operating at variable frequencies according to the requirements of executing applications. While these previous works used variable cache sizes, in our work, we keep the cache size constant and focus on the STT-RAM cache's retention time variability. However, we note that the idea of variable cache sizes is orthogonal to the work presented herein, and we leave this exploration for future research.

Peneau et al. [40] discussed the impact of frequency on cache access cycles for SRAM and STT-RAM. However, the authors focused their analysis on the selection of operating frequencies that have same latencies for SRAM and STT-RAMs for loop nest optimization. To the best of our knowledge, no prior study has analyzed how retention time design choices are affected by variable clock frequencies. Thus, we focus our analysis on the impact of variable frequencies on the right choice of retention times for executing applications' requirements, and suggest designs that leverage the synergy of DVFS and various retention times for energy savings.

2.4 Heterogeneous Multicore Architectures

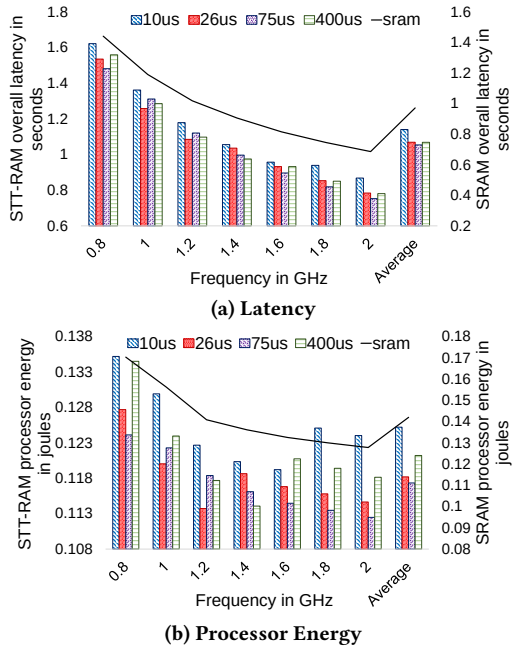
There has been much prior work on heterogeneous multicore architectures to enable the specialization of system resources to application requirements. These prior techniques have been leveraged for energy reduction, load balancing, thermal management, etc. [30, 44, 46, 51]. In addition, there is much prior work on scheduling of applications to the appropriate core during runtime [1, 47, 48]. Even though several system components (e.g., branch predictors, in-order vs out-of-order execution hardware, etc.) can be leveraged for heterogeneity, we focus on using the benefits of variable frequencies in synergy with heterogeneous retention time STT-RAM cores, which we call *asymmetric-retention cores (ARC)*, for energy savings in a multicore system. We believe that the work proposed herein will be beneficial for current and emerging heterogeneous core systems, such as the big.LITTLE cores [19].

3 INTERPLAY OF DVFS AND STT-RAM CACHES

In the analysis presented herein, we focus on L1 data cache since it exhibited more application variability than the instruction cache. A carefully chosen static instruction cache retention time—we empirically selected a 100*ms* retention time—sufficed for the considered applications in our work (similarly to prior work [28]). In this section, we analyze the impact of clock frequency variations on STT-RAM cache retention time requirements and expiration misses to provide a basis for the proposed ARC architecture.

3.1 Impact of Variable Clock Frequency on STT-RAM Caches

To explore the impact of variable clock frequency on STT-RAM caches in comparison to SRAM caches, we performed experiments using 30 benchmarks from the SPEC 2006 [22], MiBench [20], and GAP [2] benchmark suites to represent a variety of applications. An application's retention time requirement is a function of the application's *cache block lifetimes*—the duration for which a block must remain in the cache before it is evicted or invalidated [24]. Thus, to



(a) Latency

(b) Processor Energy

Figure 2: Impact of frequency scaling to performance and processor energy compared to SRAMs. SRAM caches are faster than STT-RAM caches but consume high energy

empirically determine a set of retention times for our experiments, we exhaustively analyzed the average cache block lifetimes of our benchmarks to determine the range of retention times that satisfied the different benchmarks' needs. We considered several retention times, but, based on the applications' characteristics, narrowed our focus down to: 10μs, 26.5μs, 75μs, and 400μs. (details of our experimental setup are in Section 5).

We observed and analyzed the changes in the cache and processor energy and latency with different retention time and frequency values. Ideally, the goal is to keep the cache access cycles (read and write) as low as possible. However, as expected, the clock frequency dictates the changes in the cache access cycles. The access cycles can be expressed with respect to frequency and access latency as $access\_cycles = ceil(frequency * access\_latency)$. Whereas the read and write latencies remain constant for SRAM, for STT-RAM, the read latency changes very slightly across different retention times, while the write latency increases more drastically as the retention time increases. As such, we observe a more pronounced impact of frequencies on STT-RAM caches than on SRAM caches.

Figure 2a depicts the latency achieved using the STT-RAM (with different retention times) and SRAM caches. All other processor configurations remain constant. Each bar and line represents average results across all the benchmarks. Our first observation from the figure is that the best frequency for SRAM, with respect to latency, is the highest—2GHz. For the STT-RAM system, on the other hand, the best retention time for latency varies with different frequencies—longer retention times do not necessarily improve the performance compared to shorter retention times. For instance, at 0.8GHz frequency, 75μs had the shortest latency as it requires one cycle per write operation, whereas at 1GHz to 1.2 GHz, 26μs performed better for latency due to the lower write cycles than

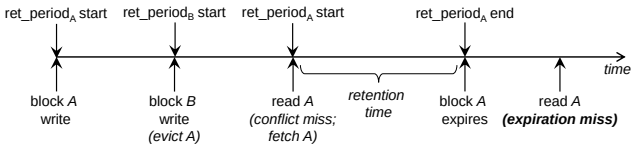


Figure 3: Illustration of expiration misses. Assume that blocks A and B are in the same memory location, i.e., a write from one block would evict the currently resident block

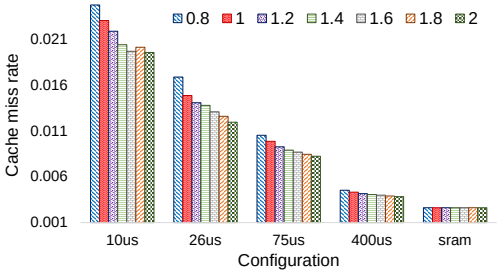


Figure 4: Change in miss rate with respect to frequency. The change in miss rates is observed due to decrease in expiration misses with increase in frequency

75μs and 400μs. We also observed that apart from the access cycles, the *expiration misses* also impacted the retention time requirements. For example, although 10μs and 26.5μs have equal cache access cycles, we observed that benchmarks exhibited fewer expiration misses on the 26.5μs cache, since several more cache blocks were prematurely evicted on the 10μs cache. As such, the 26.5μs cache was more favorable for latency than 10μs. Additional analysis of expiration misses is presented in Section 3.2.

Figure 2b depicts the changes in overall processor energy achieved by the STT-RAM and SRAM caches for different frequencies. Much like the latency results, the best frequency for energy in the SRAM cache is 2GHz. Even though the 2GHz would have a higher power consumption than lower frequencies, benchmarks are run much faster at 2GHz resulting in overall lower energy. These observations are similar to prior work [31]. However, we observed that the best frequency for energy changed for different retention times—higher frequencies were not necessarily always better. For instance, for 10μs, the best frequency is 1.6GHz, for 26.5μs the best is 1.2GHz, for 75μs the best is 2.0GHz, and for 400μs the best is 1.4GHz. Note that these results are on average across all the benchmarks; we observed variability among the benchmarks, revealing optimization potential when DVFS is employed in synergy with different retention times. These variable behaviors, similar to the latency results, result from the variance in cache access cycles and expiration misses, which we discuss in the following subsection.

3.2 Impact of Frequency on Expiration Misses

We define an *expiration miss* as a miss that results from a reference to a block that expired prematurely due to a short retention time. Figure 3 illustrates the occurrence of an expiration miss. Assume that blocks *A* and *B* reside in the same memory location; that is, if

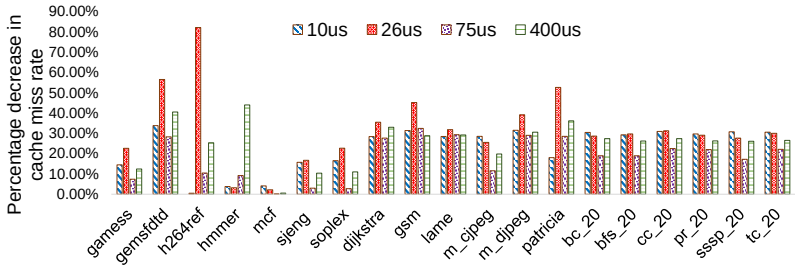


Figure 5: Decrease in cache miss rate with increase in frequency from 0.8GHz to 2.0GHz for various retention times. We observe specific retention times having high change in cache miss rates with respect to frequency due to variance in cache block lifetimes for different benchmarks

A is first written into the cache, a write of B evicts A from the cache. Thereafter, a read of A results in a conflict miss; A is brought into the cache, and its retention period is restarted. After the retention time elapses (at ret_time_A end), a reference to A results in an expiration miss, since the miss only occurred because of an elapsed retention time. Therefore, shorter retention times increase the expiration misses, and in effect, overall miss rates.

We observed that due to the presence of expiration misses in reduced retention STT-RAM caches, frequency has a much larger impact on STT-RAM cache miss rates than SRAM. Figure 4 shows the average miss rates of our benchmarks for the different retention times and SRAM with different frequencies. Whereas the miss rates of SRAM are relatively unaffected by frequency changes, the cache miss rates vary for STT-RAMs: in general, as the frequency increases, the miss rate decreases. This is because as the frequency increases, the cache block accesses also increase. As such, blocks that may have expired at lower frequencies may be replaced due to faster computations, thereby reducing the expiration misses. For most of the retention times, the highest decrease in cache expiration misses occurs when the frequency is increased from 0.8 GHz to 1GHz. However, due to increased cache access cycles with increase in frequency, we observe an increase in expiration misses for some cases.

To further illustrate that the impact of frequency in STT-RAM caches is also application-dependent, Figure 5 shows the reduction in cache miss rates observed when the frequency is increased from 0.8GHz to 2.0GHz. For brevity, the figure only shows an arbitrary subset of the considered benchmarks. The key takeaway from the figure is that the frequency-retention time interplay varies substantially for different applications. For instance, although $10\mu s$ typically has the highest miss rates, the frequency change had a higher impact for $26\mu s$ for some benchmarks, like $h264ref$, $soplex$ (from SPEC), and $patricia$ (from MiBench), whereas the highest impact observed for $hmmer$ (from SPEC) was with the $400\mu s$ cache. Apart from showing the impact of frequency and retention times on STT-RAM cache performance, these results also illustrate the variability of this interplay for different applications. This behavior makes it challenging—and necessary—to not only specialize retention times to applications requirements, but also to predict the best retention time, while considering the frequency and application-specific cache block lifetimes.

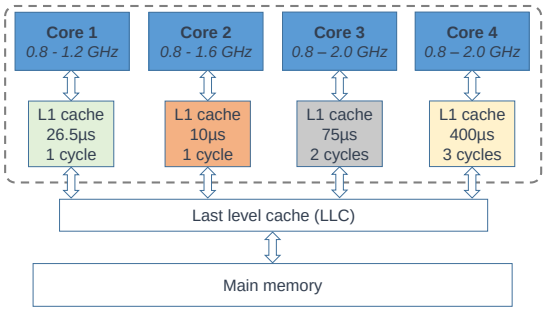


Figure 6: Proposed DVFS-aware asymmetric-retention core architecture. Each core features different frequency ranges and different retention times in the L1 STT-RAM cache, resulting in different cache write cycles at each core

4 DVFS-AWARE ASYMMETRIC-RETENTION CORE (ARC)

This section describes our proposed architecture for leveraging the interplay of DVFS and application-specific retention times for energy savings. Given the analysis in Section 3, it is clear that a balance must be achieved between leveraging the impacts of variable retention time and clock frequency requirements, while considering these factors' impacts on expiration misses.

4.1 Proposed ARC Architecture

We propose a DVFS-aware asymmetric-retention core (ARC) architecture featuring different retention times and frequency ranges to satisfy a variety of runtime execution needs. The ARC architecture also features a runtime prediction model that allows different applications to be scheduled to the appropriate core during runtime, based on the applications' characteristics. Note that the asymmetry of the proposed architecture can be extended to any number of configurations. For this work, however, we limit the architecture to heterogeneous L1 cache retention times and their frequency ranges, which dictate the caches' write access cycles.

Figure 6 illustrates the proposed architecture, which comprises of n cores ($n = 4$ in this work), each featuring a different range of clock frequencies and retention times. The range of frequencies and retention times were empirically selected, based on extensive

design-time analysis, to satisfy the variety of application characteristics in our benchmarks. The configurations for a different set of applications may be different. For all the cores, the frequencies range from 0.8GHz to different caps in order to constrain the write cycles for each core. The frequency for the 10μs core is capped at 1.6 GHz and the 26.5μs core is capped at 1.2 GHz to maintain a 1-cycle cache write latency. The 75μs and 400μs cores can operate at up to 2.0 GHz, with two and three cache write cycles, respectively.

Given the analysis in Section 3 and the characteristics of the cores in Figure 6, we expect that when performance optimization is the objective, application execution will be biased towards cores 3 and 4 (at 2 GHz). When energy optimization is the objective, there will be a higher variability in the best core. The experiments bear these out and are detailed in Section 6. We also observed a trend in the best frequencies for different retention times when energy is the objective function—higher frequencies were generally better for energy optimization. For instance, in general, the best frequencies for energy optimization for the 10μs and 26.5μs retention times were 1.6 GHz and 1.2 GHz, respectively, and for 75μs and 400μs, 2 GHz. Note that this is not necessarily *always* the case; these were general observations. These observations play into the experiments presented herein, since our focus is on energy consumption. However, we concede that the analysis may change if other optimizations (e.g, thermal management [12]), which are outside the scope of this paper, come into play.

With reduced retention times, there is a chance for data blocks to become unstable if the retention time elapses prior to a block's eviction [53]. To prevent occurrences of data corruption and maintain the data integrity, the cache blocks are augmented with monitor counter bits. The counter is implemented using a k-state finite state machine (FSM), which is controlled by the cache controller, and has a clock period that is k times smaller than the retention time. When the counter reaches state $k-1$—i.e., before the retention time elapses—the block is first written back to lower level memory (if dirty) and invalidated. For low-overhead implementation, we assumed 2-bit counters per block in our work for a 4-state counter FSM. A similar technique has been used in prior work [24].

4.2 Overview of ARC Prediction Approach

There are several prior works that employ machine learning for predicting the best core or configurations in computer systems [25, 26, 36], and our work is along the lines of these prior works. Our goal was to develop a low-overhead model for predicting the best core on which to run an application, based on the applications' execution characteristics. Since this is a runtime model, we also sought to develop a model that allows prediction via easily obtainable execution statistics from hardware performance counters. These requirements preclude the direct use of expiration misses (Section 3.2) for prediction, since expiration misses are difficult to estimate during runtime. Thus, we opted to explore a combination of application characteristics that correlate with the expiration misses to enable runtime prediction of the best core for new applications.

Figure 7 depicts the high-level flowchart of our approach for determining the best core on which to execute an application. When an application A is run, a history table is checked to see if A has been encountered before, in which case, A is run on the previously

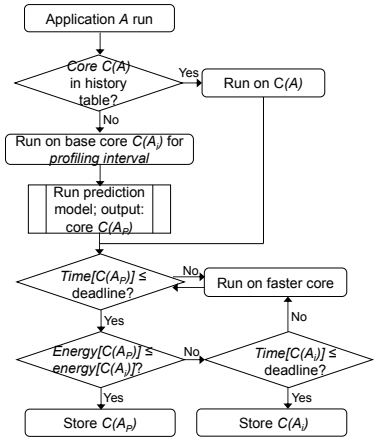


Figure 7: High-level overview of our approach. The flowchart also illustrates how runtime deadline constraints are handled

stored core $C(A)$. Otherwise, A is run on a *base* core $C(A_i)$—the fastest available core—for a *profiling interval*, which represents a brief period of time during which the application execution statistics are profiled. We experimented with various profiling intervals ranging from 1M instructions to 100M instructions, and found that 3M instructions was sufficient to obtain stable statistics for the required prediction. The statistics, including the cache statistics (e.g., miss rates, average accesses), memory controller statistics (e.g., bus write requests, bus utilization), are used as input parameters to the prediction model. Based on extensive empirical analysis, we determined that these statistics correlate strongly with an application's average cache block lifetimes and expiration miss behavior, which dictate the retention time requirements.

If some a priori performance constraints (e.g., deadlines) are specified for the application, the constraint information can be used to select a different base core in order to improve optimization potential. For instance, if an application's worst-case execution time on the fastest core is known, and a specified timing/deadline constraint substantially relaxes the worst-case time, a slower core (e.g., Core 2 vs. Core 4 in Figure 6) can be used as the base to reduce the energy consumption of the profiling stage. Note that the worst-case execution time (WCET) analysis [41] is outside the scope of this paper.

Next, the prediction model (Section 4.3) is run using statistics from the profiling stage as input, and the model outputs the best core $C(A_P)$ for running application A. Our approach checks that any specified deadline is met (default to ∞ if none is specified), and thereafter checks that the energy consumed on the predicted core $C(A_P)$ is less than the base. If the predicted core does not meet the deadline, a faster core is used (i.e., a core with a higher frequency cap), and the checking process is repeated for both time and energy. This checking process acts as a feedback that enables the best core for an application to be updated, when necessary (e.g., if that application is executed with a new input). Once a core is found that reduces the energy consumption or meets a specified deadline, the

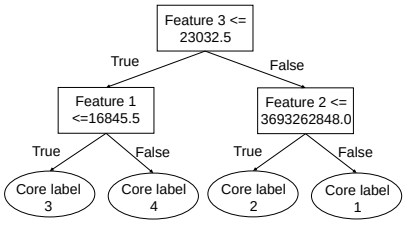


Figure 8: Illustration of decision tree for core predictions

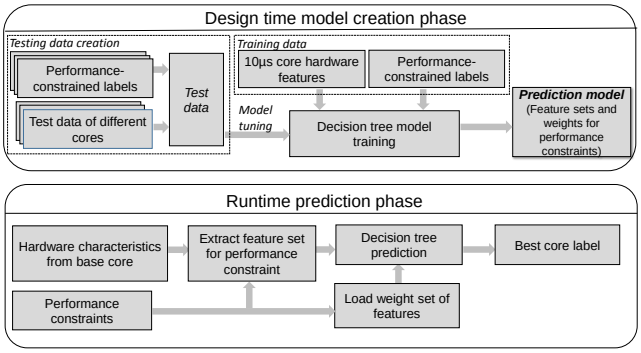


Figure 9: Flow of ARC prediction model. The model comprises of the design time model creation phase (featuring model training and testing) and runtime prediction phase

core is stored in the history table for subsequent executions of the application.

4.3 ARC Prediction Model

We explored several machine learning prediction models (e.g. Support Vector Classifiers, K-nearest neighbours, Naive Bayes, Random Forest classifier, etc.) to predict the best core/retention time, and found *decision trees* [42] to be the best fit for our purposes. Decision trees offers several advantages, including simplicity, low computational requirements, and fast prediction time. These features are especially important for a runtime implementation in a resource and timing constrained environment [17]. Furthermore, decision trees enable accurate predictions in the presence of small datasets, which enables us to further reduce the overhead of the prediction process [17, 55].

A decision tree splits a dataset into smaller subsets and uses a tree-like graph of decisions and their possible outcomes. Figure 8 illustrates a sample decision tree structure in our ARC prediction model. The topmost node in the tree is called the root node which is used for all the predictions. The decision tree then splits itself into either decision nodes or the leaf nodes (i.e, data output) depending on the dataset. To determine the quality of data splitting, our model uses an impurity criterion called *Gini index* [6], which measures how good a split is by determining how mixed the classes are in the two groups created by the split. The splitting of data and the formation of decision tree structure happen during the training stage. During the prediction stage at runtime, the corresponding hardware characteristics are input to the decision tree to predict an output core label. Depending on the data pattern, the complexity and depth of the decision tree will vary substantially. Thus, for different performance constraints, we used different tree structures. Additional low-level details and benefits of decision trees for resource-constrained prediction has been detailed in prior work [29].

Figure 9 presents the flow of the ARC prediction model[1]. The model comprises of two phases: 1) the *model creation phase*, which comprises of the model training and testing, and takes place at design time and 2) the *prediction phase*, which occurs during runtime for unknown applications or known applications with different timing/deadline constraints. In the model creation phase, we determined the feature set and their respective weights for different *performance constraints*. The performance constraints, which can be dictated by a user-specified deadline, for example, allow the model to explicitly relax the strictness of performance optimization in order to allow for higher energy savings. We observed that different performance constraints required different models for accurate runtime prediction, due to the variation of labels for different constraints. Thus, the model creation phase outputs different models for different levels of slack from the best performance possible.

To maintain low memory footprint and complexity, we used the $10\mu s$ core (Core 2) to obtain the hardware features for the model training. We used this core because it exhibited the highest rate of expiration misses and the highest amount of variability in execution characteristics compared to the other cores. The high variability in the data ensures more exact boundary conditions, thereby enabling more accurate training of our models. To validate this choice, we also experimented with using the other cores to obtain the training data and found that the highest runtime prediction accuracy was obtained using the $10\mu s$ core to obtain the training data.

To create/train the model, we began with 41 features (execution characteristics). Using feature selection [11], we determined the most relevant features for different performance constraints, and substantially reduced the feature set to 3 to 6 depending on the performance constraints. Table 1 lists the different performance constraints considered and their feature sets. We considered prediction scenarios for different target constraints, including no constraint (i.e., best energy), best performance possible, 10% slack on the best performance, and 20% slack. For the initial model creation, we used SPEC benchmarks as our training dataset, due to the diversity of execution characteristics, and used MiBench and GAP benchmarks for testing. We used a pre-trained model in order to substantially reduce runtime overheads and complexities.

During the prediction phase, the model first checks the performance constraint and then loads the respective model trained for the specified constraint. The downside of this technique is that in its current state, our approach only satisfies a limited set of performance constraints (Table 1). However, this does not degrade the overall effectiveness of our approach, and it can be extended for additional constraints. The corresponding features from the applications are extracted from hardware performance counters

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

Table 1: Feature sets for different deadline constraints

No constraint (best energy)	Best performance/10% slack	20% slack
L1 data cache hits	L1 data cache read misses	L1 data cache hits
L1 data cache read misses	Memory idle time	L1 data cache read accesses
L1 data cache total misses	Memory read hits	L1 data cache read misses
L1 instruction cache total misses	–	Memory idle time
Memory bus utilization for read operations	–	Memory bus utilization for write operations

Table 2: Processor and cache configurations

Processor configuration	8GB RAM, in-order, 2-wide, 0.8 – 2 GHz				
Cache	22nm, 32KB, 64B line size, 4-way				
Memory device	SRAM	STT-RAM			
Retention times	–	10μs	26.5μs	75μs	400μs
Hit latency	0.453ns	0.464ns	0.454ns	0.445ns	0.443ns
Write latency	0.312ns	0.601ns	0.769ns	0.981ns	1.389ns
Read energy (per access)	0.007nJ	0.003nJ	0.003nJ	0.003nJ	0.003nJ
Write energy (per access)	0.006nJ	0.026nJ	0.030nJ	0.035nJ	0.045nJ
Leakage power	50.328mW	13.1448mW			

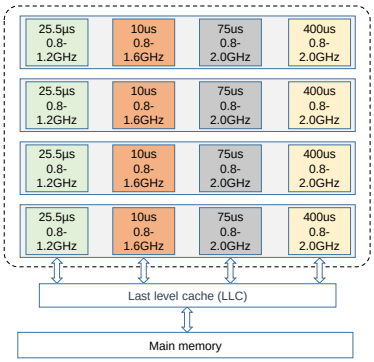


Figure 10: Scaling ARC for 16 cores featuring clusters of four cores. Each core has a different frequency range and cache retention time to satisfy different application requirements

after executing them for a profiling interval. Thereafter, the appropriate weights obtained from the training phase are then used in the decision tree model [42] to predict the best core label for the executing application.

4.4 Scalability

In this paper, we do not provide a comprehensive study of ARC's scalability to more than four cores; we leave this study for future work. However, in this subsection, we briefly describe how we anticipate ARC to scale beyond four cores to many-core systems.

There are multiple alternatives for scaling ARC beyond four cores, depending on the specific use case. Figure 10 illustrates a candidate design for a sixteen-core system. Using the analysis described in the Section 4.1, we empirically found that a set of four cores could optimize a variety of applications. We anticipate that for most systems, even in many-core systems, a subset of configurations will suffice to optimize the energy and/or performance for the variety of applications that execute on such systems. As such, ARC can be scaled to more cores simply by replicating the design in clusters of cores. In Figure 10, for example, each core's parameters are similar to the base ARC architecture and all the four different cores within each cluster are similarly configured. The advantage of this approach is that the prediction technique for the base ARC architecture can be applied to this architecture as well. In a system where a different set of configurations are required, as dictated by the executing applications or application domains, ARC's prediction model may need to be updated to support the new labels. We do not anticipate that ARC will introduce substantial communication traffic; thus, the system's interconnection network can also be used for ARC's traffic without creating a bottleneck in the system.

5 EXPERIMENTAL SETUP

For our simulations, we modeled a quad-core ARM processor with parameters similar to Cortex A-53 using an in-house modified version of the GEM5 simulator [3]. The modified GEM5 models the behavior of relaxed retention STT-RAM L1 caches with specified retention times. Each core has private 32KB, 4-way L1 instruction and data caches, with 64B line sizes. The data cache retention times used were: 10μs, 26.5μs, 75μs and 400μs, while 100ms was used for all instruction caches. Through extensive design space exploration, we found that these retention times sufficed for the average cache block lifetimes of the benchmarks used in our experiments. The core frequencies ranged from 0.8 GHz to 2 GHz with an incremental step size of 0.2 GHz, and voltages ranged from 0.9V to 1.35V similar to prior work [18]. The specific asymmetry of the STT-RAM cache retention times, frequency ranges, and cache write access cycles featured in the modeled cores are shown in Figure 6. We modeled unit cache read cycles for all STT-RAM caches. For comparison to SRAM, we assumed unit cache read and write access cycles.

To model the STT-RAM and SRAM cache energy, we used NVSim [14] integrated with the GEM5 statistics. The NVSim and GEM5 statistics were then integrated with the McPAT [34] simulator to model the energy consumption of the whole processor. Additional details of the modeled processor and cache configurations are shown in Table 2.

To represent a variety of workloads, we used 30 benchmarks from three benchmark suites: 18 from SPEC 2006[2] [22], 6 from MiBench [20] and 6 benchmarks from GAP [2]. We used thirteen arbitrary SPEC benchmarks for training, and the rest, combined with MiBench and GAP, for runtime experiments. Each benchmark was run for a maximum of 1 billiion instructions (some benchmarks, e.g., from MiBench have fewer than 1 billion instructions). To model different performance constraints, we assumed that the strictest constraint could be met using the fastest core and relaxed the constraints by slacks of 10% and 20%. Performance constraints can be determined by designer-specified deadlines or implicitly specified by the executing application [23]. We used the Scikit learn (Sklearn) [39] library in python to implement the decision trees in our model.

[2]We were unable to use the full suite due to compilation errors.

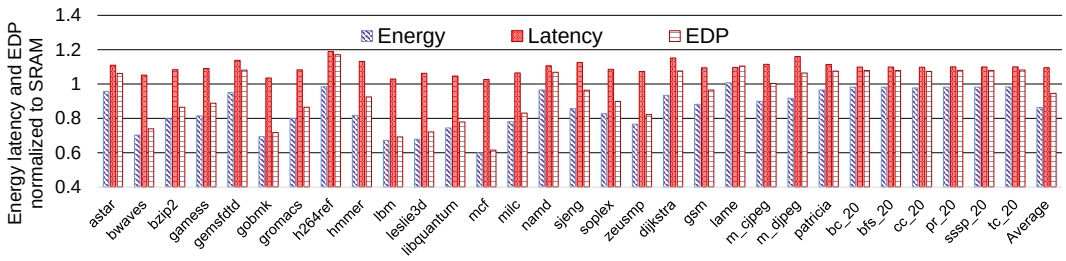


Figure 11: Energy savings potential of ARC vs a processor with SRAM caches (baseline of one) for different benchmarks. Results are with respect to the whole processor

6 ARC EVALUATION

We evaluate the proposed ARC architecture by comparing with SRAM, a processor with homogeneous STT-RAM caches, and prior work that proposed logically adaptable retention STT-RAM caches (LARS) [28]. We implemented LARS in a multicore scenario for comparsion to our work. Note that the results presented herein are with respect to the *whole* processor, which, as expected, are substantially less savings than cache-specific evaluations. To enable per-application evaluation, we assume a system with preemption; thus, applications can preempt background tasks in order to run on the predicted core. This is a realistic assumption for common resource-constrained devices, such as smartphones [21]. Section 6.3 evaluates ARC in the context of multiprogrammed workloads.

6.1 Comparison to SRAM

First, we evaluate the *energy savings potential* of the ARC architecture by comparing with a processor featuring SRAM caches. Figure 11 depicts the energy and latency achieved using the ARC architecture normalized to SRAM. On average across all the benchmarks, ARC reduced the energy by 13.66% compared to SRAM, with energy savings as high as 40.02% for mcf. The average cache-only energy savings was 39.21%. We observed that ARC performed best for workloads that exhibited low write accesses and high miss rates. As a result, ARC generally achieved higher energy savings for the SPEC and MiBench benchmarks than for the GAP benchmarks. However, ARC outperformed SRAM for all the benchmarks with respect to energy. Compared to SRAM, ARC improved the processor's energy-delay product (EDP) by an average of 5.44% with savings as high as 38.45% for mcf. Considering just the cache, ARC reduced the average energy by 39.21% compared to a baseline SRAM cache, with energy optimization as high as 68.90% for mcf (detailed graphs omitted for brevity).

The energy savings was achieved at the cost of some latency overhead. Compared to SRAM, ARC degraded the latency by 9.52% on average across all the benchmarks, with degradations as high as 18.95% for $h264ref$. SRAM achieved these latency benefits because of lower write access cycles and lower cache miss rates. However, we note that the degradation with respect to SRAM is not a unique flaw of our proposed approach. Prior work observed similar trends. For instance, Cheng et al. [7] observed 24% degradation in memory access latency resulting from replacing SRAM with STT-RAM, prompting their proposal to use a hybrid (SRAM + STT-RAM) cache to reduce the performance overhead. The hybrid cache is orthogonal to our work, and the latency overheads can be reduced by augmenting the cache asymmetry proposed herein by using SRAM in some of the cores.

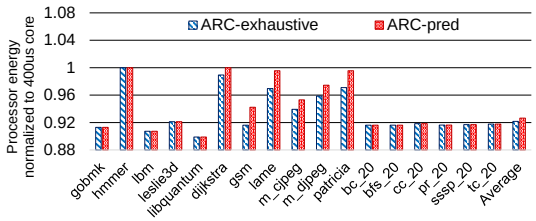
6.2 Comparison to Homogeneous Retention STT-RAM and Exhaustive Search

In this section, we compare the energy savings of the ARC prediction model to a processor with an optimized homogeneous retention time and to a system called *ARC-exhaustive*. In the homogeneous system, the best average retention time is determined via a priori design space exploration and featured in all the cores. In ARC-exhaustive, sampling is used to determine the *best* core for each application. For rigorous experimentation, our training data comprised of 13 benchmarks from SPEC benchmark suite, and our test data comprises of 17 benchmarks: five from SPEC, six from MiBench, and six from GAP. We evaluated the energy savings in the context of four performance constraints: no constraint, 20% slack, 10% slack, and best performance required. The results obtained by our ARC prediction model are compared with both the homogeneous system and with ARC-exhaustive.

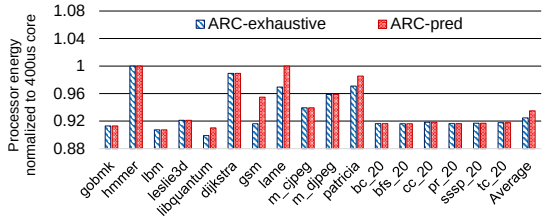
Figure 12 presents the energy savings of exhaustive search (ARC-exhaustive) and the prediction model (ARC-pred) normalized to a processor featuring 400μs (Core 4 in Figure 6) STT-RAM caches across all the cores. We used the 400μs for comparison, since it achieved the lowest miss rates on average across all the benchmarks. On average, ARC-pred achieved energy savings of 7.35%, 6.51%, 6.04% and 4.34% for no constraint, 20% slack, 10% slack, and best performance, respectively, and the results were within 0.57% of ARC-exhaustive, on average. Energy savings were up to 10.11% for the 10% slack for *libquantum*. We reiterate that the comparisons are with respect to the whole processor, and the energy savings are a lot higher (> 20% on average) when considering just the cache.

Whereas ARC-pred performed similar to ARC-exhaustive for most benchmarks, there were a few benchmarks where ARC-pred degraded the energy compared to ARC-exhaustive. In the worst case, ARC-pred degrade the energy by up to 3.04% for *lame* under 20% slack, due to false prediction; however, our prediction approach never degraded the energy savings compared to the base core as a result of the feedback process (Section 4.2).

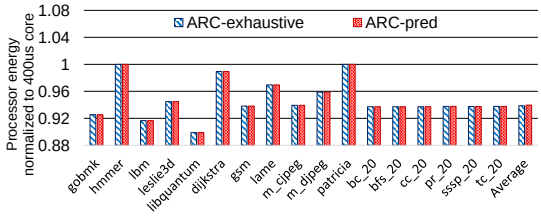
As expected, ARC's energy savings decrease as the performance constraints become more stringent. As seen in Figures 12a, 12b, and 12c, as the performance constraint becomes stricter, the energy



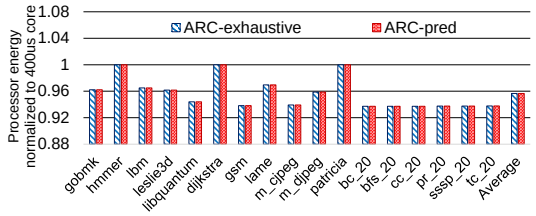
(a) No constraint



(b) 20% slack



(c) 10% slack



(d) Best performance required

Figure 12: ARC energy savings normalized to a base processor featuring the 400μs STT-RAM cache on all cores (baseline of one) for different performance constraints. Results are with respect to the whole processor

Table 3: Multiprogrammed workload distribution

#	Workload1	Workload2	Workload3	Workload4	Workload5	Workload6
1	dijkstra	lbm	gsm	cc_20	patricia	hmmer
2	pr_20	hmmer	gobmk	bc_20	cc_20	m_cjpeg
3	m_djpeg	m_cjpeg	sssp_20	leslie3d	pr_20	tc_20
4	lame	bfs_20	libquantum	tc_20	libquantum	bc_20

savings also decrease. Overall, ARC-pred met the constraints for all the benchmarks (Figures 12c and 12b). In a few cases (e.g., for *patricia* with 10% slack), constraints were initially violated due to mispredictions. However, ARC was able to correct the best core during the feedback process.

We note that the processor energy savings achieved is a result of the proper mapping of applications to the respective cores. This mapping enabled right-provisioned clock frequencies in consonance with an optimal number of write cycles among the available cores. Notably, we also explored using ARC for cache-specific savings and found that ARC reduced the cache energy by more than 20% in the different execution scenarios. However, we also found that the best configurations for cache-specific energy optimization were not necessarily the best when considering the whole processor. As such, we focused our analysis on the processor-wide results.

6.3 ARC Evaluation in a Multiprogrammed Scenario

In this section, we evaluate ARC in a multiprogrammed execution scenario. We created six multiprogrammed workloads comprising of four randomly selected benchmarks from our 17 test applications, ensuring that all 17 benchmarks are represented in the workloads. Table 3 depicts the workload compositions used. Unlike for previous experiments, we assume no preemption in this case. That is, if the predicted best core is unavailable, the next best available core is used. As such, we augmented the classifier in our prediction model to include a ranking of the output labels.

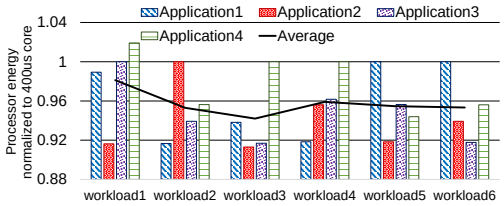


Figure 13: ARC energy savings in a multi-programmed scenario. We present results assuming no performance constraints are specified. If the predicted core is occupied, the next best available core is selected

Figure 13 depicts the overall processor energy of ARC normalized to a homogeneous STT-RAM design. The average energy savings was 1.88%, 4.70%, 5.80%, 4.10%, 4.53% and 4.67% for workloads one through six, respectively. The average energy savings across all the workloads was 4.28%. Overall, energy savings were lower since some benchmarks were forced to run on sub-optimal cores. We also observed higher energy consumption than the base core for *workload*1 due to the unavailability of the best core for application 4 (*lame*). While these results illustrate the energy savings potentials of ARC, we note that some nuances are ignored herein. For instance, the impact of preemption can be studied further; i.e., benefits of waiting on the best core vs. running on a sub-optimal core. These nuances become more important in performance-constrained scenarios. We plan to explore these nuances in future work.

6.4 Comparison to Prior Work

We compared ARC-pred to a multicore implementation of the logically adaptable retention STT-RAM (LARS) cache proposed in [28]. LARS features four STT-RAM units with different retention times within a single chip to enable specialization to applications'

needs. We observed similar energy savings between ARC and LARS (graphs omitted for brevity). However, notably, ARC only features a single STT-RAM unit per core, thereby reducing the per-core physical L1 cache area by 75% compared to LARS. Furthermore, we compared our work to prior art with respect to the migration overheads. Each migration in LARS took approximately 10μs. By directly predicting the best core, ARC reduced the migration overheads by 67.80%, on average, compared to LARS.

6.5 ARC Overheads

For brevity, we only report overheads for when no performance constraints are specified, since the performance-constrained scenarios are extensions of this. To minimize hardware overheads, we assume a software implementation of the prediction model and the history table (Section 4.2). For both the model and history table (assuming a 120-entry table), ARC consumed a total of 0.0167MB of RAM. The history table was large enough for all the applications considered in our experiments, and the overhead included the models for the different performance constraints. When the history table is full, however, entries can be replaced using a replacement policy, such as least recently used. As described in Section 4.1, the monitor counter introduced a 2-bit per block overhead, resulting in a total overhead of 128 B per core (i.e., a two-block overhead per core).

We also evaluated ARC overheads with respect to the prediction time. On average across all the benchmarks, the prediction time—including collecting the hardware characteristics and running the ARC prediction algorithm—was 3.23μs. We analyzed the migration overheads and found that, on average, ARC accrued migration overheads of 7.94μs. We plan to explore the tradeoffs of a hardware implementation in future work.

7 CONCLUSION AND FUTURE WORK

In this paper, we explored the behavior of STT-RAMs with variable clock frequencies, with respect to applications' runtime cache block requirements. Our analysis showed that, unlike SRAM caches, STT-RAMs exhibit substantial variability regarding the best clock frequencies; the applications' retention time requirements must also be taken into consideration. Thus, to enable energy savings by exploiting the interplay of DVFS and variable retention time requirements, we proposed the *Asymmetric-Retention Core (ARC)* architecture for multicore systems. ARC features STT-RAM caches with different retention times, clock frequencies, and access cycles in different cores, such that applications are run on the core that best satisfies the application's needs. We also proposed a runtime decision tree-based ARC prediction model that directly predicts the best core on which to run an application, based on the application's execution characteristics. Using extensive simulations and experiments with several execution scenarios, results reveal that ARC can reduce the average cache energy by 39.21% and the overall processor energy by 13.66%, compared to a system with SRAM caches. Compared to a system with optimized homogeneous STT-RAM cache, ARC can reduce the average cache energy and overall processor energy by 20.19% and 7.66%, respectively. Future work includes exploring ARC in more complex many-core heterogeneous systems with multilevel cache hierarchies.

ACKNOWLEDGEMENT

This work was supported in part by the National Science Foundation under grant CNS-1844952 (CAREER). 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] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.

[2] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[4] B. Brock and K. Rajamani. 2003. Dynamic power management for embedded systems [SOC design]. In *IEEE International [Systems-on-Chip] SOC Conference, 2003. Proceedings.* 416–419.

[5] D. Brooks and M. Martonosi. 2001. Dynamic thermal management for high-performance microprocessors. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture.* 171–182.

[6] B Chandra and P Paul Varghese. 2009. Fuzzifying Gini Index based decision trees. *Expert Systems with Applications* 36, 4 (2009), 8549–8559.

[7] Wei-Kai Cheng, Yen-Heng Ciou, and Po-Yuan Shen. 2016. Architecture and data migration methodology for L1 cache design with hybrid SRAM and volatile STT-RAM configuration. *Microprocessors and Microsystems* 42 (2016), 191–199.

[8] Kihwan Choi, R. Soma, and M. Pedram. 2004. Dynamic voltage and frequency scaling based on workload decomposition. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design (IEEE Cat. No.04TH8758).* 174–179.

[9] K. C. Chun, H. Zhao, J. D. Harms, T. H. Kim, J. P. Wang, and C. H. Kim. 2013. 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* 48, 2 (Feb 2013), 598–610. https://doi.org/10.1109/JSSC.2012.2224256

[10] T. R. da Rosa, V. LarrÃĺa, N. Calazans, and F. G. Moraes. 2012. Power consumption reduction in MPSoCs through DFS. In *2012 25th Symposium on Integrated Circuits and Systems Design (SBCCI).* 1–6. https://doi.org/10.1109/SBCCI.2012.6344429

[11] Manoranjan Dash and Huan Liu. 1997. Feature selection for classification. *Intelligent data analysis* 1, 1-4 (1997), 131–156.

[12] James Donald and Margaret Martonosi. 2006. Techniques for multicore thermal management: Classification and new exploration. In *ACM SIGARCH Computer Architecture News*, Vol. 34. IEEE Computer Society, 78–88.

[13] Xiangyu Dong, Xiaoxia Wu, Guangyu Sun, Yuan Xie, Helen Li, and Yiran Chen. 2008. Circuit and Microarchitecture Evaluation of 3D Stacking Magnetic RAM (MRAM) As a Universal Memory Replacement. In *Proceedings of the 45th Annual Design Automation Conference (DAC '08).* ACM, New York, NY, USA, 554–559.

[14] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P. Jouppi. 2012. NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 31, 7 (July 2012), 994–1007.

[15] Brvan Donyanavard, Amir Mahdi Hosseini Monazzah, Nikil Dutt, and Tiago Mück. 2018. Exploring Hybrid Memory Caches in Chip Multiprocessors. In *2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC).* IEEE, 1–8.

[16] Yunus Emre, Chengen Yang, Ketul Sutaria, Yu Cao, and Chaitali Chakrabarti. 2012. Enhancing the reliability of STT-RAM through circuit and system level techniques. In *2012 IEEE Workshop on Signal Processing Systems.* IEEE, 125–130.

[17] AS Galathiya, AP Ganatra, and CK Bhensdadia. 2012. Improved decision tree induction algorithm with feature selection, cross validation, model complexity and reduced error pruning. *International Journal of Computer Science and Information Technologies* 3, 2 (2012), 3427–3431.

[18] Rem Gensh, Ali Aalsaud, Ashur Rafiev, Fei Xia, Alexei Iliasov, Alexander Romanovsky, and Alex Yakovlev. 2015. *Experiments with odroid-xu3 board.* Newcastle University, Computing Science.

[19] P Greenhalgh. 2011. big. little processing with arm cortex-a15 and cortex-a7. 2011. *Citado na* (2011), 46.

[20] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. 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).* 3–14.

[21] Sangwook Shane Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, and Jihong Kim. 2017. Improving user experience of android smartphones using foreground app-aware I/O management. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*. ACM, 5.

[22] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[23] Christopher J Hughes, Praful Kaul, Sarita V Adve, Rohit Jain, Chanik Park, and Jayanth Srinivasan. 2001. Variability in the execution of multimedia applications and implications for architecture. *ACM SIGARCH Computer Architecture News* 29, 2 (2001), 254–265.

[24] Adwait Jog, A. K. Mishra, C. Xu, Y. Xie, V. Narayanan, R. Iyer, and C. R. Das. 2012. Cache revive: Architecting volatile STT-RAM caches for enhanced performance in CMPs. In *DAC Design Automation Conference 2012*. 243–252.

[25] Hwisung Jung and Massoud Pedram. 2010. Supervised learning based power management for multicore processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29, 9 (2010), 1395–1408.

[26] Sukhun Kang and Rakesh Kumar. 2008. Magellan: a search and machine learning-based framework for fast multi-core design space exploration and optimization. In *2008 Design, Automation and Test in Europe*. IEEE, 1432–1437.

[27] Yusung Kim, Sumeet Kumar Gupta, Sang Phill Park, Georgios Panagopoulos, and Kaushik Roy. 2012. Write-optimized reliable design of STT MRAM. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*. ACM, 3–8.

[28] Kyle Kuan and Tosiron Adegbija. 2019. Energy-Efficient Runtime Adaptable L1 STT-RAM Cache Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019).

[29] Ashish Kumar, Saurabh Goyal, and Manik Varma. 2017. Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70 (ICML'17)*. JMLR.org, 1935–1944. http://dl.acm.org/citation.cfm?id=3305381.3305581

[30] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. 2003. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, Washington, DC, USA, 81–. http://dl.acm.org/citation.cfm?id=956417.956569

[31] Etienne Le Sueur and Gernot Heiser. 2010. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*. 1–8.

[32] J. Li, C. J. Xue, and Yinlong Xu. 2011. STT-RAM based energy-efficiency hybrid cache for CMPs. In *2011 IEEE/IFIP 19th International Conference on VLSI and System-on-Chip*. 31–36. https://doi.org/10.1109/VLSISoC.2011.6081626

[33] Qingan Li, Jianhua Li, Liang Shi, Chun Jason Xue, Yiran Chen, and Yanxiang He. 2013. Compiler-assisted refresh minimization for volatile STT-RAM cache. In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 273–278. https://doi.org/10.1109/ASPDAC.2013.6509608

[34] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 469–480.

[35] Yongpan Liu, Huazhong Yang, Robert P Dick, Hui Wang, and Li Shang. 2007. Thermal vs energy optimization for dvfs-enabled processors in embedded systems. In *8th International Symposium on Quality Electronic Design (ISQED'07)*. IEEE, 204–209.

[36] Jose F Martinez and Engin Ipek. 2009. Dynamic multicore resource management: A machine learning approach. *IEEE micro* 29, 5 (2009), 8–17.

[37] James Montanaro, Richard T. Witek, Krishna Anne, Andrew J. Black, Elizabeth M. Cooper, Daniel W. Dobberpuhl, Paul M. Donahue, Jim Eno, Gregory W. Hoeppner, David Kruckemyer, Thomas H. Lee, Peter C.M. Lin, Liam Madden, Daniel Murray, and Mark H. Pearce. 1997. 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *Digital Technical Journal* 9, 1 (1997), 49–62.

[38] K. J. Nowka, G. D. Carpenter, E. W. MacDonald, H. C. Ngo, B. C. Brock, K. I. Ishii, T. Y. Nguyen, and J. L. Burns. 2002. A 32-bit PowerPC system-on-a-chip with support for dynamic voltage scaling and dynamic frequency scaling. *IEEE Journal of Solid-State Circuits* 37, 11 (Nov 2002), 1441–1447. https://doi.org/10.1109/JSSC.2002.803941

[39] 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. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[40] P. Peneau, R. Bouziane, A. GamatiÃĪ, E. Rohou, F. Bruguier, G. Sassatelli, L. Torres, and S. Senni. 2016. Loop optimization in presence of STT-MRAM caches: A study of performance-energy tradeoffs. In *2016 26th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 162–169.

[41] Archana Ravindar and Y. N. Srikant. 2011. Relative Roles of Instruction Count and Cycles Per Instruction in WCET Estimation. In *Proceedings of the 2Nd ACM/SPEC International Conference on Performance Engineering (ICPE '11)*. ACM, New York, NY, USA, 55–60. https://doi.org/10.1145/1958746.1958758

[42] S Rasoul Safavian and David Landgrebe. 1991. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics* 21, 3 (1991), 660–674.

[43] K. Saito, R. Kobayashi, and H. Shimada. 2016. Reduction of cache energy by switching between L1 high speed and low speed cache under application of DVFS. In *2016 International Conference On Advanced Informatics: Concepts, Theory And Application (ICAICTA)*. 1–6. https://doi.org/10.1109/ICAICTA.2016.7803082

[44] S. Sarma, T. Muck, L. A. D. Bathen, N. Dutt, and A. Nicolau. 2015. SmartBalance: A sensing-driven linux load balancer for energy efficiency of heterogeneous MPSoCs. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2744769.2744911

[45] Avesta Sasan (Mohammad A Makhzan), Houman Homayoun, Ahmed Eltawil, and Fadi Kurdahi. 2009. Process Variation Aware SRAM/Cache for Aggressive Voltage-frequency Scaling. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '09)*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 911–916. http://dl.acm.org/citation.cfm?id=1874620.1874845

[46] S. Sharifi, A. K. Coskun, and T. S. Rosing. 2010. Hybrid dynamic energy and thermal management in heterogeneous embedded multiprocessor SoCs. In *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 873–878.

[47] Daniel Shelepov and Alexandra Fedorova. 2008. Scheduling on heterogeneous multicore processors using architectural signatures. (2008).

[48] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. 2009. HASS: a scheduler for heterogeneous multicore systems. *ACM SIGOPS Operating Systems Review* 43, 2 (2009), 66–75.

[49] K. Skadron, M. R. Stan, W. Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and D. Tarjan. 2003. Temperature-aware microarchitecture. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* 2–13. https://doi.org/10.1109/ISCA.2003.1206984

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

[51] Thannirmalai Somu Muthukaruppan, Anuj Pathania, and Tulika Mitra. 2014. Price Theory Based Power Management for Heterogeneous Multi-cores. *SIGARCH Comput. Archit. News* 42, 1 (Feb. 2014), 161–176. https://doi.org/10.1145/2654822.2541974

[52] Z. Sun, X. Bi, H. Li, W. Wong, and X. Zhu. 2014. STT-RAM Cache Hierarchy With Multiretention MTJ Designs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22, 6 (June 2014), 1281–1293. https://doi.org/10.1109/TVLSI.2013.2267754

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

[54] W. Wang and P. Mishra. 2010. Leakage-Aware Energy Minimization Using Dynamic Voltage Scaling and Cache Reconfiguration in Real-Time Systems. In *2010 23rd International Conference on VLSI Design*. 357–362.

[55] Desheng Wu. 2009. Supplier selection: A hybrid model using DEA, decision tree and neural network. *Expert Systems with Applications* 36, 5 (2009), 9105 – 9112. http://www.sciencedirect.com/science/article/pii/S095741740800910X