A Study of Runtime Adaptive Prefetching for STTRAM L1 Caches

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

***Abstract*—Spin-Transfer Torque RAM (STTRAM) is a promising alternative to SRAM in on-chip caches due to several advantages. These advantages include non-volatility, low leakage, high integration density, and CMOS compatibility. Prior studies have shown that relaxing and adapting the STTRAM retention time to runtime application needs can substantially reduce overall cache energy without significant latency overheads, due to the lower STTRAM write energy and latency in shorter retention times. In this paper, as a first step towards efficient prefetching across the STTRAM cache hierarchy, we study prefetching in reduced retention STTRAM L1 caches. Using SPEC CPU 2017 benchmarks, we analyze the energy and latency impact of different prefetch distances in different STTRAM cache retention times for different applications. We show that *expired_unused_prefetches*—the number of unused prefetches expired by the reduced retention time STTRAM cache—can accurately determine the best retention time for energy consumption and access latency. This new metric can also provide insights into the best prefetch distance for memory bandwidth consumption and prefetch accuracy. Based on our analysis and insights, we propose *Prefetch-Aware Retention time Tuning (PART)* and *Retention time-based Prefetch Control (RPC)*. Compared to a base STTRAM cache, PART and RPC collectively reduced the average cache energy and latency by 22.24% and 24.59%, respectively. When the base architecture was augmented with the state-of-the-art near-side prefetch throttling (NST), PART+RPC reduced the average cache energy and latency by 3.50% and 3.59%, respectively, and reduced the hardware overhead by 54.55%.**

I. Introduction

Much research has focused on optimizing caches' performance and energy efficiency due to the caches' non-trivial impact on processor architectures. These optimization efforts are especially important for resource-constrained devices for which low-overhead energy reduction remains a major concern. An increasingly popular approach for improving caches' energy efficiency involves replacing the traditional SRAM with emerging non-volatile memory (NVM) technologies.

Among several NVM alternatives, Spin-Transfer Torque RAM (STTRAM) has emerged as a promising candidate for replacing traditional SRAMs in future on-chip caches. STTRAMs offer several attractive characteristics, such as non-volatility, low leakage, high integration density, and CMOS compatibility. However, some of STTRAM's most important challenges include its long write latency and high write energy [1], [2]. These challenges are attributed, in part, to the STTRAM's long *retention time*—the duration for which data is maintained in the memory in the absence of power. For caches, the intrinsic STTRAM retention time of up to 10 years is unnecessary, since most cache blocks need to be retained in the cache for no longer than 1s [3]. Furthermore, different applications or application phases may have different retention time requirements [4]. Thus, prior research has proposed reduced retention STTRAMs that can be specialized to the needs of various applications [4] or different cache levels [5].

To further improve cache efficiency, cache prefetching is a popular technique that fetches data blocks from lower memory levels before the data is actually needed. While prefetching can be very effective for improving cache access time, inaccurate prefetching can cause cache pollution, increase memory bandwidth contention, and in effect, degrade the cache's performance and energy efficiency [6], [7]. Apart from determining the right prefetch targets, the prefetch distance must also be well-monitored such that it maintains good prefetch accuracy [7]. This is especially important in reduced retention STTRAMs, which, our analysis show, exhibit different locality behaviors than traditional SRAM caches due to cache block expiration.

In this paper, as an important first step towards understanding prefetching across the STTRAM cache hierarchy, we study data prefetching in the context of a reduced retention L1 STTRAM cache—simply referred to hereafter as 'STTRAM cache'. We assume an STTRAM cache that features the ability to adapt to different applications' retention time requirements (e.g., [5], [4]). We focus on the potentials of data prefetching for improving STTRAM cache's energy efficiency. To motivate this study, we performed extensive experiments using a variety of SPEC 2017 benchmarks and a PC-based stride prefetcher that prefetches memory addresses based on the current program counter (PC) [8]. We observed that if earlier prefetched data blocks are expired because of the reduced retention time, a conventional prefetcher would not reload these blocks. However, a prefetcher could be modified to reload these blocks, thereby reducing the miss penalty caused by premature expiration of blocks (i.e., *expiration misses* [9]). Furthermore, the low write energy in reduced retention STTRAM also mitigates the negative impact of writing blocks in addition to demand requests. We also observed that common metrics for determining the best retention time during runtime (e.g., cache miss rates [4]) may not be accurate in the presence of a prefetcher and can unnecessarily waste energy. As such, prefetching, if carefully designed in the context of reduced

retention STTRAMs, can increase energy savings as compared to prior reduced retention STTRAM design techniques, without incurring significant latency overheads.

Based on the above observations, we propose a new metric, which we call *expired_unused_prefetches*, to evaluate the quality of a current retention time and prefetch distance. The *expired_unused_prefetches* represents the number of prefetched blocks that were not accessed by a demand request before expiry. Using this metric, we developed *Prefetch-Aware Retention time Tuning (PART)* and *Retention time-based Prefetch Control (RPC)*. During a brief runtime profiling phase for each application, PART uses the ratio of *expired_unused_prefetches* to total prefetches to determine if the current retention time suffices for the application. The retention time selected by PART indicates the average amount of time for which cache blocks used by an application reside in the cache. As such, if too many prefetches are expired without being used, it is likely that those prefetches were inaccurate. RPC uses this idea to map *expired_unused_prefetches* to the prefetch distance.

Our major contributions are summarized as follows:

- We study prefetching in STTRAM caches and propose a metric—*expired_unused_prefetches*—that can be used to effectively determine both retention time and prefetch distance, without the need for any complex hardware overhead.
- Using *expired_unused_prefetches*, we proposed an algorithm to determine retention time and prefetch distance during runtime.
- Compared to a base state-of-the-art reduced retention time STTRAM cache, PART+RPC reduced the average energy and latency by up to 22.24% and 24.59%, respectively. Furthermore, when the base architecture was augmented with the state-of-the-art near-side prefetch throttling (NST) prefetching, our approach reduced the average energy and latency by 3.50% and 3.59%, respectively, and substantially reduced the hardware overhead by 54.55%.

II. Background and Related Work

STTRAM's basic structure, comprising of magnetic tunnel junction (MTJ) cells, and characteristics have been detailed in prior work [10]. Earlier works suggest the use of very short retention times (e.g., 26.5 μs [5]) with a DRAM-style refresh scheme for cache implementation [5], [3]. Recent works show that adapting a set of pre-determined retention times to applications' needs, specifically the cache block lifetimes, can further improve energy consumption [4], [11]. In this section, we present a brief overview of prior work on adaptable retention time STTRAM cache—the architecture on which we build the analysis presented herein—and an overview of prefetch distance control.

A. Adaptable Retention Time STTRAM Caches

Recent optimizations on STTRAM cache exploit the variable cache block needs of different applications for energy minimization. For example, Sun et al. [5] proposed a multi-retention time cache featuring various retention times enabled by various MTJ designs, wherein different applications could be run on the retention time that suits them best. More recently, Kuan et al. [4] analyzed the retention times of different applications and proposed a logically adaptable retention time (LARS) cache [4] that used multiple STTRAM units with different retention times. LARS involves a hardware structure that samples the application's characteristics during its very first run. Based on the applications' retention time requirements, each application is executed on the retention time unit that best satisfies their retention time needs. In this paper, we assume a similar multi-retention time architecture to LARS. For brevity, we direct readers to [4] for additional low-level details of the architecture, but omit those details herein.

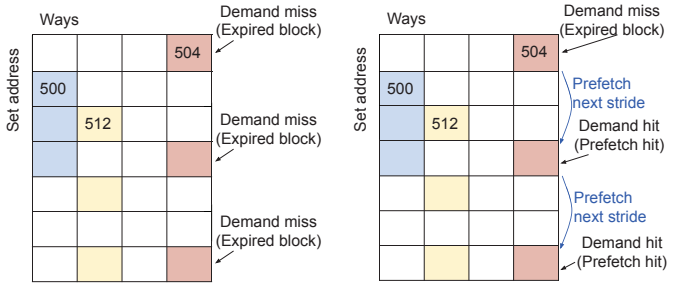
B. Prefetch distance control

Prefetch distance refers to how far into a demand miss stream that a prefetcher can prefetch [8]. Effective prefetching relies on accurate prefetch addresses and timely arrival of data blocks to hide the latency between processor and main memory. As such, the prefetch distance must not be so short as to generate excessive *late_prefetches* [6] or too long to lose prefetch accuracy [6], [7]. Inaccurate prefetches can cause performance degradation due to the saturation of memory bandwidth and cache pollution. As such, lots of prior works discuss various techniques for controlling prefetch distance, feedback directed prefetching techniques, ways to monitor the number of total prefetches and late prefetches to evaluate prefetch accuracy and lateness, and how to determine the prefetcher aggressiveness. For example, Ebrahimi et al. [12] proposed a rule-based control method to separate global throttling and local throttling, and reduce inter-core interference. Both [12] and [6] looked at the number of useless prefetches, which is determined by prefetches that are not used before they are evicted. Heirman et al. [7] referred to the aforementioned methods as *farside* throttling, since they maintained high prefetch distance and throttled down when negative effects were observed. Heirman et al. [7] proposed near-side prefetch throttling (NST), which monitored the ratio of late prefetches and total prefetches, kept prefetch distance low and only raised the distance if necessary. None of these techniques, however, have considered prefetching in STTRAM caches. As we show in our analysis herein, state-of-the-art prefetchers may under-perform if simply implemented on STTRAM caches without considering execution characteristics and metrics that are unique to STTRAM caches.

III. Enabling Prefetching in STTRAM Cache

A. Effectiveness of prefetching expired blocks

Expired blocks in STTRAM caches incur misses when a demand request accesses an expired block prior to eviction. We refer to these misses as *expiration misses*, similar to prior work [9]. As the retention time becomes shorter, expiration misses increase, until expiration misses become the majority of misses and essentially disables the cache's ability to exploit



(a) Prefetch disabled (b) Prefetch enabled

Fig. 1: Prefetching expired blocks. In (a) the prefetch does not bring back previously expired blocks into the cache; in (b) the previously expired blocks are brought back into the cache

temporal locality. Given the uniqueness of expiration misses in STTRAM caches, we first studied the impact of prefetching on expired cache blocks. Figure 1 illustrates a simplified diagram of a data cache, with each cell representing a cache block. The horizontal blocks represent the cache ways (four ways in total) and the vertical blocks represent the set address (seven set addresses in total). The blocks' colors represent the prefetch stream that brought the cache blocks into the cache. We used the stride prefetcher [8] as the base to illustrate our idea and in our experiments. The number associated with the color represents the program counter (PC) value of the load/store instruction that begins the stream due to a demand miss.

Figure 1a illustrates the STTRAM cache without prefetching expired blocks. Assume that the instruction at PC 504 brought three cache blocks into the cache. Since the blocks are brought in by the same stream, they are likely to expire around the same time. If the prefetcher is disabled on those expired blocks, as in a conventional prefetcher, when the demand request accesses the blocks again, loading each block will incur the miss penalty due to expiration misses. Alternatively, enabling the prefetcher for the expired blocks can have a positive effect, since, as shown in Figure 1b, the prefetcher brings in subsequent blocks after the first demand miss (expiration miss). Thus, subsequent accesses to the prefetched blocks become demand hits without exposing the memory latency.

To quantify the benefits of prefetching expired blocks, we performed experiments using SPEC CPU 2017 *rate (_r)* benchmarks and evaluated the energy and latency changes. We used a base stride prefetcher of prefetch distance 16, similar to [13] and considered retention times from 25μs to 1ms. Our detailed simulation setup is described in Section IV-A. We use the term *prefetchable expired blocks* to represent expired blocks that can be accurately predicted and reloaded through the stride prefetcher, and therefore would incur no expiration miss. Figure 2 shows the percentage of prefetchable expired blocks in total expired blocks across the benchmarks, assuming the best retention times. On average across all benchmarks, 10.85% of expired blocks can be reloaded into the cache for reuse. Depending on the applications' access pattern and cache block lifetimes, the reused expired blocks can be as high as 29.66% for *leela*, while over the half of benchmarks (13 of 21) have reuse rates over 10%. To further illustrate this behavior, Figure 3 shows the percentage of prefetchable expired blocks in total expired blocks for different retention times. For brevity, the geometric mean is shown for the different retention times. In general, the percentage of reused expired blocks increases as the retention time decreases, with the highest being 8.69% at 25 μs. These analysis motivate us to explore low-overhead techniques for prefetching *and* determining the best retention time in STTRAM caches during runtime.

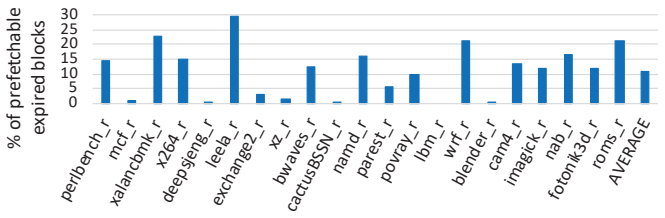


Fig. 2: Percentage of prefetchable expired blocks in total expired blocks across SPEC CPU 2017 benchmarks

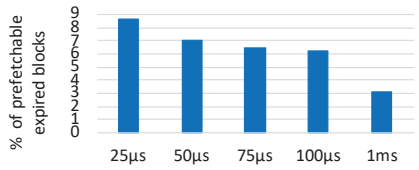


Fig. 3: Percentage of prefetchable expired blocks in total expired blocks for different retention times for SPEC CPU 2017 benchmarks (Geometric mean is shown for brevity)

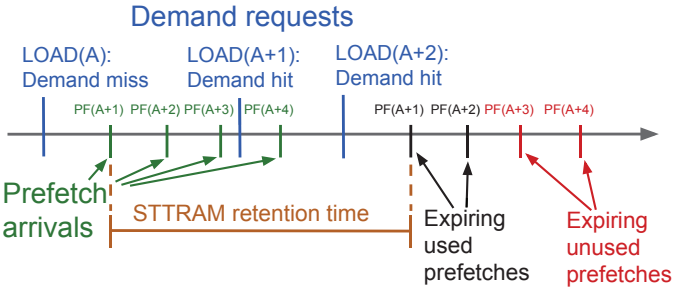


Fig. 4: Retention time expiration detect potentially unused prefetches

B. Prefetch-Aware Retention time Tuning (PART)

A key point of our analysis so far is that, as illustrated in Figure 1b, expiration of cache blocks must be considered in the design of prefetchers. Furthermore, we also analyzed prior adaptable retention time techniques (e.g., [4]) that used miss rates to predict the best retention time. We found that these techniques only accurately predicted the best retention time using cache miss rates in the absence of a prefetcher. When a prefetcher is introduced, using miss rates may not be as accurate due to the interplay of expiration misses and prefetching. Thus, we designed the *prefetch-aware retention time tuning (PART)* technique to take into account the expiration misses.

Algorithm 1: Prefetch-Aware Retention Time Tuning

Data: Retention time set $R = \{25\mu s, 50\mu s, 75\mu s, 100\mu s, 1ms\}$
Result: OutputRetentionTime

```
1  OutputRetentionTime ← 1ms;
2  foreach r ∈ R do
3      allPF ←totalPrefetches(r)/
        totalMSHRRequests(r);
4      expiredPF ←
        expiredUnusedPrefetches(r)/
        totalPrefetches(r);
5      if allPF > 0.1% then
6          if baseExpiredPF is set then
7              if expiredPF < 2*baseExpiredPF then
8                  OutputRetentionTime ← r;
9              end
10             else
11                 return OutputRetentionTime;
12             end
13         end
14         else
15             OutputRetentionTime ← r;
16             if expiredPF > 0.02% then
17                 baseExpiredPF ← expiredPF;
18             end
19         end
20     end
21     else
22         OutputRetentionTime ← r;
23         missBasedTuning(OutputRetentionTime);
24         return OutputRetentionTime;
25     end
26 end
27 return OutputRetentionTime;
```

To motivate PART, Figure 4 illustrates the timeline of when prefetched blocks are brought into the cache and then expired. Assume that `LOAD(A)` instruction accesses memory address *A* and causes a demand miss, the prefetcher sends out four requests from address *A+1* to *A+4*. The prefetch arrival times are marked in green color. After the retention time elapses, prefetched blocks begin to expire. We record the number of blocks that were not used by demand requests before expiration; we refer to these blocks as *expired_unused_prefetches*. The basic idea of PART is to use the shortest retention time that does not excessively increase the *expired_unused_prefetches*. To this end, PART tracks the changes in *expired_unused_prefetches* at prefetch degree 1 during different tuning intervals to determine the best retention time.

Algorithm 1 depicts the PART algorithm, which takes as input the available retention times in the system and outputs the best retention time. PART iterates through the available retention time set starting from the longest to the shortest (e.g., 1ms to 25μs), runs the application for a sampling period, and takes the ratio of total prefetches to total MSHR requests (*allPF*) and the ratio of *expired_unused_prefetches* to total prefetches (*expiredPF*), as shown in lines 3-4. If *allPF* is smaller than 0.1%, we infer that prefetches do not substantially contribute to memory traffic. Therefore, the algorithm switches to a subroutine that predicts the retention time based on cache misses, similar to prior techniques [4] (line 23). If *allPF* is greater than 0.1%, the algorithm first checks if *expiredPF* is significant enough ($>$ 0.02%). If *expiredPF* is greater than 0.02 %, this *expiredPF* is stored as *baseExpiredPF* and used in subsequent tuning stages. Otherwise, PART iterates the next available retention times to see if the thresholds are satisfied (line 15-18). Note that we determined the thresholds empirically through extensive experiments and analysis. After obtaining *baseExpiredPF*, PART explores shorter retention times to find the one that does not excessively increase *expiredPF* as compared to *baseExpiredPF*. PART checks if *expiredPF* is smaller than twice *baseExpiredPF*. If so, it proceeds to the next shorter retention time, otherwise, the current retention time is returned as the tuning result (line 7-12).

C. Retention Time-based Prefetch Control (RPC)

We also developed a simple heuristic, called *retention time-based prefetch control (RPC)*, that works in conjunction with PART to determine the best prefetch distance during runtime. To minimize tuning overhead, RPC determines the best prefetch distance in 'one-shot' along with the retention time tuning by the PART algorithm. PART tracks *expired_unused_prefetches* at prefetch degree 1 for tuning the retention time. The determined retention time represents the period that suffices, on average, for the executing applications' cache block lifetimes. A prefetch degree of 1 is usually considered conservative in prefetch distance throttling [6], [12]. As such, if *expired_unused_prefetches* is excessively high after retention time tuning, it is likely that wrong addresses were prefetched. In this case, we maintain the prefetch distance of 1 to minimize cache pollution and memory bandwidth contention. RPC takes *expiredPF* in Algorithm 1 as input to determine the prefetch aggressiveness, and maps the prefetch distance similarly to [6]. Table I shows the distribution of this mapping, representing different ranges of *expiredPF* and the associated prefetch distance. If *expiredPF* is above 5%, the stride pattern does not match the current application's data access. Thus, the prefetch distance is kept at 1 in order to maintain prefetch functionality. On the other extreme, we observed that some applications have the lowest *expiredPF* (and energy consumption) at prefetch distance 32, which indicates that the stride prefetcher captures the applications' data access pattern and is able to recover expired blocks.

D. Overhead

Assuming a base architecture that has the capability of multiple retention times (e.g., [4]), PART's major advantage is that it imposes negligible hardware and tuning overhead.

TABLE I: Prefetch distances for different *ExpiredPF*

ExpiredPF at prefetch degree 1	Prefetch distance
Above 5%	1
1.01% - 5%	4
0.51% - 1%	8
0.05% - 0.5%	16
Below 0.05%	32

PART exploits most of the hardware components described in [4] for tuning. In addition to the four 32-bit registers and one division circuit used in prior work, PART only requires one additional 32-bit register for *allPF* and *expiredPF*. To keep track of expired unused prefetches, PART only requires one custom hardware counter, which increments when an expiring block's prefetch bit is valid. Using the shared *expiredPF* in PART, RPC requires only one 32-bit comparator. In total, we estimate that the area overhead is less than 1% of modern processors like ARM Cortex-A72 [13].

We note that the base architecture incurs energy and latency switching overheads from migrating the cache state from one STTRAM unit to another. Switching occurs when an application is first executed during its sampling period. For example, given a tuning interval of 10 million instructions and five retention time options, sampling would require 50 million instructions. However, PART does not increase the switching overhead with respect to the base. In the worst case, each migration takes approximately 2560 cycles and 8.192nJ energy, resulting in total time and energy overheads of 10240 cycles and 32.768nJ, respectively. While these overheads are minimal in the context of full application execution, we reiterate that PART did not contribute to this overhead.

IV. Simulation Results

A. Experimental Setup

To perform our analysis and evaluate PART, we implemented PART using an in-house modified[1] version of the GEM5 simulator [14]. We modified GEM5 [14] to model cache block expiration, variable tag lookup and cache write latency, variable retention time units, and variable prefetch distance as described herein. To enable rigorous comparison of PART against the state-of-the-art, we used two recent prior works to represent the state-of-the-art—LARS [4] to represent adaptable retention time and NST [7] to represent variable prefetch distance. We also implemented these two techniques in GEM5. We used configurations similar to the ARM Cortex A72 [13], featuring a 2GHz clock frequency, and a private L1 cache with separate instruction and data caches. For this work, we focused on data cache prefetching, since it provides much opportunity for runtime adaptability, as opposed to the instruction cache [4]. Every MSHR request from the L1 data cache is directly sent to an 8GB main memory, and incurs memory latency. We intend to explore the impact of our work on the instruction cache and lower level caches in future work.

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

We considered five retention times: 25μs, 50μs, 75μs, 100μs, 1ms, which we empirically found to be sufficient for the considered benchmarks. We used the MTJ modeling techniques proposed in [15] to model the different retention times, and used NVSim [16] to estimate the energy for the different retention times. Table II depicts prefetcher configurations and the STTRAM cache parameters used in our experiments as obtained from the modeling tools and techniques. We used twenty-one SPECrate CPU2017 benchmarks [17], cross-compiled for the ARMv8-A instruction set architecture. Each benchmark was run using the *reference* input sets for 1B instructions after restoring checkpoints from 240B instructions.

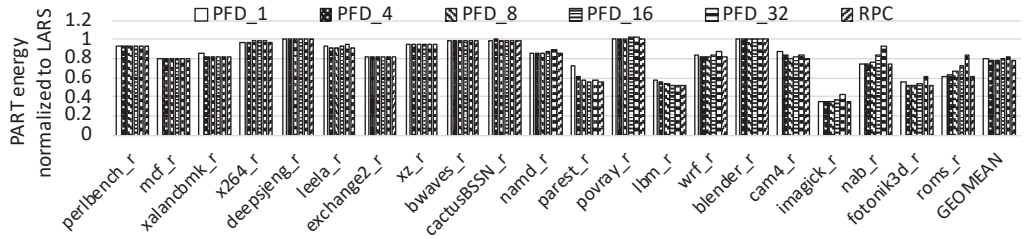
B. Results and Comparisons

In this section, we compare the cache energy and access latency benefits of our work to prior work in various prefetch distance control scenarios. We denote uniform prefetch distance 1 to 32 as *PFD_N*, where N represents the memory address distance. RPC represents the optimal static distance among PFD_N, since RPC accurately determines the distance in the sampling phase and uses that distance throughout the application's run. We use NST [7] to represent the state-of-the-art dynamic prefetch distance throttling. We compare PART to the miss-based tuning algorithm used in LARS. We start with a direct comparison of PART to LARS without prefetching. Next, we compare PART to LARS with a uniform stride prefetcher and use moderate prefetch aggressiveness: prefetch degree 2 and prefetch distance 16 (LARS+PFD_16), similar to prior work [6]. Thereafter, we compare PART to LARS with the NST stride prefetcher (LARS+NST) to evaluate the improvement over dynamic prefetch distance throttling. Lastly, we compare PART to an SRAM cache with the NST stride prefetcher (SRAM+NST) to show the collective improvements of adaptable retention time STTRAM cache when prefetching is active. All energy and latency results of PART are normalized to the subject of comparison.

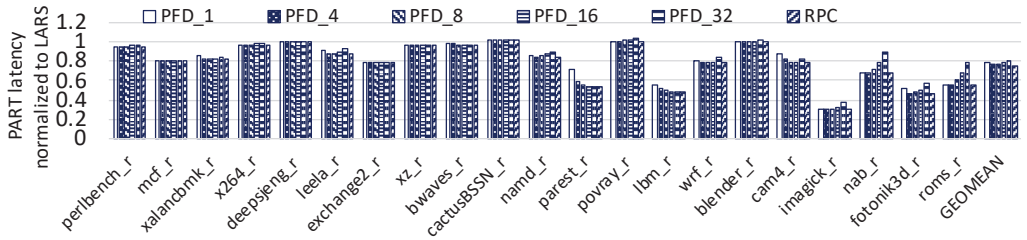
1) Comparison to the base STTRAM cache (LARS): Figure 5a depicts the energy consumption of PART in different prefetch distance scenarios normalized to LARS. On average across all benchmarks, PART reduced the energy by 19.53%, 21.25%, 21.29%, 20.09%, and 17.68% for PFD_1, PFD_4, PFD_8, PFD_16, and PFD_32, respectively. RPC properly mapped expired unused prefetches (*expiredPF*) to prefetch distance and ensured that the ideal static prefetch distance was selected. As such, PART+RPC reduced the average energy by 22.24%, with savings as high as 65.96% for *imagick*. For *parest*, *imagick*, *lbm*, *roms*, and *fotonik3d*, PART+RPC reduced the energy by more than 40%, and no benchmarks' energy consumption was degraded by PART. Figure 5b depicts the cache access latency normalized to LARS without prefetching. On average across all benchmarks, PART reduced the latency by 21.52%, 23.50%, 23.51%, 22.08%, 19.29%, and 24.59% for PFD_1, PFD_4, PFD_8, PFD_16, PFD_32, and RPC, respectively. PART+RPC reduced the latency by up to 70.41% for *imagick*. PART only incurred a negligible latency

TABLE II: Prefetcher configuration and STTRAM cache parameters with different retention times

Prefetcher Configuration	Type: stride prefetcher, degree: 4, adaptable prefetch distance: 1, 4, 8, 16, 32					
Cache Configuration	32KB, 64B line size, 4-way, 22nm technology					
Memory device	SRAM	STTRAM-25μs	STTRAM-50μs	STTRAM-75μs	STTRAM-100μs	STTRAM-1ms
Write energy (per access)	0.002nJ	0.006nJ	0.007nJ	0.007nJ	0.008nJ	0.011nJ
Hit energy (per access)	0.008nJ			0.005nJ		
Leakage power	75.968mW	11.778mW	11.778mW	11.778mW	11.778mW	11.365mW
Hit latency (cycles)	2			1		
Write latency (cycles)	2	2	3	3	3	4



(a) Energy



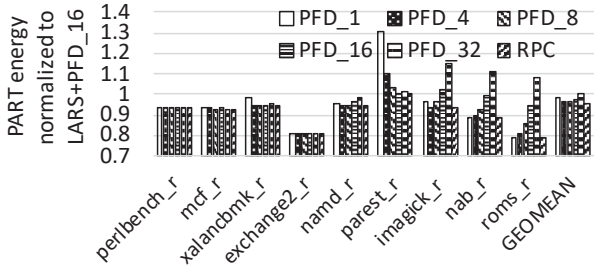
(b) Latency

Fig. 5: PART with different prefetch scenarios (PFD_N and RPC) normalized to the base STTRAM cache (LARS)

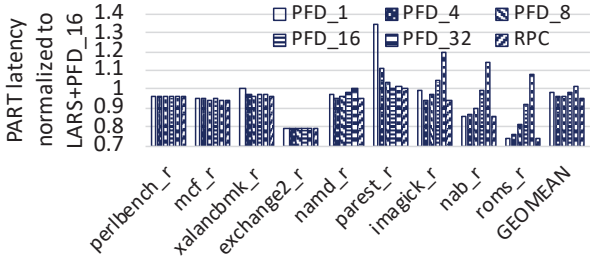
overhead (1.07%) for *cactusBSSN* while latency reductions were achieved for the rest of the twenty benchmarks.

Compared to LARS, we observed that the energy reduction trends were similar to the latency. Since prefetching can reduce compulsive misses, increased latency benefits are achieved as a result of the impact of expiration misses as discussed in Section III-A. As shown in Figure 2, the average expired blocks that can be accurately prefetched and 'reused' are up to 10.85%. Thus, the reduced expiration misses contributed significantly to miss latency reduction.

2) Comparison to LARS with uniform prefetch distance (LARS+PFD_16): Figure 6 depicts the energy and latency of PART normalized to LARS+PFD_16. For brevity, only the geometric mean (across all the twenty-one benchmarks in Figure 5) and a subset of notable benchmarks are shown. Figure 6a shows that across all the benchmarks, PART+RPC reduced the average energy consumption by 4.75%, compared to LARS+PFD_16 (the uniform prefetch distance). PART+RPC reduced the energy by up to 20.51% and 18.77% for *roms* and *exchange2*, respectively, with energy savings over 5% for *perlbench*, *mcf*, *xalancbmk*, *namd*, *nab*, and *imagick*. We observed that PART generally selected shorter retention times than LARS+PFD_16. By incorporating the expiration misses into the decision making about prefetching, PART achieved a balance of short retention times without translating into increases in miss latency. PART allowed the stride prefetcher to recover expired blocks in short retention times. PART+RPC



(a) Energy



(b) Latency

Fig. 6: PART with different prefetch scenarios (PFD_N and RPC) normalized to LARS+PFD_16

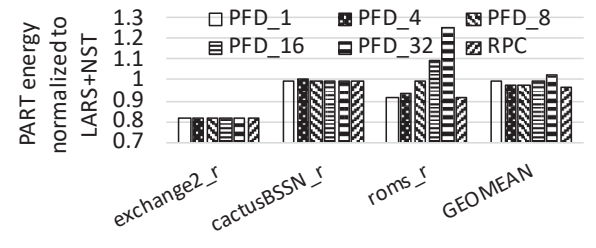
only degraded the energy (by 0.48%) for *parest*.

As described in Section III-A, due to the reduced latency achieved by prefetching expired blocks, PART uses shorter retention times to improve energy consumption, since the

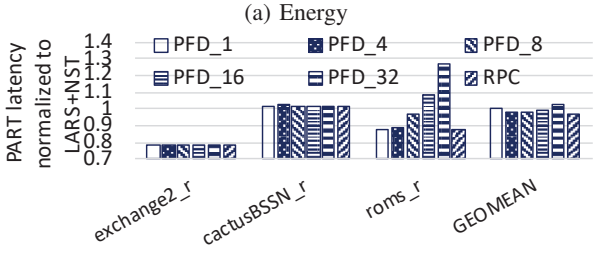
short retention times do not substantially increase the latency. Figure 6b shows that, similar to the energy improvement, PART+RPC reduced the average latency by 4.99%, as compared to LARS+PFD_16. PART+RPC reduced the latency by up to 25.76%, 21.09%, and 14.15% for *roms*, *exchange2*, and *nab*, respectively. To understand why PART performed so well for these benchmarks, we studied their execution more closely. For *exchange2*, we observed that LARS selected a long retention time (1ms) due to low miss rates at 1ms, whereas shorter retention times increased the miss rates substantially (by up to 9x). However, the large amounts of misses at shorter retention times were rapidly amortized by stride prefetching and did not have substantial negative impact on the latency. We observed that even though shorter retention times increased *totalPrefetches* for expired blocks, the *expired_unused_prefetches* increased at a much slower rate, thereby substantially reducing *expiredPF* (by up to 42%). As such, PART selected short retention times (e.g., 25μs) and was able to improve the latency for these benchmarks.

On the other hand, for *roms*, LARS selected a short retention time of 25μs due to the low miss rates. However, the *expiredPF* were substantially higher at shorter retention times than 1ms. As such, PART selected 1ms for *roms* to save potentially useful prefetches with the longer retention time. The reduced latency in *nab* resulted from the optimal prefetch distance (at PFD_1) as determined by RPC. These results illustrate the importance of adaptable prefetch distance to satisfy different applications' needs. PART incurred minor latency overheads of up to 1.6% and 0.19% for *cactusBSSN* and *parest*, but reduced the latency for majority of the benchmarks (19 of 21).

3) Comparison to LARS with dynamic prefetch distance (LARS+NST): We further compared PART with LARS+NST to evaluate the improvement when the dynamic prefetch throttling is enabled as in previous work [7]. For brevity, Figure 7 compares PART to LARS+NST using a subset of notable benchmarks and the geometric mean of all the benchmarks. Figure 7a shows that on average, PART+RPC improved the energy by 3.50% over LARS+NST, with energy savings of up to 18.77% for *exchange2*. On the other hand, on average, LARS+NST only improved over LARS+PFD_16 by 1.43%. We observed that in STTRAM cache without PART, the dynamic prefetcher (NST) offered minimal energy savings, even if it recovered expired blocks. As shown in Figure 7b, PART+RPC reduced the average latency by 3.59% compared to LARS+NST, with reductions of up to 21.09% and 12.23% for *exchange2* and *roms*, respectively. In the worst case, the latency overhead was 1.60% for *cactusBSSN*, while the rest of benchmarks benefited from latency reduction.



(a) Energy



(b) Latency

Fig. 7: PART with different prefetch scenarios (PFD_N and RPC) normalized to LARS+NST

In a few cases, PART+RPC did not improve the latency or energy as compared with LARS+PFD_16 or LARS+NST (for example, for *cactusBSSN*). *CactusBSSN* was one of the benchmarks with a low prefetch percentage in total MSHR requests. As defined in Algorithm 1 (line 3), the *allPF* in *cactusBSSN* was very low at 0.0002%. Thus, PART reverts to miss based tuning for *cactusBSSN*, as described in Section III-B. However, to provide a clear contrast between our work and prior work, we used *expiredPF*-based tuning in all PART+RPC results. For *cactusBSSN*, the RPC table was unable to map the correct prefetch distance for latency or energy improvement. We note, however, that in almost all cases (20 out of 21 benchmarks), PART+RPC outperformed both LARS+PFD_16 and LARS+NST in both energy and latency. Importantly, we also reiterate that LARS+NST required additional hardware structures to implement the NST prefetcher, whereas RPC's overhead was marginal compared to LARS+PFD_16, as described in Section III-D. The main advantage of PART+RPC is the negligible hardware overhead compared to NST. For instance, NST required seven 32-bit registers for storage [7], whereas PART only introduced one additional register to LARS in order to track the number of outgoing MSHR requests, total prefetches, and expired unused prefetches. Overall, PART+RPC reduced the implementation overhead by 54.55% compared to LARS+NST.

4) Exploring the synergy of PART and NST: We also explored the extent of the benefit, if any, of combining PART with NST (i.e., PART+NST). Figure 8 summarizes the energy and latency of PART+RPC and PART+NST normalized to LARS+NST. For brevity, only the geometric mean of all the SPEC CPU 2017 benchmarks are shown. On average, PART+NST improved the energy and latency by 2.75% and 2.63%, respectively, compared to LARS+NST, whereas PART+RPC reduced the energy and latency by 3.50% and 3.59%, respectively. The results show that while providing dynamic prefetch distance control, NST's increased hardware overhead compared to PART does not translate to energy or latency benefits. In fact, PART still reduced the energy and latency, albeit marginally, while substantially reducing the implementation overheads (Section IV-B3). The results also reveal the promise of a low overhead dynamic prefetch distance control for STTRAM cache based on *expiredPF*. We anticipate that even more energy and latency benefits can be

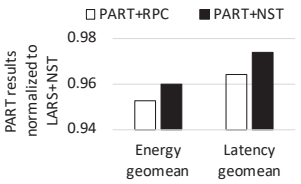


Fig. 8: PART normalized to LARS+NST

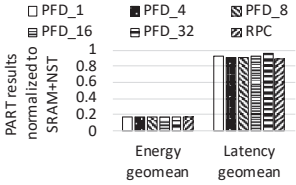


Fig. 9: PART normalized to SRAM+NST

achieved in larger STTRAM caches (such as LLC), and we intend to explore and quantify these benefits in future work.

5) Comparison to SRAM with dynamic prefetch distance (SRAM+NST): We also compare PART to SRAM cache with NST prefetcher enabled (SRAM+NST). Figure 9 summarizes the energy and latency of PART in the different configurations normalized to SRAM+NST. On average, in all prefetch configurations, PART reduced the energy by more than 80%. We attribute this reduction largely to the STTRAM's low leakage power (Table II) and PART's ability to select retention times that satisfied the different applications' cache block requirements. As a result of this specialization, PART was also able to reduce the latency (e.g., by 10.28% for PART+RPC). As shown in Table II, STTRAM has advantages in hit latency but not write latency. However, with the help of PART, STTRAM was able to select shorter retention times that satisfy the applications' needs while maintaining write latencies that were close to SRAM. We took a closer look at benchmarks with high write activity, where write requests and miss responses were greater than 40%, such as *perlbench*, *cactusBSSN*, *povray*, *lbm*, *cam4*, and *fotonik3d*. Our analysis revealed that the synergy of prefetching and PART's retention time selection made the write performance for these benchmarks comparable to SRAM. As a result, the STTRAM cache with PART did not degrade the latency compared to SRAM.

V. Conclusions and Future Work

In this paper, we studied prefetching in reduced retention STTRAM L1 caches. We showed that using *expired_unused_prefetches*, and practically, tracking changes in expired prefetches (*expiredPF*) with respect to total prefetches (*allPF*), we could provide an accurate description of the best retention with regards to energy consumption and derive insights into the best prefetch distance. Based on these insights, we proposed prefetch-aware retention time tuning (PART) and retention time based prefetch control (RPC) to predict the best retention time and the best prefetch distance during runtime. Experiments show that PART+RPC can reduce the average cache energy and latency by 22.24% and 24.59%, respectively, compared to a base architecture, and by 3.50% and 3.59%, respectively, compared to prior work, while reducing the implementation hardware overheads by 54.55%. For future work, we plan to explore the implications of PART on shared lower level caches and in the presence of workload variations.

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] J. Ahn, S. Yoo, and K. Choi, "Dasca: Dead write prediction assisted stt-ram cache architecture," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 25–36.

[2] N. Sayed, R. Bishnoi, F. Oboril, and M. B. Tahoori, "A cross-layer adaptive approach for performance and power optimization in STT-MRAM," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 791–796.

[3] 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.

[4] K. Kuan and T. Adegbija, "Energy-Efficient Runtime Adaptable L1 STT-RAM Cache Design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 6, pp. 1328–1339, 2020.

[5] 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.

[6] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, Feb 2007, pp. 63–74.

[7] W. Heirman, K. D. Bois, Y. Vandriessche, S. Eyerman, and I. Hur, "Near-Side Prefetch Throttling: Adaptive Prefetching for High-Performance Many-Core Processors," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '18. New York, NY, USA: Association for Computing Machinery, 2018.

[8] Tien-Fu Chen and Jean-Loup Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609–623, May 1995.

[9] D. Gajaria and T. Adegbija, "Arc: Dvfs-aware asymmetric-retention stt-ram caches for energy-efficient multicore processors," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 439–450. [Online]. Available: https://doi.org/10.1145/3357526.3357553

[10] 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.

[11] K. Kuan and T. Adegbija, "HALLS: An Energy-Efficient Highly Adaptable Last Level STT-RAM Cache for Multicore Systems," *IEEE Transactions on Computers*, vol. 68, no. 11, pp. 1623–1634, Nov 2019.

[12] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated Control of Multiple Prefetchers in Multi-Core Systems," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: Association for Computing Machinery, 2009, p. 316–326.

[13] "ARM Cortex-A72 MPCore Processor Technical Reference Manual Revision r0p3 Revision r0p3 Documentation." [Online]. Available: https://developer.arm.com/docs/100095/0003

[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] 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.

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

[17] "SPEC CPU® 2017." [Online]. Available: https://www.spec.org/cpu2017/