A Study of STTRAM-based Page Walker Caches for Energy-Efficient Address Translation

Kyle Kuan and Tosiron Adegbija
Department of Electrical & Computer Engineering
University of Arizona, Tucson, USA
ckkuan@email.arizona.edu; tosiron@arizona.edu

***Abstract*—This paper studies spin-transfer torque RAM (STTRAM) as an energy-efficient alternative to SRAM for implementing page walker caches (PWCs) in resource-constrained systems' memory management units. We analyze the access characteristics and persistence of PWC blocks in a set of multithreaded workloads, revealing that individual threads might have different runtime behaviors. Given this observation, we explore and analyze the benefits of *heterogeneous retention time STTRAM*—wherein STTRAM's data retention time is specialized to cache blocks' persistence needs—for implementing the PWC in multicore systems. Based on our analysis, we propose *NECTAR*, an energy-efficient heterogeneous STTRAM-based page walker cache architecture. Experimental results using multithreaded PARSEC benchmarks show that NECTAR enables runtime adaptability and offers substantial energy benefits for implementing the PWC, reducing the average energy by 81.36% compared to SRAM, without introducing significant overheads.**

***Index Terms*—Page walker cache, energy-efficient address translation, spin-transfer torque RAM cache**

I. Introduction

An important component of modern processors, with critical implications for performance, is the memory management unit (MMU) [1], [2]. The MMU is a hardware unit that performs virtual address translation, and contains a translation look-aside buffer (TLB), a page walker, and *page walk caches (PWCs)*. The TLB is a small data structure that caches the most recently used page table entries (PTEs) of address translations. However, due to dynamic runtime applications, virtual addresses can exceed the TLB's reach, resulting in TLB misses, and causing the page walker to initiate expensive *page walks*. The page walks involve fetching the desired PTE on the page table stored in the main memory.

The PWC can substantially mitigate the cost of page walks by caching page table entries [3]. Prior works have proposed various page table schemes to efficiently organize PTEs and PWCs [3]. However, most prior PWC research focus on high-performance computing where energy efficiency is often not prioritized. While PWCs' main goal is improving performance, the energy consumption is increasingly important, especially in resource-constrained systems (like mobile devices, embedded systems) [2]. Depending on the design, the PWC can contribute $\sim$10% to the processor's overall energy consumption. This paper aims to bridge this gap in the state-of-the-art by focusing on energy in exploring *spin-transfer torque RAM (STTRAM)* as an alternative to traditional SRAM for implementing PWCs. STTRAM is a non-volatile memory (NVM) technology that has emerged as one of the top contenders for replacing traditional technologies like SRAM and DRAM in emerging architectures. STTRAM offers several advantages, such as low leakage power, higher density than SRAM, compatibility with CMOS, etc [4]. However, STTRAM also introduces important challenges that must be addressed for it to be effectively used for PWCs.

978-1-6654-3922-0/21/$31.00 ©2021 IEEE

Some of STTRAM's drawbacks include its long write latency and high write energy [5], [6]. These challenges are partly attributed to STTRAM's long *retention time*—the duration of data retention in the absence of power. However, we observed that PWC accesses are dominated by read operations; thus, the PWC is less likely than a traditional data cache to be negatively impacted by STTRAM's write latency and energy overheads. Figure 1 illustrates this observation using the percentage of reads and writes in the PWC accesses for eleven PARSEC 3.0 [7] benchmarks. In all of the applications, write accesses accounted for less than 7% of all memory accesses. Furthermore, like data/instruction caches, PWC blocks only remain in the PWC for short periods of time ($<$ 1s). Thus, STTRAM's retention time can be substantially reduced and specialized to PWC access patterns to mitigate the write energy/latency overheads. In this work, we leverage reduced retention STTRAM for implementing energy-efficient PWC.

Through detailed experiments and analysis, this paper derives new insights into: (1) the block sharing capability enabled by a shared PWC in comparison to private per-core PWCs; (2) how block sharing among different cores affects the impacts of prematurely expired data blocks in reduced retention STTRAM PWCs; and (3) the variability of STTRAM retention times across different applications in the context of page walk and the PWC. Based on our analysis, we propose an energy-efficient STTRAM-based page walker cache architecture, called *NECTAR*. NECTAR features heterogeneous retention STTRAM cache ways and a new least recently used (LRU) retention time-aware replacement policy—*LRU-RT*—

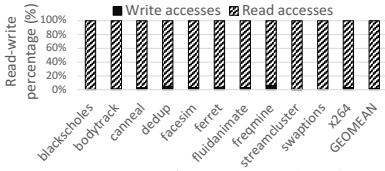


Fig. 1: Percentages of PWC read/write accesses

that uses hardware performance counters to determine the best retention time to service core requests. To the best of our knowledge, we are the first to take energy efficiency into consideration in the PWC design space. We are also the first to adopt STTRAM for implementing PWCs and consider STTRAM's retention time in PWC's replacement policy.

In summary, this paper makes the following contributions:

- We analyze the access footprints and persistence of PWC blocks in multithreaded workloads, in the context of STTRAM, and explore the use of reduced retention time STTRAM in PWCs to achieve low-overhead energy benefits compared to SRAM.
- We propose a novel PWC architecture, *NECTAR*, for energy-efficient page walk. NECTAR features heterogeneous retention STTRAM PWC ways and a replacement policy (*LRU-RT*) that enables the runtime assignment of blocks to the PWC ways that satify the blocks' persistence needs.
- We extensively analyze NECTAR for multithreaded PARSEC benchmarks. Results show that NECTAR can achieve average energy savings of 81.36% compared to SRAM, with minimal overheads. Furthermore, NECTAR specializes the PWC to individual application threads, improving the latency by up to 35.51% compared to uniform retention times.

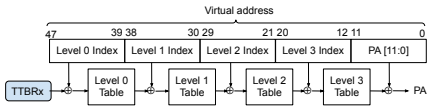
II. Background and Related Work

A. Implementing reduced retention STTRAMs

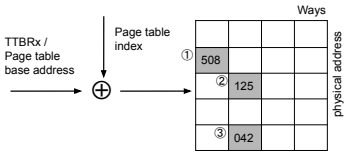
STTRAM has been widely studied, prototyped, and commercially implemented, making it one of the top contenders for implementing caches in resource-constrained systems [8]. STTRAM uses a magnetic tunnel junction (MTJ) cell as the binary storage cell. MTJ contains two ferromagnetic layers separated by an oxide barrier/tunnel layer [4]. Updating the MTJ cell's data bits relies on the magnetization switching of MTJ's free layer [4]. Due to the long switching time of the free layer's magnetization state, it takes more time and energy to write to STTRAM than to SRAM. This overhead can be mitigated by reducing STTRAM's thermal factor, thereby also reducing the *retention time*—the time until a bit-flip occurs and disrupts the data's integrity [9].

B. Adaptable retention STTRAM caches

Recent STTRAM cache optimizations have exploited the variability of applications' cache block needs to minimize energy. For example, Sun et al. [6] proposed a multi-retention STTRAM cache hierarchy with different retention times enabled by different MTJ designs, wherein applications can be run on the cache level with the best retention time for the applications' access patterns. More recently, Kuan et al. [5] analyzed the retention times of different applications and proposed a sampling-based logically adaptable retention time (LARS) cache that used multiple STTRAM units with different retention times. Each executing application is executed on the best retention time unit for its retention time needs. Like prior work, we explore different retention times for different applications. Ours is, however, the first that explores the design of STTRAM for page walk caches (PWC).



(a) Address translation of 4K page in ARM-64 architecture



(b) Translation descriptor stored in page walker cache

Fig. 2: Illustration of a page table walk in the ARM-64 architecture. (a) The page walker walks through all levels of page table entries on a TLB miss; (b) cached translation descriptor can shorten the latency of a page walk

C. Address translation and PWC in ARM-64 architecture

To exploit temporal locality, the memory management unit (MMU) features a TLB to track the most recent address translations. However, to keep up with the processor speed, the TLB size is usually small, and the processor often requests translations outside the TLB. When a TLB miss occurs, the page walker must search the page table to find the right entries.

Figure 2a illustrates the address translation process using an ARM-64 page walker [10] as a proxy for resource-constrained systems. The translation table base registers (TTBRx)—the ARM equivalent of x86-64's 'CR3' register—points to the base address of the first level page table. ARM uses TTBR1 to indicate the translation tables for the kernel address space, and TTBR0 for the application's table. The page walker adds the base address taken from TTBR to the level 0 virtual address index (VA[47:39]), obtains the physical address of the level 1 table from level 0, and repeats the process for each level until the desired physical address (PA) is obtained. ARM-64 uses an 8B descriptor format to wrap the page table base address across all levels of the page table [1].

Figure 2b illustrates how a page walker cache stores copies of translation descriptors, which contain the information of the base address. In Figure 2a, the page walker obtains the physical address of the level 1 table's base address by adding the TTBR and level 0 virtual index, and must retrieve the level 1 table's base address. To speed up the process, the memory request for the level 1 table base address can be passed to a PWC, as depicted in Figure 2b. This reference returns the table's base address, represented by the number at label ① (PPN 508). In this case, three tables' base addresses can be retrieved from the PWC (①, ②, and ③), thus reducing the latency overhead caused by accesses to the main memory.

III. STTRAM Page Walker Cache Analysis

We begin by analyzing PWCs in the context of STTRAM and explore the PWC access footprints based on application behavior. We performed the analysis based on the ARM-64 architecture using the PARSEC 3.0 benchmarks. Our detailed experimental setup is described in Section V.

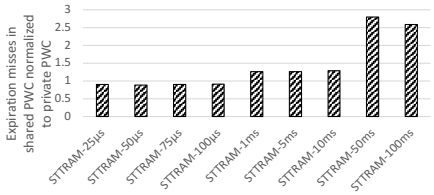


Fig. 3: Expiration misses in the shared PWC normalized to private PWCs. The average is shown for brevity.

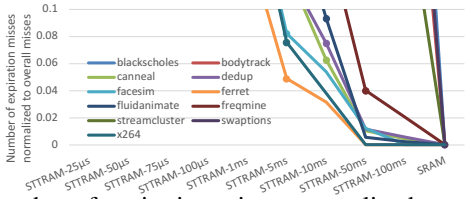


Fig. 4: Number of expiration misses normalized to total misses in the shared PWC. The points indicate how long the PWC blocks generally remain in the cache before eviction.

A. Effectiveness of shared STTRAM PWC

Like prior work [3], our analysis showed that shared SRAM PWC in a multicore system substantially reduces the PWC miss rates compared to private PWCs. We observed similar trends for STTRAM with different retention times. The shared PWC reduced the average miss rates by 40.08%, 40.12%, 36.29%. and 40.42% for 5ms, 10ms, 50ms, and 100ms retention times, respectively. Shared PWCs significantly reduce the miss rate compared to private per-core PWCs due to the substantial reduction in shared block misses. However, due to increased hit rates from block sharing in shared PWCs, the cache blocks must remain in the cache longer, i.e., the *block lifetimes* increase. This observation raises a unique concern to reduced retention STTRAM—*expiration misses*.

B. Expiration misses in STTRAM shared PWCs

Expiration misses occur when a cache block is prematurely expired because the retention time is shorter than the block's lifetime. Subsequent references to that block then result in misses that would not have occurred in an SRAM or non-volatile cache. Figure 3 quantifies the number of overall expiration misses in the shared PWC normalized to the private PWC as the retention time increased. In short retention times (25μs, 50μs, 75μs, and 100μs, most of which were under-provisioned for the block lifetimes), shared PWC reduced the expiration misses compared to private PWC (by 9.46%, 11.70%, 9.24%, and 8.91%, respectively). However, the shared PWC expiration misses *increased* by 26.30% on 1ms, and by more than 2.5X on the 50ms and 100ms retention times. We observed that cache blocks' lifetimes in private PWCs were generally shorter than in shared PWC, due to lack of block sharing in private PWCs. As such, on the longer retention times, even though shared PWC reduced the overall misses, the expiration misses dominated the misses, while most blocks in private PWC had sufficient retention times. This motivated us to further study the expiration misses in shared PWCs, and how to minimize their impacts, in order to fully exploit the benefits of shared PWCs in higher retention times.

C. Sensitivity of shared PWC blocks to retention times

We analyzed the sensitivity of different applications to the PWC's retention time. If diversity exists in the applications' retention time requirements, we can further improve the PWC's efficiency by using heterogeneous retention times that enable a tighter specialization to applications' PWC block lifetimes. Since shorter retention times result in less latency and energy overhead, the idea is to use retention times that are only *as long as needed* for the applications' PWC block needs. On the other hand, if the retention time is inordinately short, it could result in substantial latency and energy overheads due to expiration misses that a shared PWC is unable to mitigate.

Figure 4 depicts the number of expiration misses normalized to the total misses for the PARSEC 3.0 applications. The dots in the figure represent the lowest retention time that incurred expiration misses not exceeding an *expiration miss threshold* of 10% of the total misses. Our analysis revealed that the impact of expiration misses on latency and energy was most significant when the expiration misses exceeded 10% of the total misses. Satisfying this threshold implies that the cache misses are dominated by misses due to regular block replacements (i.e., conflicts) and not due to an insufficient retention time. Therefore, a retention time that satisfies this threshold for an application is long enough for the application's PWC blocks. We repeated this analysis for SPEC 2017 benchmarks and found that the 10% threshold still holds.

We observed that none of the benchmarks satisfied the threshold at any retention time below 1ms because PWC blocks generally have longer lifetimes. Furthermore, there was diversity in the PWC blocks' needs. For instance, whereas 5ms and 10ms satisfied the expiration miss threshold among several applications, some (e.g., *freqmine, streamcluster*) required longer retention times to satisfy the threshold. This diversity motivated us to explore the heterogeneous retention time principle for implementing STTRAM PWCs.

IV. Heterogeneous Retention STTRAM PWC

An ideal STTRAM PWC should dynamically adapt to different applications' needs. However, due STTRAM cell's physical constraints, the retention time cannot be physically changed during runtime. The need for a shared STTRAM PWC precludes the multi-unit design in prior work [5] due to the resultant area overhead. Therefore, we propose *NECTAR*—a *shared heterogeneous retention time STTRAM page walk cache architecture*—that allows accesses from different parallel threads without the area overhead of prior designs [5].

A. Hardware architecture

NECTAR uses ARM's PWC architecture (described in Section II-C) as a baseline. Each address translation descriptor in the ARM architecture is 8B [1]. NECTAR's cache block size is set to 64B to allow the transfer of eight entries per access. Figure 5 presents an overview of NECTAR's design and the details of two hardware counters used in the design:

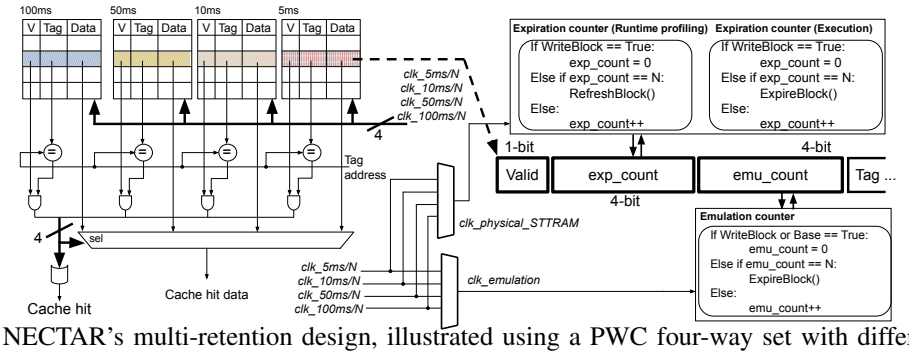


Fig. 5: Overview of NECTAR's multi-retention design, illustrated using a PWC four-way set with different retention times in each way. Details shown of expiration and emulation counters for ensuring data validity and emulating retention times

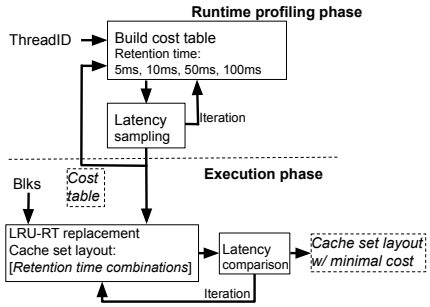


Fig. 6: Overview of LRU-RT replacement policy

the *expiration counter* and *emulation counter*. We describe these two counters in section IV-B1 and IV-B2. The figure also depicts the connection of retention time clocks used by the two counters. The left-hand side of the figure depicts an overview of NECTAR's implementation, illustrated using a base 4-way cache with different retention times in each way. The different retention times—100ms, 50ms, 10ms, 5ms—are determined through analysis of the target applications.

It remains possible that retention times elapse before some data blocks are evicted or invalidated through normal cache accesses. When this occurs, the data blocks can become corrupted. To address this challenge, we used a per-block *expiration counter* and *preemptive block eviction* process to maintain the validity of data and tag memory. The right-hand side of Figure 5 illustrates how NECTAR uses the expiration counter. The expiration counter *exp_count* is implemented as a simple 4-bit counter that increments at every positive clock edge. When a block is written into the PWC, *exp_count* is reset to 0 and increments until the maximum count *N*. Thereafter, the PWC controller preemptively evicts the block by invalidating it (a write-back occurs if the block is dirty). The counter up-counts at a period of $retention\_time/N$, such that the block is evicted just before the retention time elapses.

B. Determining the best configuration: retention time-aware block (re)placement

An important consideration in NECTAR is how to place blocks in the appropriate PWC way during runtime. Figure 6 depicts our approach to orchestrate the block placement process. When the PWC controller receives a request from an unmapped thread ID in a process, the controller enters the *runtime profiling phase*, during which a *CostTable* is created for that thread to keep track of the lowest latency retention time. When a replacement is triggered during the *execution phase*, the cache controller uses an *LRU retention time-aware (LRU-RT)* policy to determine the new block's placement.

1) Runtime profiling phase: The main goal of the profiling phase, which occurs every time an unmapped thread is run, is to measure the runtime latency in different retention times for a thread. This profiling phase incurs negligible overhead within the context of the total runtime of executing applications. We used the latency as our overhead indicator due to observations that different retention times had high impacts on latency overheads, whereas the energy was relatively invariant to different retention times. To measure the correct latency for a retention time option, the retention time associated with the thread being profiled should be fixed during the sampling period. However, the PWC controller may be unable to guarantee that every block in a thread accesses a specific retention time way without severely degrading the latency for other running threads. We address this challenge using the *emulation counter*, which enables the PWC controller to emulate different retention times within each physical retention time. The PWC controller simultaneously monitors both the emulation counter and the expiration counter during the profiling phase. When the expiration counter reaches its maximum count, but the emulation counter has not, the controller refreshes the block to keep the block alive until the emulated retention time has elapsed. The refresh mechanism features a refresh buffer to temporarily store the cache block before it is written back to the PWC. To minimize overhead, the refresh buffer is only used during the runtime profiling phase, and otherwise disabled.

The emulation counter, *emu_count*, is implemented similarly to *exp_count* (Section IV-A). When a write request is sent to a PWC block in the profiling phase, the PWC controller enables and resets *emu_count* (and *exp_count*). When *emu_count* reaches its maximum count(*N*), the PWC controller invalidates and preemptively expires the block regardless of the status of *exp_count*. The emulation counter enables a runtime emulation of any arbitrary retention time for a PWC block, and in effect, a sample of the miss latency for any desired retention time. To achieve this, *emu_count* uses a different clock than *exp_count*, allowing the PWC controller to maintain data validity while emulating the desired retention time.

2) Execution phase and LRU retention time-aware (LRU-RT) replacement policy: We propose a replacement policy—called *LRU-RT*—that accounts for the retention time needs of threads' PWC blocks (Figure 6). LRU-RT first ensures that the victim block in a replacement is the least recently used block. Thereafter, LRU-RT ensures that the placement of blocks within the PWC set are in the cache ways that best serve the blocks' retention time needs. When a replacement is required, NECTAR reads all PWC blocks of the indexed cache set into a small *replacement buffer* and replaces the LRU victim block in the buffer. Next, LRU-RT assigns the blocks to the best PWC ways by iteratively calculating the cost of assigning each block to different retention times to determine the lowest-cost assignment. LRU-RT first picks an arbitrary retention time from the retention time set R as the start point $rStart$. Thereafter, a sequence of retention times, *CacheSetLayout*, is created to iterate through different retention time combinations. For each iteration, the cost is stored in a *RetentionCost* hashmap until all the combinations are explored. The combination with the lowest latency cost is returned as the final *CacheSetLayout*, representing the retention time mapping with the minimum latency cost.

C. Implementation overhead

One of STTRAM's important benefits is its high density [11]. As a result, NECTAR's area remained $\sim$50% less than SRAM. NECTAR uses 4 bits each for *exp_count* and *emu_count* to monitor the cache block's retention status (Figure 5), resulting in a per-block overhead of 1.56% (8 bits per cache block). We used a 1KB 4-way 64B cache for the profiling phase's refresh buffer. To minimize overhead, we used a 50μs retention time STTRAM, which has a 1 cycle read/write access latency, and a leakage power of 17.314mW. For the replacement buffer, we used a 512B direct-mapped 64B cache that proved sufficient for the replacement process and only incurs a 3.916mw leakage power overhead.

Finally, the *CostTable* and PWC block attributes (e.g., the block's thread ID) are stored in the software meta-data field of the translation descriptor [1] without incurring additional overheads. For the profiling phase calculation, NECTAR requires four 32-bit registers and one 32-bit division circuit; and two 32-bit registers, one 32-bit comparator, and one 32-bit adder to calculate the minimum cost retention time. Overall, NECTAR imposes $< 1\%$ area overhead compared to modern processors like the ARM Cortex-A78 [10].

V. Experimental Setup

We performed our analysis and implemented the proposed PWC design using a modified[1] version of gem5 [12]. We performed full system simulations of both shared and private PWC configurations using the Linaro Linux operating system. We modeled a 4-core processor with configurations like the ARM Cortex A78 [10], with a 2GHz clock. We focused our studies on the data PWC, since it dominates the page walk requests (more than 10x the instruction page walker requests). We used 11 PARSEC 3.0 multithreaded benchmarks [7], cross-compiled for the ARMv8-A ISA. Each benchmark was run with four threads using the *simlarge* inputs for 1B instructions after restoring the checkpoints from the region of interest.

[1]The modified gem5 is at www.ece.arizona.edu/tosiron/downloads.php.

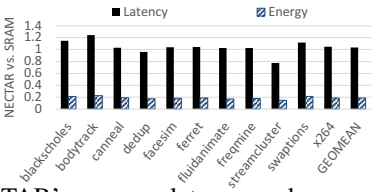


Fig. 7: NECTAR's average latency and energy normalized to SRAM

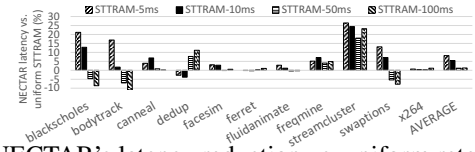


Fig. 8: NECTAR's latency reduction vs. uniform retention time STTRAMs

For comparison, we also modeled private PWCs with 8KB size (1024 TLB-equivalent entries) per core. We used the equivalent total cache size for the shared PWC—32KB size (4096 TLB-equivalent entries) for the four-core system. We considered four retention times: 5ms, 10ms, 50ms, 100ms, which we empirically found to be sufficient for the considered applications. We used the MTJ modeling techniques proposed in [13] to model the different retention times, and used NVSim [14] to estimate the STTRAM energy.

VI. Results

This section first quantifies the PWC access latency and energy of NECTAR compared to SRAM. Thereafter, we compare LRU-RT to LRU, and then compare the latency and energy of shared PWC to private PWC to contextualize the benefits of shared PWCs with current PWC designs [10].

A. Latency/energy comparison of NECTAR to SRAM

Figure 7 depicts NECTAR's latency and energy compared to SRAM. On average, NECTAR reduced the energy by 81.36%, and only incurred a 3.45% latency overhead. NECTAR achieved such high energy savings, in large part, because of STTRAM's low leakage power–approximately one-sixth that of SRAM. Furthermore, since the majority of the PWC accesses are read operations, the high dynamic write energy is not a prominent source of overhead for STTRAM-based PWCs. NECTAR traded latency for energy savings for *bodytrack*, whose threads required long (100m) retention times, but performed consistently well for the other benchmarks.

An important benefit of NECTAR, compared to a uniform retention time, is the ability to satisfy diverse workloads' retention time needs. Thus, while a uniform retention time (e.g., 50ms or 100ms) would also achieve high energy savings, NECTAR enables the savings to be maximized for *individual* applications and robust to new applications, while also minimizing the latency overheads compared to SRAM.

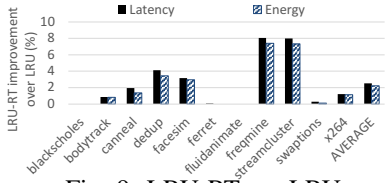


Fig. 9: LRU-RT vs. LRU

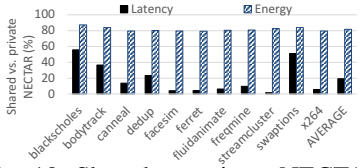


Fig. 10: Shared vs. private NECTAR

Figure 8 illustrates NECTAR's ability to mitigate the latency overhead of uniform retention STTRAMs for individual applications. NECTAR reduced the average latency compared to STTRAM-5ms, STTRAM-10ms, STTRAM-50ms, and STTRAM-100ms by 8.24%, 5.51%, 1.25%, and 1.34%, respectively. Even though these average improvements seemed modest, NECTAR improved the latency by up to 26% (for *streamcluster*). A closer look at individual threads revealed even higher latency improvements of up to 35.31% (additional figures are omitted due to limited space). In rare instances, NECTAR incurred overhead compared to a uniform retention time when all the threads within an application required long retention times (e.g., *blackscholes* and *bodytrack*).

B. LRU-RT vs. LRU in a shared PWC

Here, we quantify LRU-RT's improvement over LRU replacement policy used in prior work [3]. To isolate the impact of LRU-RT, we compare NECTAR+LRU-RT with NECTAR+LRU. Figure 9 shows the percentage of latency improvement of LRU-RT over LRU. On average across the applications, LRU-RT reduced the average latency overhead by 2.52% compared to LRU, and by up to 26.30% for some individual threads (additional figures are omitted for brevity). LRU-RT consistently outperformed LRU for all threads and applications and achieved similar energy benefits.

C. Shared vs. private NECTAR configurations

We also quantified the benefits of the shared NECTAR proposed herein to a private per-core NECTAR implementation. Figure 10 shows the percentage of latency and energy savings of shared NECTAR over private NECTAR configurations. On average, the shared NECTAR reduced the latency by 15.74%, with latency savings of up to 46.47% for *blackscholes*. The shared NECTAR substantially reduced the average energy by 81.60% compared to the private configuration, making the shared configuration a clear choice for energy- and latency-efficient STTRAM-based PWC implementation.

VII. Conclusion

In this paper, we performed the first study of STTRAM's viability for implementing energy-efficient page walker caches (PWC). Our analysis provides compelling new insights regarding the benefits of STTRAM for PWCs and challenges that must be addressed to achieve low-overhead implementations. Based on our analysis, we proposed *NECTAR*, a heterogeneous retention time STTRAM PWC for energy-efficient page walk. NECTAR includes a new LRU-based replacement policy, *LRU-RT*, that enables the placement of blocks to the PWC ways based on the blocks' retention time needs. Overall, for a variety of multithreaded applications, NECTAR reduces the average energy by 81.36% compared to SRAM, while imposing minimal latency and area overhead.

Acknowledgement

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

References

[1] "Armv8-A Address Translation." [Online]. Available: https://developer.arm.com/documentation/100940/0101

[2] H. Jang, K. Han, S. Lee, J. Lee, and W. Lee, "Mmnoc: Embedding memory management units into network-on-chip for lightweight embedded systems," *IEEE Access*, vol. 7, pp. 80 011–80 019, 2019.

[3] A. Bhattacharjee, "Large-reach memory management unit caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: Association for Computing Machinery, 2013, p. 383–394. [Online]. Available: https://doi.org/10.1145/2540708.2540741

[4] Z. Diao, Z. Li, S. Wang, Y. Ding, A. Panchula, E. Chen, L.-C. Wang, and Y. Huai, "Spin-transfer torque switching in magnetic tunnel junctions and spin-transfer torque random access memory," *Journal of Physics: Condensed Matter*, vol. 19, no. 16, p. 165209, 2007.

[5] 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.

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

[7] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[8] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong *et al.*, "Spin-transfer torque magnetic random access memory (STT-MRAM)," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 9, no. 2, p. 13, 2013.

[9] 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.

[10] "Arm® Cortex®-A78 Core Technical Reference Manual Revision r1p2." [Online]. Available: https://developer.arm.com/documentation/101430/latest

[11] P. Chi, S. Li, Y. Cheng, Y. Lu, S. H. Kang, and Y. Xie, "Architecture design with stt-ram: Opportunities and challenges," in *2016 21st Asia and South Pacific design automation conference (ASP-DAC)*. IEEE, 2016, pp. 109–114.

[12] 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.

[13] 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.

[14] 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.