SLAM: High Performance and Energy Efficient Hybrid Last Level Cache Architecture for Multicore Embedded Systems

Swapnil Bhosale, Sudeep Pasricha
Department of Electrical and Computer Engineering
Colorado State University, Fort Collins, CO, U.S.A.
swapnil.bhosale@colostate.edu, sudeep@colostate.edu

Abstract - Memory performance is a significant bottleneck in modern embedded multicore chip designs. In this paper, we propose a hybrid last level cache (LLC) architecture called SLAM that combines SRAM and emerging Spin-Transfer Torque Random Access Memory (STTRAM) to provide better power-performance trade-off compared to conventional SRAM-only, STTRAM-only, and previously proposed hybrid STTRAM-SRAM LLC architectures. We propose a framework (called SLAM) to reduce write operations to the STTRAM region of the hybrid LLC and consequently minimize the write energy of STTRAM. Our experimental results show that SLAM achieves 29.23% and 5.94% total LLC energy savings and 6.863% and 0.407% performance improvement compared to two state-of-the-art proposed techniques to reduce the write latency and write energy of STTRAM for various parallel applications.

I. INTRODUCTION

With the proliferation of parallel and data-intensive applications, memory performance is fast becoming the key bottleneck in multicore systems that execute these workloads [1]-[4]. Traditional SRAM based caches exploit locality to reduce costly main memory accesses, but lack the density needed to handle large datasets.

Recent years have seen the advent of non-volatile memories such as PCRAM (Phase-Change Random Access Memory) [5]-[6] and STTRAM (Spin-Transfer Torque Random Access Memory) [7] to replace conventional DRAMs and SRAMs. In particular, STTRAM has the potential to replace SRAM in the cache memory hierarchy. STTRAM possesses many advantages over SRAM, such as higher density, non-volatility, and lower leakage power consumption. The name STTRAM was first coined by Grandis Inc. Hynix Semiconductor and Grandis formed a partnership in 2008 to explore the commercial development of STTRAM technology. Hitachi and Tohoku University demonstrated a 32-Mbit STTRAM in June 2009. In 2011, Qualcomm showed a 1 Mbit Embedded STT-MRAM, manufactured in TSMC's 45nm LP process technology. Many companies are today working on exploring STTRAM memory technology, including Everspin Technologies, Crocus Technology, and Spin Transfer Technologies.

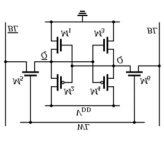
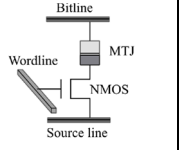
While STTRAM has several advantages, it also has a few key drawbacks. In the rest of this paper, we discuss the background and drawbacks of STTRAM, and describe our new *SLAM* architecture to overcome these drawbacks. Our work is an important step towards achieving STTRAM integration viability in future embedded multicore computing.

II. BACKGROUND

Here we discuss STTRAM cells and their advantages and disadvantages over traditional SRAM cells.

A. STTRAM cell: An STTRAM cell consists of a Magnetic Tunnel Junction (MTJ) combined with an access transistor [8]. The MTJ, which is the storage element, has an oxide (barrier) layer sandwiched between two ferromagnetic (FM) layers. The fixed layer is also called as 'hard layer' that has fixed magnetic orientation and the free layer has variable magnetic orientation. Data is stored in MTJ as relative orientation of the magnetization of two FM layers. When the layers are parallel (P) relative to each other, it represents a stored value of '0'; when the layers are anti-parallel (AP) relative to each other, it represents a stored value of '1'. Writes are performed using a directional spin polarized current. While writing a '1', a positive voltage is applied and the MTJ changes its state from P to AP; while a negative voltage is applied across the MTJ to change its state from AP to P and write a '0'.

Table 1: Comparison of SRAM and STTRAM cells

	SRAM	STTRAM
Cell structure		
Leakage power (for 1MB, 45nm)	14.63mW	2.32mW (5-6× less than SRAM)
Area (for 1MB, 45nm)	3.77mm^2	0.95mm^2 (4-5× denser than SRAM)
Write latency	3.18ns	12.01ns (approx. 4× of SRAM)
Write energy	0.08nJ	0.64nJ (approx. 8× of SRAM)

978-1-7281-2437-7/19/$31.00 ©2019 IEEE

B. SRAM vs STTRAM: The key characteristics of STTRAM that make it superior to SRAM are shown in Table 1, based on data from [9]. *Low leakage* power dissipation in STTRAM can be attributed to its MTJ cell structure as opposed to the CMOS-based structure in SRAM. The reverse biased parasitic diode formation in CMOS substrates [10] results in leakage current that contributes to leakage power. A typical SRAM cell has six transistors vs. just one in a typical STTRAM cell. Hence, leakage power is significantly lower in STTRAM compared to SRAM. *High density* of STTRAM can be attributed to its cell structure as well. The total area of an STTRAM cell can be as small as 6 F^2 vs. 120 F^2 for an SRAM cell [11], where F refers to the minimum feature size. *Scalability* of STTRAM is another important characteristic which enables it to be embedded with CMOS access transistors on silicon substrates across technology nodes [12].

C. Challenges with STTRAM: Two challenges that prevent STTRAM from replacing SRAM in conventional memory hierarchies are its *high write latency* and *high write energy* compared to SRAM (Table 1) [18]. There is a critical need for techniques that can overcome these two drawbacks to implement STTRAM in emerging cache memory hierarchies. Some researchers have proposed hybrid cache architectures (as shown in figure 1) that can intelligently utilize both STTRAM and SRAM technologies to provide superior performance and energy-efficiency than SRAM-only or STTRAM-only caches. Each set in a typical hybrid cache is partitioned into a small number of SRAM lines and a larger number of STTRAM lines to take advantage of both memory technologies. The basic idea is to migrate write intensive cache lines to the SRAM region, to minimize the number of write operations to STTRAM. In other words, the SRAM acts as a filter for write accesses in the hybrid cache. This allows the overall latency and energy consumption of the hybrid cache architecture to be reduced, while still benefitting from the higher density and lower leakage characteristics of STTRAM cells. Some state-of-the-art techniques that achieve this goal are discussed in the next section.

III. Related Work

The data brought into the last level cache (LLC) is less frequently modified than data at higher (e.g., L1) levels of cache [13]. LLCs are also designed to be large in size to accommodate frequently accessed data in higher level caches that may get evicted prematurely. The high density of STTRAM and comparable read time with SRAM makes it a good candidate for implementing LLC architectures. However, an STTRAM-only LLC architecture has latency and energy overheads that are not advantageous even when compared to conventional SRAM-based LLC architectures. Hybrid STTRAM-SRAM LLC architectures can however allow overcoming the disadvantages of both technologies to provide a superior solution.

Several prior works have proposed optimizations for such hybrid STTRAM-SRAM LLC architectures. In [10], a dynamic cache partitioning scheme is proposed for the hybrid LLC that considers different workloads in each core and changes the capacity of SRAM and STTRAM regions in each partition to take into account unbalanced write traffic in each partition. RWHCA [11] implements a counter based scheme that identifies a block with the highest write count as 'write intensive' and migrates it to the SRAM region after the count has reached a certain threshold, in a hybrid LLC architecture. PTHCM [14] maintains access history of every cache line, before it was evicted the last time, in a prediction table that helps predict write intensive cache line, and migrates such lines to the SRAM region. RWEEHC [15] leverages the MESI cache coherence protocol by adding extra cache block states to identify cache blocks that get significant amount of coherency-related writebacks from higher level caches, and migrates such blocks to SRAM regions once the block transitions to a certain coherence state. *Unfortunately, all of the above techniques incur notable hardware and performance overhead, overcoming which is the motivation for our SLAM framework, as discussed next.*

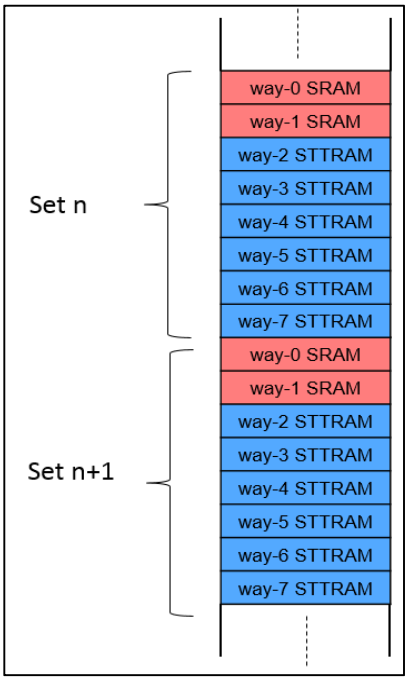


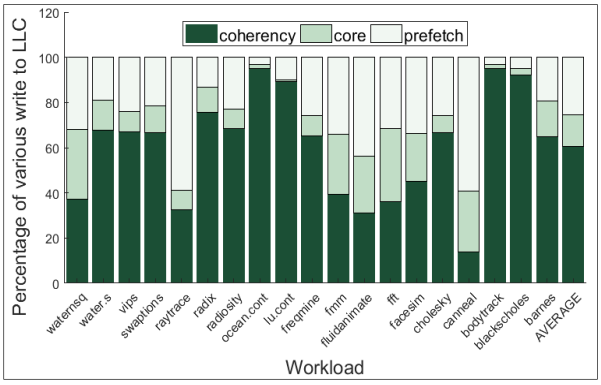
Figure 1: A hybrid SRAM-STTRAM cache architecture hybrid cache

IV. Motivation

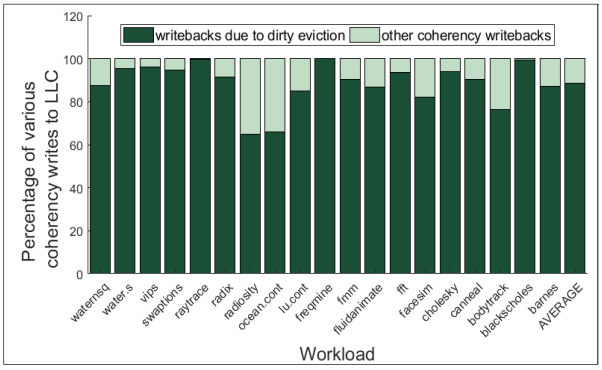
With the goal of minimizing write operations to the STTRAM region of a hybrid STTRAM-SRAM LLC, we analyzed the write operations to the LLC across multiple parallel and multi-threaded workloads [16]. We did

this analysis on an x86 based 4-core Intel processor with a 2.66GHz clock frequency. We assumed a two level cache hierarchy with the first level (L1) cache private to each core and second/last level cache (L2) shared among all the cores. We used a write-invalidate, directory based MESI cache coherence protocol for uniformity of shared data.

Our analysis found that write operations to LLC can be categorized into three types: *core, prefetch,* and *coherency*, as shown in Figure 2(a). *Core* writes refer to write operations done to the cache when data is brought into the cache on demand by a core. *Prefetch* writes indicate write operations done to the cache when data is brought into the cache in advance of it being accessed. *Coherency* writes indicate writes done to the cache upon updates to data in private caches, to maintain uniformity of data across the cache hierarchy. On average, *coherency* writes constitute 60% of the total writes done to the LLC as shown in figure 2(a).



(a)



(b)

Figure 2. (a) Distribution of various writes to LLC, (b) Distribution of coherency writebacks to LLC

Coherency writes can be further categorized into two types: *(i)* writebacks due to eviction of a modified block from any of the private caches (dirty eviction), and *(ii)* writebacks due to a core's request for a block that is modified in another core's private cache. Further analysis of coherency writes showed that writebacks due to dirty eviction constitute, on average, 88% of total coherency writes to the LLC as shown in figure 2(b). These writebacks can be avoided because they are not priority writebacks, i.e., the copy is not requested by another core. The procedure to achieve this is a key component of our *SLAM* framework, as discussed next.

V. SLAM Framework: Overview

This section first describes our architectural assumptions before explaining the working of the SLAM framework for hybrid LLC architectures in detail.

A. Architectural assumptions: To avoid writebacks to the LLC due to conflict misses at dirty blocks in higher level caches, our proposed *SLAM* framework monitors eviction of modified blocks from higher level caches. Figure 3 shows the architectural details of *SLAM*. We consider an *n*-core system as shown in the figure with an *N* level cache hierarchy. N-1 cache levels are private to each processing core and the last level cache (LLC) is shared among all the cores. Every set of LLC is partitioned into SRAM and STTRAM lines (e.g., as shown earlier in figure 1). We assume an inclusive cache system wherein the LLC has a copy of each entry in all higher-level caches.

If an entry is evicted from the LLC, all of its copies are invalidated from higher level caches to observe the inclusion property. All caches use the LRU replacement policy. To maintain coherency, we assume the use of the popular write-invalidate directory-based MESI cache coherence protocol with 'write-back' LLC i.e., data modified in higher level caches will be written back to LLC only after eviction from higher level caches. Also, whenever a dirty block is evicted from any of the private caches, it will be written back to the LLC for uniformity across entire cache hierarchy. Our *SLAM* framework tracks this type of writeback, and attempts to avoid it if it is directed to the STTRAM region of a hybrid LLC.

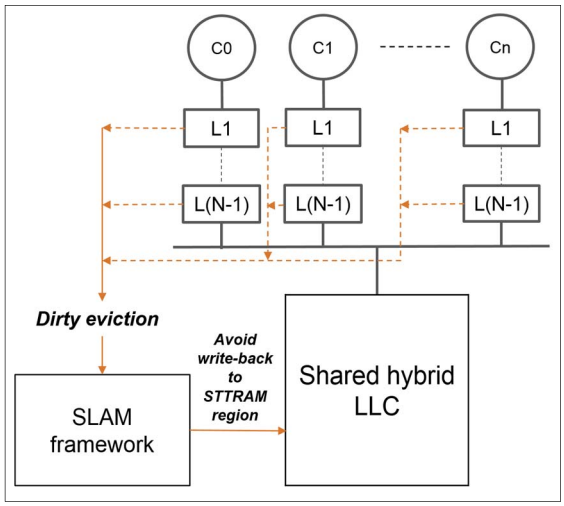


Figure 3. *SLAM* architecture

B. Working of SLAM: We consider a 4-core and 2-level cache hierarchy to explain the working of our *SLAM* framework, as shown in figure 4. We assume each L1 to be 32kB in size and 8-way set associative, and LLC to be 4MB hybrid with 16-way set associativity (4-way SRAM and 12-way STTRAM). To track the eviction of modified blocks from higher level caches, we modify the conventional LRU replacement policy. Every cache line in the entire cache hierarchy has a 3-bit 'LRU bits count' field that keeps track of block usage. If LRU bits count hits maximum (7) for a block, then that block is considered as the Least Recently Used (LRU) block and ready for eviction on a conflict miss. In figure 4, suppose line 4 in the MODIFIED state (as per the MESI protocol) is selected as the LRU block ('LRU bits count' = 7) for eviction from an L1 cache. Before this line is dropped into the write buffer for writing back to LLC, we store its address in an 'address buffer' and check if its copy exists in the STTRAM region of the shared hybrid LLC. If yes, then eviction of line 4 would cause a writeback operation to the STTRAM region.

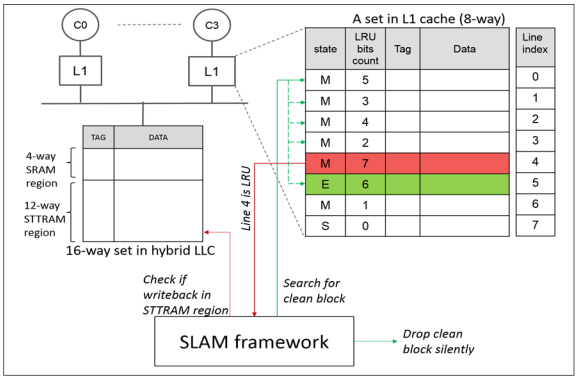


Figure 4. Example showing working of *SLAM*

To avoid this writeback, we search for a clean block, i.e. a block with an EXCLUSIVE (E) or SHARED (S) state, in the same set, and select it for eviction because eviction of a clean block from L1 would not cause a writeback to LLC. Our experimental results in Section VI show that eviction of such a block from L1 does not negatively affect performance in general. In this manner, we can reduce the number of writebacks from higher level caches to the STTRAM region in the hybrid LLC. Line 4 will be written back to LLC at the point when another processor will request (i.e., read/write) it. In that case, it will be a priority writeback, as discussed earlier.

We also experimented with strategies for the selection of the clean block for eviction. In our sensitivity analysis (not shown for brevity), we explored using a minimum threshold for the 'LRU bits count' value as a criterion for selecting a clean block for eviction. We considered values of 2, 4, and 6 for the threshold, and also utilized a simple random selection strategy. Our results indicated that random selection did not reduce overall system performance, and provided a more energy efficient (lower overhead) approach than a threshold-based strategy. As per *SLAM*'s algorithm, if the clean block is not found, then a writeback to STTRAM becomes inevitable. However, in our analysis across various workloads, a clean block was found on almost all of the MODIFIED block evictions, and its eviction did not affect overall system performance negatively.

C. Overhead: The hardware implementation of the *SLAM* framework requires a 32-bit buffer for each L1 private cache, to calculate and store the address of the line selected for eviction from L1 by the LRU replacement policy. This stored address is then used to check if the copy of this line resides in the SRAM or the STTRAM region in the hybrid LLC. This is done even before the selected line is actually evicted from L1 and put in the write buffer. The total hardware overhead for *SLAM* sums up to approximately 16B which is negligible compared to 4MB LLC.

The performance overhead in *SLAM* is mainly because of an extra access made to the LLC to check if the writeback due to dirty block eviction would be done to the STTRAM region of the hybrid LLC. It takes 8 clock cycles for 4MB L2 LLC to perform this access at the 45nm technology node, based on our analysis. It would take 1 cycle to load cache block states from L1 into a buffer for comparison and 1 cycle for performing the comparison. In the best case scenario, the clean block is found in the first iteration which requires 2 + 8 = 10 cycles. In the worst case scenario, the clean block is found in 2*7 + 8 = 22 cycles (for an 8-way cache), assuming that we iteratively consider the remaining 7 blocks in the set for possible eviction. This performance overhead is included in our simulation-based analysis, as discussed in the next subsection. Note that since each writeback to STTRAM requires 32 cycles, the performance overhead for our approach to minimize writebacks to STTRAM is justified in most cases, as will also become clear from the results discussed next. Also, any delays incurred while accessing the additional hardware, that may have affected critical path, were incorporated in modification of the cache memory subsystem using a delay parameter wherever required. The overall system performance and total execution time were calculated considering these delays.

VI. Simulation Results

In this section, we first describe our simulation setup before discussing the simulation results.

A. Simulation setup: We evaluate our proposed architecture on the x86-based multicore, parallel, and trace-driven Sniper simulator. Carlson et al. [17] validated the Sniper simulator against real hardware (4-socket 6-core Intel Xeon X7460 Dunnigton shared-memory with simultaneous multithreading (SMT) support) using the SPLASH-2 benchmark suite.

Table 2 shows the architectural parameters of our experimental system. We select an x86 based 4-core Intel processor with 2.66GHz clock that supports out-of-order execution. Each core has a private 8-way L1 cache with instruction and data in separate halves. The L2/LLC is 16-way inclusive and shared between all private caches with each set partitioned into 12-way STTRAM and 4-way SRAM. All the caches have a block size of 64B and use the LRU replacement policy.

Table 2: System configuration

CPU	x86, 2.66GHz, 4-cores, out of order execution
L1 cache	32kB SRAM split I/D caches 8-way, 64B block size 4-cycle latency (read and write) LRU replacement policy write-invalidate, write-back directory-based MESI
L2 cache/LLC	4MB hybrid inclusive (1MB SRAM + 3MB STTRAM); 64B block size 4-way SRAM and 12-way STTRAM 8-cycle *read* and *write* latency for SRAM 8-cycle *read* latency for STTRAM 32-cycle *write* latency for STTRAM LRU replacement policy; write-back cache
Simulator	SNIPER v6.1 (multi-core, parallel, trace-driven, high-speed and accurate x86 simulator)
Benchmarks	PARSEC-2.1 and SPLASH-2

The power and energy parameters for 2MB SRAM LLC, 8MB STTRAM LLC and 4MB hybrid LLC (1MB SRAM + 3MB STTRAM) are extracted from CACTI, NVSim, and SPICE simulations for the 45nm technology node and 16-way associativity as shown in Table 3. These parameters are used to evaluate the total LLC energy considering the overheads of our *SLAM* framework as discussed in Section V.

Table 3: SRAM/STTRAM power and energy parameters

	2MB SRAM LLC	**8MB STTRAM LLC**	**4MB Hybrid LLC (1MB SRAM + 3MB STTRAM)**
Read energy (nJ/access)	0.144	0.568	0.112/0.264
Write energy (nJ/access)	0.144	2.472	0.112/1.216
Static power (mW)	3825	1040	2302.5

We selected the PARSEC-2.1 and SPLASH-2 benchmark suites [16] for their wide collection of parallel and multithreaded workloads that represent various application domains viz. data mining (*freqmine*), financial analysis (*swaptions*), blocked matrix transpose kernel (*fft*), dense matrix factorization kernel (*lu.cont*), graphics (*raytrace*), etc. They are designed to exploit multicore platforms with varying memory intensities. We selected workloads with large usage and exchange of shared data to exploit cache coherency to the fullest. This is evident from the percentage of coherency writes to LLC in the selected workloads (94% in *ocean.cont*, 89% in *lu.cont*, 75% in *radix*, 68% in *freqmine*, etc). Although *SLAM* performed well on workloads with less percentage of coherency writes as well.

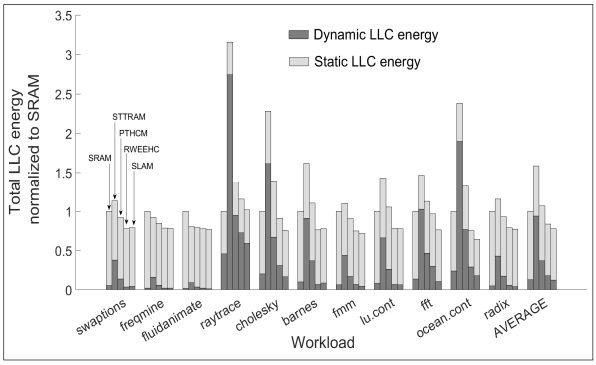
We evaluated *SLAM* against two prior works that use hybrid LLC viz. PTHCM [14] and RWEEHC [15] (discussed in Section III), and two baseline architectures viz. SRAM based LLC and STTRAM based LLC. For a fair comparison, we kept the LLC chip area for all 5 comparison architectures the same. The chip area for 2MB SRAM LLC at 45nm technology node was calculated as $44.7305mm^2$. On the same chip area, it is possible to fit 4MB of hybrid LLC (3MB STTRAM + 1MB SRAM) and 8MB of STTRAM LLC. The system configuration for all of the LLC architectures is the same as *SLAM*, except for the LLC size in SRAM-only and STTRAM-only LLC architectures.

The metrics used for our evaluation are: total LLC energy consumption and overall system performance measured in terms of IPC (instructions per cycle) of the executing workloads. To obtain these metrics, the selected workloads were run to completion in the detailed simulation mode on the Sniper simulator. Accesses to LLC were recorded for the entire application runtime, and used to evaluate the reduced write operations to the STTRAM region of the LLC and the reduction in total LLC energy consumption for our *SLAM* framework, as well as for the other architectures.

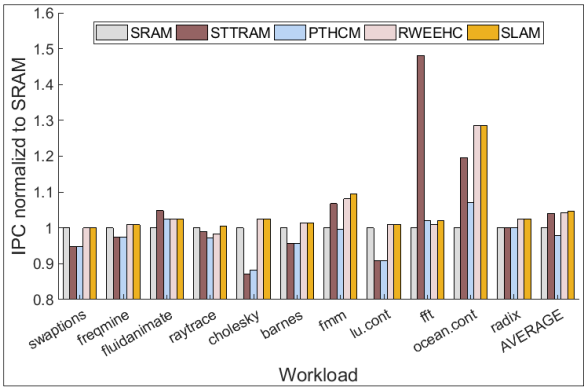
B. Simulation results: SLAM achieves 22.33%, 80.79%, 29.23%, and 5.94% reduction in total LLC energy consumption compared to SRAM, STTRAM, PTHCM [14], and RWEEHC [15] respectively as shown in figure 5(a). *SLAM* uses 16B of external hardware, which is negligible compared to the 262kB used in PTHCM, and hence *SLAM* saves a significant amount of dynamic energy consumption compared to PTHCM. The static energy savings because of effectively using STTRAM further reduces the energy costs with *SLAM*. SRAM-only LLC consumes more static energy while STTRAM-only LLC consumes more dynamic energy compared to the hybrid LLC in *SLAM*. As a result, *SLAM* saves more energy compared to both SRAM-only and STTRAM-only LLC architectures.

In RWEEHC, the hardware overhead is low, similar to *SLAM*, but performance is affected because of *dataless inserts* in the LLC on misses. As the data field in the LLC is not updated on an insert in RWEEHC, the number of writebacks on eviction from higher level caches is increased, irrespective of whether the evicted block is clean or dirty. Consequently, the write-back

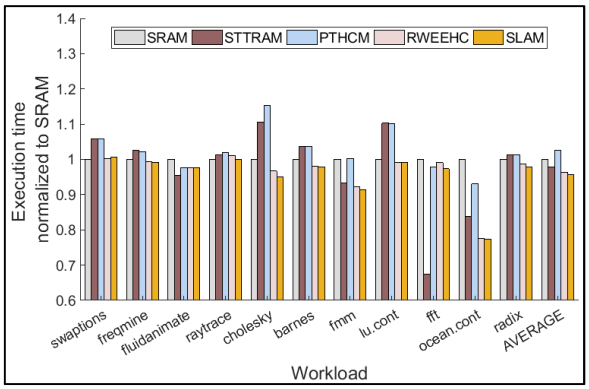
buffer gets full more often and needs to be emptied to make room for future writebacks. Hence, the system stalls more often with RWEEHC, which reduces performance and increases energy consumption. In contrast, *SLAM* avoids the writeback operations, thus avoiding saturating the writeback buffer, which improves system performance and reduces energy consumption.



(a)



(b)



(c)

Figure 5. (a) Total LLC energy consumption, (b) IPC, and (c) Total execution time, normalized to the SRAM-only cache architecture

Note that the goal of our *SLAM* framework was to achieve savings in total LLC energy while maintaining the overall system performance. *SLAM* is able to additionally achieve 4.631%, 0.607%, 6.863% and 0.407% improvement in application performance (IPC) compared to SRAM, STTRAM, PTHCM and RWEEHC respectively, as shown in figure 5(b). In terms of total execution time, *SLAM* achieved 4.24%, 2.02%, 6.91%, and 0.58% improvement compared to SRAM, STTRAM, PTHCM and RWEEHC respectively as shown in figure 5(c). *SLAM* eliminates most migration/swapping operations between SRAM and STTRAM regions that require write operations to the STTRAM, which explains its performance gains over PTHCM and RWEEHC. As *SLAM* avoids writeback operations, thus avoiding saturating the writeback buffer, it outperforms SRAM-only and STTRARM-only LLC architectures, as the dirty block evictions account for 88% of total coherency writes to LLC in both the baseline LLC architectures.

Note that *SLAM* does not perform well on *raytrace*. This can be explained as follows. As per *SLAM*'s algorithm, if the *clean block* is not found, then writeback to STTRAM becomes inevitable. In case of *raytrace*, the *clean block* is not found on a large number of dirty evictions. So the writeback operation to STTRAM cannot be avoided, which reduces performance for *SLAM*. Another observation is the spike in IPC for STTRAM on the *fft* workload. This is because the *fft* algorithm performs NxN blocked matrix transpose; with blocks being allocated in contiguous rows in the cache to exploit cache line reuse. As 8MB of STTRAM LLC has a lot more number of lines than 4MB of hybrid LLC and 2MB of SRAM LLC, it can accommodate larger data sets, and hence the memory access time is considerably less for *fft*, with the STTRAM-only LLC.

VII. Conclusion and Ongoing Work

Due to the increasing need of high capacity and power efficient on-chip caches, STTRAM has been considered as a potential replacement to traditional SRAM. However, its high write energy has been a hurdle in its path to replace SRAM completely from the cache memory hierarchy. Prior work to minimize the write energy of STTRAM has proposed some interesting hybrid STTRAM-SRAM LLC architectures, but at the cost of notable extra hardware that consumes additional dynamic energy. Our proposed framework, *SLAM*, attempts to minimize cache write energy with minimal hardware overhead and also ensure good overall system performance in hybrid STTRAM-SRAM LLC architectures. *SLAM* modifies the cache controller to track writeback operations to the LLC and mitigates them to avoid writes to the STTRAM region of the hybrid STTRAM-SRAM LLC. Compared to prior architectures, *SLAM* achieves up to 38.79% total LLC energy savings. Compared to baseline SRAM-only and STTRAM-only LLC architectures, *SLAM* achieves 18.94% and 32.31% total LLC energy savings respectively. As far as performance is concerned, *SLAM*

achieves up to 6.86% improvement in performance compared to prior architectures. Compared to the baseline SRAM-only and STTRAM-only architectures, *SLAM* achieves 4.631% and 0.607% improvement in overall system performance, respectively. These results highlight the potential of our *SLAM* framework to make hybrid STTRAM-SRAM LLC caches more viable for multicore computing.

Our ongoing work is exploring the integration of *SLAM* into systems with exclusive LLCs where the LLC is populated only through writebacks from higher level caches; and the impact of *SLAM* for lower technology nodes where the writes to STTRAM tend to be unstable because of small MTJ thickness.

ACKNOWLEDGEMENTS

This research is supported by grants from the NSF (CCF-1813370) and Rockwell-Anderson Foundation.

REFERENCES

[1] S. Pasricha, N. Dutt, “A Framework for Co-synthesis of Memory and Communication Architectures for MPSoC”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26, No. 3, pp. 408-420, Mar 2007

[2] S. Pasricha, N. Dutt, “COSMECA: Application Specific Co-Synthesis of Memory and Communication Architectures for MPSoC”, IEEE/ACM Design Automation and Test in Europe Conference (DATE), Mar 2006.

[3] I. Thakkar, S. Pasricha, “Massed Refresh: An Energy-Efficient Technique to Reduce Refresh Overhead in Hybrid Memory Cube Architectures,” IEEE International Conference on VLSI Design (VLSID), Jan 2016.

[4] L. Bathen, Y. Ahn, S. Pasricha, N. Dutt, “MultiMaKe: Chip-Multiprocessor Driven Memory-aware Kernel Pipelining”, ACM Transactions on Embedded Computing Systems (TECS), 12(1), Mar 2013.

[5] I. Thakkar, S. Pasricha, “DyPhase: A Dynamic Phase Change Memory Architecture with Symmetric Write Latency and Restorable Endurance”, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, Volume: 37, Issue: 9, Sept. 2018.

[6] I. Thakkar, S. Pasricha, “DyPhase: A Dynamic Phase Change Memory Architecture with Symmetric Write Latency,” IEEE International Conference on VLSI Design (VLSID), Jan 2017.

[7] A. Nigam, C. W. Smullen, V. Mohan, E. Chen, S. Gurumurthi and M. R. Stan, "Delivering on the promise of universal memory for spin-transfer torque RAM (STT-RAM)," IEEE/ACM International Symposium on Low Power Electronics and Design, 2011, pp. 121-126.

[8] A. Khvalkovskiy, D. Apalkov, S. Watts, R. Chepulskii, R. S. Beach, A. Ong, X. Tang, A. Driskill-Smith, W. H. Butler, P. B. Visscher, “Basic principles of STT-MRAM cell operation in memory arrays,” Journal of Physics D: Applied Physics, Volume: 46, Start Page: 074001, Year:2013.

[9] J. Ahn, S. Yoo, K. Choi, "DASCA: Dead Write Prediction Assisted STT-RAM Cache Architecture," IEEE International Symposium on High Performance Computer Architecture (HPCA), 2014, pp. 25-36.

[10] A. Jadidi, M. Arjomand, H. Sarbazi-Azad, "High-endurance and performance-efficient design of hybrid cache architectures through adaptive line replacement," IEEE/ACM International Symposium on Low Power Electronics and Design, Fukuoka, 2011, pp. 79-84.

[11] X. Wu, J. Li, L. Zhang, E. Speight, Y. Xie, "Power and performance of read-write aware Hybrid Caches with non-volatile memories," IEEE Design, Automation & Test in Europe Conference, 2009, pp. 737-742.

[12] S. Yazdanshenas, M. Pirbasti, M. Fazeli and A. Patooghy, "Coding Last Level STT-RAM Cache for High Endurance and Low Power," IEEE Computer Architecture Letters, 13(2), pp. 73-76, July-Dec. 3 2014.

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

[14] B. Quan, T. Zhang, T. Chen, J. Wu, "Prediction table based management policy for STT-RAM and SRAM hybrid cache," Int’l Conference on Computing and Convergence Technology (ICCCT), 2012, pp. 1092-1097.

[15] S. Agarwal, H. Kapoor, "Restricting writes for energy-efficient hybrid cache in multi-core architectures," IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), Tallinn, 2016, pp. 1-6.

[16] C. Bienia, S. Kumar, K. Li, "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on Chip-Multiprocessors," IEEE Int’l Symposium on Workload Characterization, 2008, pp. 47-56.

[17] A. Ayaz, L. Sawalha, "A Comparison of x86 Computer Architecture Simulators" Computer Architecture and Systems Research Laboratory, 2016.

[18] M. Krounbi, D. Apalkov, X. Tang, K. Moon, V. Nikitin, A. Ong, V. Nikitin, E. Chen, “Status and Challenges for Non-Volatile Spin-Transfer Torque RAM (STT-RAM),” International Symposium on Advanced Gate Stack Technology Albany, NY, Sep 29 –Oct 1, 2010.