Improving the Performance of STT-MRAM LLC Through Enhanced Cache Replacement Policy

Pierre-Yves Péneau(✉), David Novo, Florent Bruguier, Lionel Torres, Gilles Sassatelli, and Abdoulaye Gamatié

LIRMM, CNRS and University of Montpellier,
161 rue Ada, 34095 Montpellier, France
{peneau,novo,bruguier,torres,sassatelli,gamatie}@lirmm.fr

Abstract. Modern architectures adopt large on-chip cache memory hierarchies with more than two levels. While this improves performance, it has a certain cost in area and power consumption. In this paper, we consider an emerging non volatile memory technology, namely the Spin-Transfer Torque Magnetic RAM (STT-MRAM), with a powerful cache replacement policy in order to design an efficient STT-MRAM Last-Level Cache (LLC) in terms of performance. Well-known benefits of STT-MRAM are their near-zero static power and high density compared to volatile memories. Nonetheless, their high write latency may be detrimental to system performance. In order to mitigate this issue, we combine STT-MRAM with a recent cache The benefit of this combination is evaluated through experiments on SPEC CPU2006 benchmark suite, showing performance improvements of up to 10% compared to SRAM cache with LRU on a single core system.

1 Introduction

Energy consumption has become an important concern of computer architecture design for the last decades. While the demand for more computing resources is growing every year, much effort has been put on finding the best trade-off between performance and power consumption in order to build *energy-efficient* architectures. Current design trends show that the memory speed is not growing as fast as cores computing capacity, leading to the so-called *memory-wall* issue. Caching techniques, which have been pushed in the past for mitigating the memory-wall, are facing the silicon area constraints. Typically, up to 40% of the total area of processors [9] is occupied by the caches hierarchy. As a consequence, the energy consumed by this part of the CPU is important. As an example, it constitutes up to 30% of the total energy of a StrongARM chip [11]. In particular, as the technology scaling continues, the static power consumption is becoming predominant over the dynamic power consumption [4].

Data accesses that occur beyond the Last-Level Cache (LLC) are usually time and energy-consuming as they have to reach the off-chip main memory. An

M. Berekovic et al. (Eds.): ARCS 2018, LNCS 10793, pp. 168–180, 2018.
https://doi.org/10.1007/978-3-319-77610-1_13

intelligent design of the LLC reducing such accesses can save power and increase the overall performance. An usual technique adopted in the past consists in increasing the cache storage capacity so as to reduce the cache miss rate. This approach is no longer desired due to area and energy constraints. Increasing the cache size has a negative impact on the financial cost and increases the static power consumption.

In this paper, we consider an emerging memory technology, the Spin-Torque Transfer Magnetic RAM (STT-MRAM), a Non-Volatile Memory (NVM) that has a near-zero leakage consumption. This memory has a higher density than SRAM, providing more storage capacity for the same area. While STT-MRAM read latency is close to SRAM read latency, the gap for write access is currently one obstacle to a wide STT-MRAM adoption. In the present work, we study the impact in write reduction of cache replacement policies. Each read request leading to a cache miss eventually triggers a write. Upon this cache miss, the request is forwarded to an upper level in the memory hierarchy.[1] When the response is received, the corresponding data is written into the cache. Hence, the cache replacement policy has indirectly an important impact on the number of writes that occur upon cache misses. We carry out a fine-grained analysis on the actual sequence of read/write transactions taking place in the cache management strategy. On the basis of this study, we propose and evaluate the combined use of STT-MRAM and *state-of-the-art* Hawkeye cache replacement policy [8]. Thanks to Hawkeye, the number of writes due to cache misses is reduced, while benefiting from STT-MRAM density for larger LLC. Since STT-MRAM integration is known to provide energy savings [17], we put the focus on its impact on system performance so as to avoid a degradation of the overall energy-efficiency.

This paper is organized as follows: Sect. 2 presents related work; Sect. 3 introduces a motivational example and our proposed approach; Sect. 4 describes the experimental results validating our proposal; finally, Sect. 5 gives some concluding remarks and perspectives.

2 Related Work

The use of hybrid caches has been a recurrent approach to address write asymmetry in NVMs. A hybrid cache mixes SRAM and NVM memories to achieve the best of each technology. Most existing techniques rely on a combination of hardware and software techniques.

Wu et al. [18] proposed a hybrid memory hierarchy based on a larger LLC thanks to NVM density. They evaluated different memory technologies and identified eDRAM as the best choice for performance improvement, while STT-MRAM is the best choice for energy saving. Sun et al. [16] designed a hybrid L2 cache with STT-MRAM and SRAM, and employed migration based policy to mitigate the latency drawbacks of STT-MRAM. The idea is to keep as many write intensive data in the SRAM part as possible. Senni et al. [14] proposed a hybrid cache design where the cache tag uses SRAM while cache data array

[1] The first cache level (L1), the closest to the CPU, is the lowest level.

uses STT-MRAM. The cache reacts at the speed of SRAM for hits and misses, which slightly mitigate the overall latency, while power is saved on the data array thanks to low leakage. Migration techniques for hybrid memories are expensive and may suffer from inaccurate predictions, inducing extra write operations.

Zhou et al. [19] proposed another technique called early-write-termination: upon a write, if the value to write is already in the cell, i.e., a redundant write, the operation is canceled. This technique, implemented at circuit level, does not require an extra read before writing and saves dynamic writing energy. Nevertheless, it is mainly relevant to applications with many redundant writes.

Software techniques to mitigate NVMs drawbacks have been also proposed. Li et al. [10] proposed a compilation method called migration-aware code motion. The goal is to change the data access patterns in cache blocks so as to minimize the extra cost due to migrations. Instructions that access the same cache block with the same operation are scheduled by the CPU close to each other. Péneau et al. [13] proposed to integrate STT-MRAM-based cache at L1 and L2 level and to apply aggressive code optimizations to reduce the number of writes.

Smullen et al. [15] redesigned the STT-MRAM memory cells to reduce the high dynamic energy and write latency. They decreased the data retention time (i.e., the non-volatility period) and reduce the current required for writing. While this approach shows promising results, it relies on an aggressive retention reduction that incurs the introduction of a costly refresh policy to avoid data loss.

In this work, we take a complementary approach and evaluate the impact of cache replacement policies coupled with variations on LLC capacity in the reduction of critical writes. We basically re-evaluate the gap in performance between STT-MRAM and SRAM-based LLC given the latest advances in cache replacement policies.

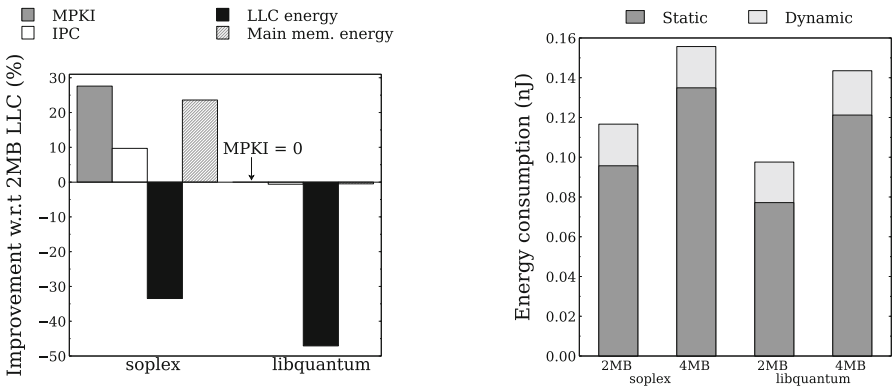
3 Motivation and Approach

In this work, we use the ChampSim [1] simulator with a subset of applications from the SPEC CPU2006 benchmark suite [7] for motivating our approach. Timing and energy results are obtained from CACTI [12] for the LLC and from datasheet information for the main memory [2]. More details of the experimental setup are given in Sect. 4.1. A common metric used to assess LLC performance is the *Miss Per Kilo Instructions* (MPKI), defined as the total number of cache misses divided by the total number of executed instructions. One possibility to reduce the MPKI is to increase the cache size. The cache contains more data and reduces the probability for a cache miss to occur. This results in penalties in terms of cache latency, energy and area.

3.1 Motivational Example

Let us evaluate the execution of two SPEC CPU2006 applications, namely *soplex* and *libquantum*. These applications have different memory access patterns. Figure 1a depicts the impact of 4 MB versus 2 MB LLC cache designs on

the MPKI, the Instruction Per Cycle (IPC) and the energy consumption of LLC and the main memory. For *soplex*, the MPKI is decreased by 27.6%, leading to a faster execution by 9.7%, while the energy consumption of the LLC and the main memory is respectively degraded by 33% and improved by 23%. While the performance for *soplex* application benefits from a larger cache, this induces a negative impact on the LLC energy consumption. On the other hand, the outcome is different for the *libquantum* application. As shown in Fig. 1a, the MPKI is unchanged (i.e., no improvement), while the IPC is slightly degraded by 0.6%. The energy consumption of the LLC and the main memory is also degraded, due to more expensive read/write transactions on the LLC. Moreover, a lower IPC, i.e., a longer execution time, increases the static energy. Here, the energy consumption of the LLC drastically grows by up to 47% with larger cache. The breakdown in static and dynamic energy consumption of the LLC is detailed in Fig. 1b: 80% of the energy comes from the static part.



(a) MPKI, IPC, and energy

(b) Energy breakdown

Fig. 1. Evaluation of 2 MB and 4 MB LLC for *soplex* and *libquantum*

Increasing the cache size shows interesting results for performance but faces two obstacles. Firstly, the LLC energy consumption is increased. Moreover, depending of the memory access pattern of the application, it may degrade the LLC energy while offering no gain in performance. Secondly, doubling the LLC size increases the silicon area on the chip. This aspect is crucial in design and larger caches are often not realistic due to area budget constraints. To tackle these two aspects, we consider STT-MRAM, which is considered as a future candidate for SRAM replacement [17]. NVMs offer near-zero leakage and are denser than SRAM (a STT-MRAM cell is composed of one transistor versus six transistors for a SRAM cell). But, they suffer from higher memory access latency and energy per access, especially for write operation. STT-MRAM offers a near-zero leakage consumption, eliminating the high static energy consumption observed with SRAM (see Fig. 1b). This is even relevant for applications

that do not benefit from larger cache such as *libquantum* (see Fig. 1b) In such a case, even though the execution time is longer, the energy consumption would not dramatically increase thanks to the low static energy of STT-MRAM.

3.2 Writes Operations at Last-Level Cache

At last-level cache, write operations are divided into two categories: *(a) write-back*, i.e., a write operation coming from a lower cache level, and *(b) write-fill*, i.e., a write operation that occurs when the LLC receives an answer from the main memory. These schemes are illustrated in Fig. 2. Let us consider L3 cache as LLC. Transaction (1) is a *write-back* coming from the L2 for data X. In this case, X is immediately written in the cache line (transaction (2a)). Possibly, a *write-back* could be generated by the LLC towards the main memory (transaction (2b)) if data D has been modified and needs to be saved. For the request (3) and (4), corresponding respectively to a *read* and a *prefetch*, the requested Y data is not in the cache. This cache miss triggers a transaction to the main memory to fetch Y, and upon receiving the response, Y data is written in the cache. This operation represents a *write-fill*. As with transaction (2b), a *write-back* is generated if data L replaced in the LLC must be saved.

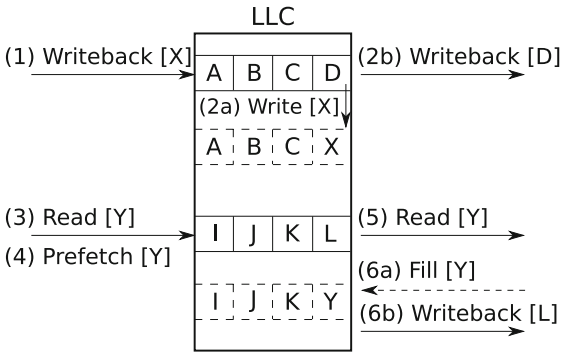
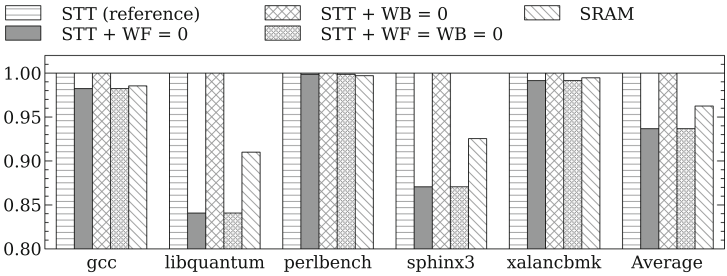


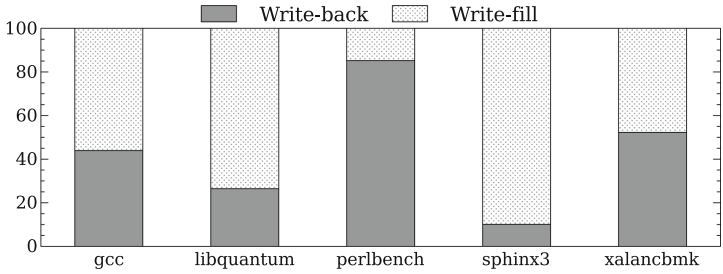
Fig. 2. Write transactions on the Last-Level Cache

Then, an important question that arises is to know whether or not *write-back* and *write-fill* have an equivalent impact on the overall system performance? For illustration, we consider five SPEC CPU2006 applications with different writes distributions to answer this question. Figure 3a reports the normalized IPC for different write latencies. Here, WF and WB respectively denote *write-fill* latency and *write-back* latency. We define the reference configuration as a 2 MB STT-MRAM LLC with $WF = WB = 38\,cycles$. Results are normalized to this reference. We also compare with a 2 MB SRAM LLC where $WF = WB = 20\,cycles$.

First, we set $WF = 0\,cycle$ in order to assess the impact of the *write-fill* operation on system performance. Then, we apply the same for WB for evaluating the impact of *write-back*. We also compare to the specific configuration where both WF and WB are set to zero. For all configurations, the write-buffer contains up to 16 elements. Moreover, bypassing is disabled for *write-back*.



(a) *Write-back* (WB) and *write-fill* (WF) effects on performances normalized to baseline STT-MRAM



(b) *Write-back* and *write-fill* distribution

Fig. 3. Write operations performance and distribution

When $WF = 0\,cycle$, i.e., *write-fill* has no impact on performance, results show a reduced execution time by 0.93× on average and up to 0.84× for *libquantum*. When $WB = 0\,cycle$, i.e., *write-back* has no impact on performance, the execution time is the same as for the reference STT-MRAM configuration. Finally, when both WF and WB are set to zero, the execution time is the same as the case where only *write-fill* latency is set to zero. Performance gains are particularly visible for applications that have a higher number of *write-fill* than *write-back* requests, such as *libquantum* or *sphinx3*. Nevertheless, even for an application with more *write-back* requests such as *perlbench* (see Fig. 3b), results show that $WB = 0\,cycle$ has no impact on performance. These results show that only *write-fill* have a high impact on performance. Indeed, a *write-back* operation coming from a lower level of the memory does not require an immediate response from the LLC. Hence, it does not stall the CPU. Conversely, a

write-fill occurs upon a cache miss, meaning that the CPU needs a data to continue the execution of an application. Unless the data becomes available, the CPU could be stalled if further instructions depend on this data.

The above analysis shows that one should primarily focus on *write-fill* operations for reducing the number of writes on the LLC and improving system performance. Let us define A the performance improvement with $WF = 0$, B the performance improvement with $WB = 0$ and C the performance improvement with $WF = WB = 0$. Figure 3a shows that $A + B = C$ for all applications. Hence, A does not have an impact on B and vice versa. Therefore, one could reduce the number of *write-fill* without a side effect on *write-back*.

3.3 Cache Replacement Policy

Write-fill operations are directly dependent on the MPKI of the LLC. A low MPKI leads to a low amount of requests to the main memory, and then a low amount of *write-fill* operations. Thus, one way to mitigate the STT-MRAM write latency is to reduce the MPKI to decrease the number of *write-fill* requests.

The cache replacement policy is responsible for data eviction when a cache line is full. For example, in Fig. 2, data X of the *write-back* transaction erases data D. It means that D has been chosen by the replacement policy to be evicted. Hence, the next access to D will generate a cache miss. Therefore, the replacement policy directly affects the number of misses, and so the MPKI. An efficient policy should evict data that will not be re-used in the future, or at least be re-used further than the other data in the same cache line. The most common used policy is the Least-Recently Used (LRU), which is cheap in terms of hardware resources. However, LRU is less efficient than advanced replacement policies such as Hawkeye [8], which targets the theoretical optimal in terms of cache eviction decision. Hawkeye identifies instructions that often generate cache misses. For each cache access, a data structure called a *predictor* keeps in memory the result of this access, i.e., hit or miss. The instruction that has generated the access is also saved. Hence, the memory of the predictor contains instructions that generate hits or misses. Predictions are made upon each access. Cache blocks, which are accessed by instructions generating cache misses have higher priority for eviction. The policy is based on the MIN algorithm [5]. To the best of our knowledge, this is the most advanced replacement policy [3].

4 Experimental Results

4.1 Environment Setup

We describe the timing and area models used in the sequel for the LLC and the main memory. Then, we introduce the used simulation infrastructure and we explain its calibration with considered timing information.

Memory Model. We first optimized SRAM and STT-MRAM cache memories respectively for low leakage with CACTI [12] and read energy-delay-product with NVSim [6]. For both LLC models, we used 32 nm technology with a temperature of 350 K. The considered STT-MRAM model is provided with NVSim and assumes optimizations for cell area, set/reset pulse duration and energy. The obtained parameter values are summarized in Table 1. The considered main memory model is based on a publicly available datasheet from Micron Technology [2]. We modeled a 4 GB DDR3 with 1 DIMM, 8 ranks, 8 banks per ranks, organized with 16×65536 columns with 64 B on each row. Thus, each bank contains 64 MB of data, each rank 512 MB, and the total is 4 GB. The extracted latency parameters are as follows: $tRP = tRCD = tCAS = 11\,cycles$, $tRAS = 28\,cycles$, $tRFC = 208\,cycles$ and $tCK = 1.25$ ns.

Table 1. SRAM and STT-MRAM timing and area results configurations

	SRAM			STT-MRAM		
	2 MB	4 MB	8 MB	2 MB	4 MB	8 MB
Read latency [ns]	1.34	1.47	1.66	1.90	2.06	2.53
Write latency [ns]	1.34	1.47	1.66	5.75	5.83	6.07
Area [mm^2]	5.32	10.88	20.49	1.19	2.19	4.00

Simulation Environment. Our evaluation is conducted with the ChampSim simulator [1] used for the Cache Replacement Championship at ISCA'17 conference [3]. The simulator executes application traces. The modeled architecture is based on an Intel Core i7 system. Cores are Out-of-Order with a 3-level on chip cache hierarchy plus a main memory. The setup is specified in Table 2. We use a set of 20 SPEC CPU2006 traces available with ChampSim. The cache warm-up period is 200 millions instructions. Reported statistics concern a period of 800 millions instructions. We calculate the average performance, i.e., IPC, by applying a geometric mean on the IPCs measured for all applications, as in [8].

Eight configurations are addressed in this study: 2 MB LLC cache with SRAM and STT-MRAM; 4 MB and 8 MB LLC caches only with STT-MRAM; and each of these four caches is combined with either LRU or Hawkeye. For the sake of simplicity, we associate the prefixes M (for Medium), B (for Big) and H (for Huge) together with technology names in order to denote respectively the 2 MB, 4 MB and 8 MB LLC configurations. The name of considered replacement policies, i.e., LRU and Hawkeye, are used as a suffix. For instance, *M_stt_hawk* denotes a 2 MB STT cache, using the Hawkeye policy.

Latency Calibration. The reference LLC latency in ChampSim is 20 cycles for a 2 MB 16-way associative cache, based on an Intel i7 processor. This corresponds to a latency of 5 ns at 4 GHz. Hence, we define the following latency relation: $L_T = L_C + L_W = 5$ ns, where L_T is the total latency for LLC to process a

Table 2. Experimental setup configuration

L1 (I/D)	32 KB, 8-way, LRU, private, 4 cycles		
L2	256 KB, 8-way, LRU, unified, 8 cycles		
L3	Varying size/policy, 16-way, shared		
L3 size	2 MB	4 MB	8 MB
L3 SRAM latency	20	21	22
L3 STT latency (R/W)	22/38	23/38	23/38
Hawkeye budget	28.2 KB	58.7 KB	114.7 KB
CPU	1core, Out-of-order, 4 GHz		
Main mem. size/latency	4 GB, hit: 55 cycles, miss: 165 cycles		

request from L2 cache, L_C is the LLC access latency and L_W is the wire latency between L2 cache and LLC. Thus, the effective latency is the sum of the wire latency and the cache latency. Thanks to CACTI, we extract $L_C = 1.34$ ns for the LLC reference configuration. Then, $L_W = L_T - L_C = 3.66$ ns. We set L_W to this value and use it as an offset to calculate each cache latency with the previous latency relation, where L_C is extracted from either CACTI or NVSim.

4.2 Results

This section presents our results as follows: firstly, we assess the impact of the LLC size in SRAM and STT-MRAM, by exploiting density to enlarge the cache capacity. Secondly, we report results when taking the Hawkeye cache replacement policy into account. Finally, we discuss this policy w.r.t. LRU. Except when it is explicitly mentioned, all results are normalized to the reference setup, i.e., *M_sram_lru*.

Impact of Cache Size and Technology. Here, all configurations use the LRU replacement policy. The top of Fig. 4 shows the MPKI improvement w.r.t. the reference configuration. We observe that the *M_stt_lru* configuration does not influence the MPKI since the cache size remains unchanged. Conversely, a reduction of MPKI is clearly visible with *B_stt_lru* and *H_stt_lru* configurations. Some applications are not sensitive to cache size, like *bwaves*, *libquantum* or *milc*. Conversely, the *lbm* application is very sensitive to cache size from 8 MB. For this application, the MPKI is decreased by a factor of 0.56× (i.e., 56%). This indicates that a large part of the working set now fits into the LLC.

The bottom part of Fig. 4 shows the normalized IPC achieved by STT-MRAM configurations w.r.t the reference configuration. The *M_stt_lru* configuration, i.e., a direct replacement of SRAM by STT-MRAM, is slower than the reference. This is due to the higher latency of STT-MRAM. The *B_stt_lru* and *H_stt_lru* configurations outperform the reference on average by 1.03× and 1.09× respectively. With *B_stt_lru*, the IPC is degraded for nine applications, while it

is only for five applications with *H_stt_lru*. The performance for the *soplex* application is correlated to the MPKI. Indeed, there is a linear trend between MPKI reduction and IPC improvement. Conversely, the following applications, *gobmk*, *gromacs* and *perlbench* exhibit a significant MPKI reduction with no visible impact on performance. This is due to the very low amount of requests received by the LLC compared to the other applications. Hence, reducing this activity is not significant enough to improve the overall performance.

On average, increasing the LLC size shows that STT-MRAM could achieve the same performance as SRAM under area constraint.

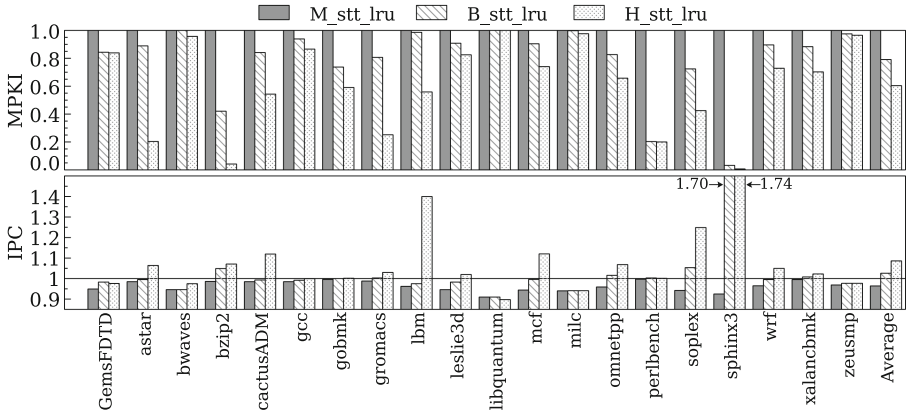


Fig. 4. MPKI (top) and IPC (bottom) with LRU normalized to *M_sram_lru*

Impact of Cache Replacement Policy. Here, all configurations use the Hawkeye replacement policy. Performance results are presented in Fig. 5. We observe the gains on the *M_sram_hawk* configuration, i.e., the Hawkeye reference. This configuration never degrades performances and achieves an average speedup of 1.05×. Larger STT-MRAM configurations, *B_stt_hawk* and *H_stt_hawk*, perform better than *M_sram_hawk* on average. Thanks to the Hawkeye policy, *M_stt_hawk* and *B_stt_hawk* outperform the reference for *lbm* or *mcf*. This was not the case with LRU, as depicted in Fig. 4. Note that for a few applications such as *bwaves*, *GemsFDTD* or *zeusmp*, the *M_sram_hawk* configuration achieves a higher speedup than larger configurations with the same MPKI. This shows that performance is still constrained by STT-MRAM latency, even with an enhanced replacement policy.

Nevertheless, Hawkeye improves performance where a larger cache only cannot. For example, all STT-MRAM configurations achieve the same IPC for the *milc* application with LRU, considering the LLC size. When Hawkeye is used, the performance is linearly increased with the cache size. As a matter of fact, Hawkeye can deal with some memory patterns not exploited by larger LLCs.

The best configuration is *H_stt_hawk*, which achieves a performance improvement of 1.1× (i.e., 10%) on average over the *M_sram_lru* baseline.

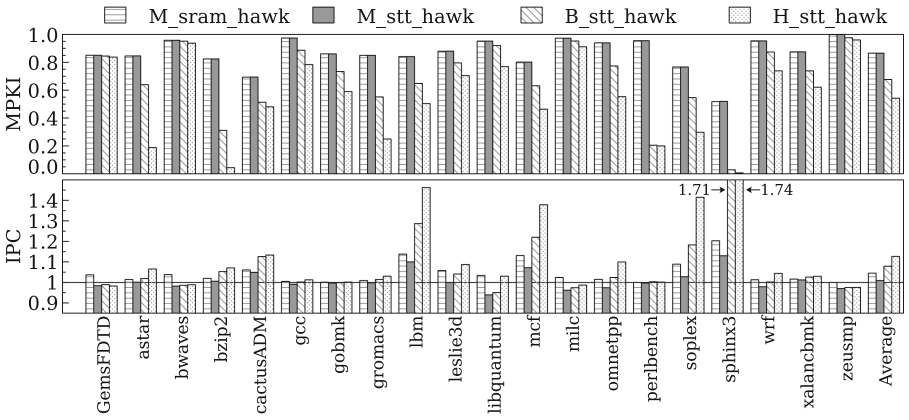


Fig. 5. MPKI (top) and IPC (bottom) with Hawkeye normalized to *M_sram_lru*

Hawkeye versus LRU. Figure 6 shows the effect of the Hawkeye policy over LRU. Results are normalized for each configuration to its counterpart with LRU. For example, *H_stt_hawk* is normalized to *H_stt_lru*. For this experiment, we also run SRAM configuration that do not fit into area constraint to illustrate the effect of Hawkeye on SRAM and STT-MRAM for the same cache size. Both SRAM and STT-MRAM configurations follow the same trend regarding the MPKI reduction over LRU since the Hawkeye policy is not impacted by cache latency. Moreover, we use a single core platform where parallel events cannot occur. Hence, eviction decision remains identical for a given cache size, regardless of the cache size. However, the average gain obtained with Hawkeye is slightly better with STT-MRAM. The performance gap between SRAM and STT-MRAM is 3.3% and 3.1%, respectively with LRU and Hawkeye. Hence, reducing the amount of *write-fill* has higher impact on STT-MRAM where writes are penalizing.

Figure 6 shows that the 8 MB configuration is not as efficient as the 4 MB configuration in terms of performance improvement. The average gain for the IPC for *H_sram_hawk* and *H_stt_hawk* is lower than *B_sram_hawk* and *B_stt_hawk*. This suggests an issue that can be due to either a larger LLC, or the Hawkeye policy, or both. Even if the overall performance improvement reported in Fig. 5 shows that the 8 MB configuration is faster, we note that there may be a limit to the performance improvement provided by the Hawkeye policy. This behavior is visible with *bzip2*, *wrf* and *sphinx3*. In Fig. 4, results show that the MPKI is reduced for *B_stt_lru* and *H_stt_lru*. Hence, increasing the cache size is efficient. Similarly, in Fig. 5, the MPKI is also reduced for the same configurations while replacing LRU by Hawkeye. However, the gains observed in Fig. 6 show that Hawkeye increases the MPKI compared to LRU for a 8 MB LLC. The reason is that Hawkeye made wrong eviction decisions. Indeed, the Hawkeye predictor exploits all cache accesses to identify the instructions that generate cache misses. Since a large cache size reduces the number of cache misses, it becomes more

difficult for the predictor to learn accurately from a small set of miss events. Note that the performance for *H_stt_hawk* is still better than other configurations despite these inaccurate decisions.

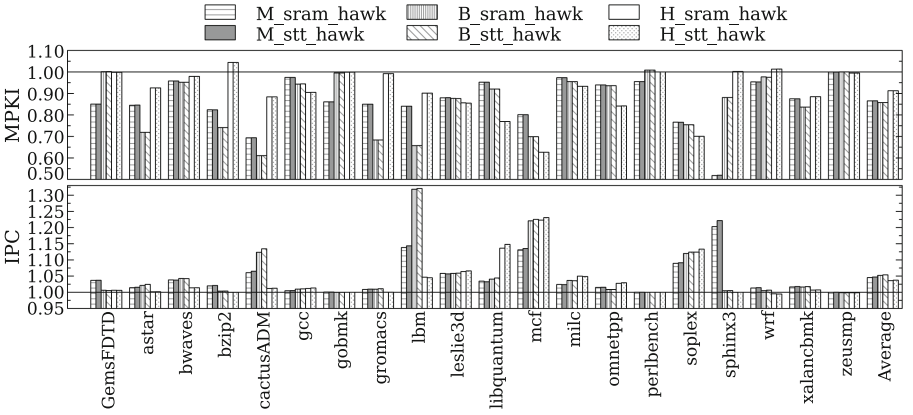


Fig. 6. Performance impact of Hawkeye normalized to LRU

5 Conclusion and Perspectives

This paper evaluates the mitigation of STT-MRAM performance obstacle by exploiting its density to increase LLC cache size and by applying an enhanced cache replacement policy to reduce the LLC *write-fill* activity due to cache misses.

We showed that *write-fill* are a side-effect of read misses and they are more important than *write-back* for performance improvement since they are on the *critical path* to main memory access. Thus, we applied the Hawkeye replacement policy which is designed for reducing cache read misses. Moreover, we showed that using such policy with STT-MRAM is more beneficial than with SRAM. Indeed, the read/write latency asymmetry of this technology allows a higher gap of improvement in terms of performance than with SRAM. However, with a large cache that drastically reduces the number of misses, the small amount of accesses makes the training of the Hawkeye predictor longer. Thus, it leads to wrong eviction decisions. The evaluation results showed that performance can be improved up to 10%. This gain, combined with the drastic static energy reduction enabled by STT-MRAM, leads to increased energy-efficiency.

Future work will focus on a deeper study of the Hawkeye policy to improve its accuracy under low LLC activity. A multicore design will be also investigated to confirm the benefits of large STT-MRAM LLC with this replacement policy.

Acknowledgements. This work has been funded by the French ANR agency under the grant ANR-15-CE25-0007-01, within the framework of the CONTINUUM project.

References

1. The ChampSim simulator. https://github.com/ChampSim/ChampSim
2. DDR3-Micron MT41K512M8DA-125 datasheet, October 2017. https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr3/4gb_ddr3l.pdf
3. ISCA 2017 Cache Replacement Championship. http://crc2.ece.tamu.edu
4. International Technology Roadmap for Semiconductors (ITRS) (2015)
5. Belady, L.A.: A study of replacement algorithms for a virtual-storage computer. IBM Syst. J. **5**(2), 78–101 (1966)
6. Dong, X., Xu, C., Xie, Y., Jouppi, N.P.: NVSim: a circuit-level performance, energy, and area model for emerging nonvolatile memory. IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. **31**(7), 994–1007 (2012)
7. Henning, J.L.: SPEC CPU2006 benchmark descriptions. ACM SIGARCH Comput. Archit. News **34**(4), 1–17 (2006)
8. Jain, A., Lin, C.: Back to the future: leveraging Belady's algorithm for improved cache replacement. In: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), pp. 78–89. IEEE (2016)
9. Kommaraju, A.V.: Designing energy-aware optimization techniques through program behavior analysis. Ph.D. thesis, Indian Institute of Science, Bangalore (2014)
10. Li, Q., Shi, L., Li, J., Xue, C.J., He, Y.: Code motion for migration minimization in STT-RAM based hybrid cache. In: 2012 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pp. 410–415. IEEE (2012)
11. Mittal, S.: A survey of architectural techniques for improving cache power efficiency. Sustain. Comput.: Inform. Syst. **4**(1), 33–43 (2014)
12. Muralimanohar, N., Balasubramonian, R., Jouppi, N.P.: CACTI 6.0: a tool to model large caches. HP Laboratories, pp. 22–31 (2009)
13. Péneau, P.Y., Bouziane, R., Gamatié, A., Rohou, E., Bruguier, F., Sassatelli, G., Torres, L., Senni, S.: Loop optimization in presence of STT-MRAM caches: a study of performance-energy tradeoffs. In: 2016 26th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), pp. 162–169. IEEE (2016)
14. Senni, S., Delobelle, T., Coi, O., Péneau, P.Y., Torres, L., Gamatié, A., Benoit, P., Sassatelli, G.: Embedded systems to high performance computing using STT-MRAM. In: 2017 Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 536–541. IEEE (2017)
15. Smullen, C.W., Mohan, V., Nigam, A., Gurumurthi, S., Stan, M.R.: Relaxing non-volatility for fast and energy-efficient STT-RAM caches. In: 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA), pp. 50–61. IEEE (2011)
16. Sun, G., Dong, X., Xie, Y., Li, J., Chen, Y.: A novel architecture of the 3D stacked MRAM L2 cache for CMPs. In: IEEE 15th International Symposium on High Performance Computer Architecture, HPCA 2009, pp. 239–249. IEEE (2009)
17. Vetter, J.S., Mittal, S.: Opportunities for nonvolatile memory systems in extreme-scale high-performance computing. Comput. Sci. Eng. **17**(2), 73–82 (2015)
18. Wu, X., Li, J., Zhang, L., Speight, E., Rajamony, R., Xie, Y.: Hybrid cache architecture with disparate memory technologies. In: ACM SIGARCH Computer Architecture News, vol. 37, pp. 34–45. ACM (2009)
19. Zhou, P., Zhao, B., Yang, J., Zhang, Y.: Energy reduction for STT-RAM using early write termination. In: IEEE/ACM International Conference on Computer-Aided Design-Digest of Technical Papers, ICCAD 2009, pp. 264–268. IEEE (2009)