Performance and Energy Impact of Enhanced Cache Replacement Policy on STT-MRAM LLC

Pierre-Yves Péneau, David Novo, Florent Bruguier, Lionel Torres, Gilles Sassatelli, Abdoulaye Gamatié

To cite this version:

HAL Id: lirmm-03341604
https://hal-lirmm.ccsd.cnrs.fr/lirmm-03341604
Preprint submitted on 11 Sep 2021
Performance and Energy Impact of Enhanced Cache Replacement Policy on STT-MRAM LLC

PIERRE-YVES PÉNEAU, DAVID NOVO, FLORENT BRUGUIER, LIONEL TORRES, GILLES SAS-SATELLI, and ABDOULAYE GAMATIÉ,
Laboratoire d’Informatique, de Robotique et de Microélectronique de Montpellier, France

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 and energy. 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 replacement policy. The benefit of this combination is evaluated through experiments on SPEC CPU2006 benchmark suite, showing performance improvements of up to 10% and 14% compared to SRAM cache with LRU respectively on single and multicore systems. Moreover, the energy consumption is on average decreased by 20% for all platforms.

Additional Key Words and Phrases: STT-MRAM, Last-Level Cache, Replacement Policy, Performance, Energy

ACM Reference Format:

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. As the memory hierarchy capacity is increased [45], the energy consumption of this part of the CPU scales accordingly. As an example, it constitutes up to 30% of the total energy of a StrongARM chip [28]. In particular, as the technology scaling continues, the static power consumption is becoming predominant over the dynamic power consumption [4].

Authors’ address: Pierre-Yves Péneau, pierre-yves.peneau@lirmm.fr; David Novo, david.novo@lirmm.fr; Florent Bruguier, florent.bruguier@lirmm.fr; Lionel Torres, florent.bruguier@lirmm.fr; Gilles Sassatelli, florent.bruguier@lirmm.fr; Abdoulaye Gamatié, abdoulaye.gamatie@lirmm.fr, Laboratoire d’Informatique, de Robotique et de Microélectronique de Montpellier, 161 rue Ada, Montpellier, France, 34095.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.
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 intelligent design of the LLC reducing such accesses can save power and increase the overall performance. A 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.

Here, 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.

The present work builds upon our previous studies [32, 34] and studies 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. 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. This investigation focuses on performance and power consumption. 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 [18]. Thanks to Hawkeye, the number of writes due to cache misses is reduced, while benefiting from STT-MRAM density for larger LLC.

This paper is organized as follows: Section 2 presents related work; Section 3 introduces a motivational example and our writes analysis; Section 4 proposes a design space exploration for the Last-Level Cache with the STT-MRAM technology; Section 5 describes the experimental setup to validate our proposal; Section 6 discusses our experimental results; finally, Section 7 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. [48] 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. [43] 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.

Kim et al. [20] promoted the design of exclusive last-level cache using STT-RAM. In such a cache hierarchy, evicted blocks from lower-level cache are copied in the last-level cache regardless. The authors showed that their solution drastically reduces energy consumption of the last-level cache while enhancing the system performance.

Priya et al. [37] introduced a mechanism for improving the lifetime of STT-RAM. They leveraged the MRU (Most Recently Used) replacement algorithm to achieve their goal. They claimed an improvement of 4x compared to the mainstream LRU replacement algorithm.

---

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

Working document
Senni et al. [39–41] proposed a hybrid cache design where the cache tag uses SRAM while cache data array 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.

Luo et al. [26], authors present a thrashing aware placement and migration policy (TAP) that address the thrashing blocks generation issue between L2 and LLC. They show how an adequate dirty thrashing blocks management in a hybrid memory architecture (combining SRAM and STT-MRAM) favors energy reduction while minimizing the performance loss.

Zhou et al. [50] 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 [7, 8, 36], 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. [22] 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. [33] proposed to integrate STT-MRAM-based cache at L1 and L2 level and to apply aggressive code optimizations to reduce the number of writes. Albeit software approaches are portable to different architectures, they break the abstraction layer between hardware and software. Indeed, compiler has to know the underlying technology.

Smullen et al. [42] 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. Later on, Bouziane et al. [9] leveraged the data retention time tradeoff to show how to efficiently map data on NVM.

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. Moreover, energy results are also provided. We assess these changes on the entire memory hierarchy, while some previous work [22–25, 30, 43, 47, 49, 50] often use a partial perspective, i.e., a cache-level only.

3 MOTIVATION AND APPROACH

In this work, we use the ChampSim [1] simulator with a subset of applications from the SPEC CPU2006 benchmark suite [17] for motivating our approach. Compared to usual simulation tools used for evaluating NVM in system architectures, e.g. gem5 [6, 11, 13, 14] and SimpleScalar [10, 44], ChampSim is a faster simulator, yet less precise tool compared to gem5. It executes application traces instead of full codes. Note that trace-driven simulation is also possible with gem5 [12, 31]. Another benefit is that it can be easily customized to evaluate various cache replacement techniques, while it would be more tedious with an environment such as gem5. Timing and energy results are obtained from NVSim [15] for the LLC and from datasheet information for the main memory [2]. More details of the experimental setup are given in Section 5.1. 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 4MB versus 2MB 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 damaged 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 Figure 1a, the MPKI is unchanged (i.e., no improvement), while the IPC is slightly decreased 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 Figure 1b: 80% of the energy comes from the static part.

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. Nowadays, 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 [46]. The STT-MRAM technology suffers from higher memory access latency and energy per access than SRAM, especially for write operation. However, STT-MRAM memory cells are composed of one transistor, while it is six transistors for SRAM cells. Hence, STT-MRAM is a denser. In addition, this technology offers a near-zero leakage power, which eliminates the high static energy consumption observed with SRAM (see Figure 1b). This aspect is particularly relevant for applications that do not benefit from larger cache such as libquantum (see Figure 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 Figure 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 requests (3) and (4), corresponding respectively to a read and a prefetch, the requested 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](image)

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, while Figure 3b depicts the write distribution for the considered applications. Here, $WF$ and $WB$ respectively denote write-fill latency and write-back latency. We define the reference configuration as a 2MB STT-MRAM LLC with $WF = WB = 38$ cycles. Results are normalized to this reference. We also compare with a 2MB 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.

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 Figure 3b), results show that $WB = 0$ cycle has no impact on performance. These results show that only write-fill have a high impact on
Write-back (WB) and write-fill (WF) effects on performances normalized to baseline STT-MRAM

Fig. 3. Write operations performance and distribution

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, with STT-MRAM, one should primarily focus on write-fill operations for reducing the number of writes on the LLC and improving system performance. Moreover, our results show that there is no side-effect between these two types of write. 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 in terms of performance.

3.3 Effects of write-fill reduction on the energy consumption

While a write-fill reduction has a positive effect in terms of performance, it could modify the type of requests on the LLC and by corollary the energy consumption. In this section, we examine what are these changes and their impact. Then, we analyze these effects on SRAM and STT-MRAM technologies.

3.3.1 Changes on memory accesses. Let us consider a memory hierarchy with inclusive L1 and L2 caches and an exclusive LLC. With such an architecture, the eviction of a block in the LLC does not evict this block in other caches.
The reduction of write-fill means that read requests from the L2 have to priority over the write requests. Whenever a block B is detected in the LLC as useless for reading, it is discarded and replaced. However, block B could be written in the future in the LLC if the L2 generates a write-back request for B. This scheme is described on the Figure 4. First, B is sent from the LLC to the L2 (step 1). Then, the LLC evicts B which is replaced by C (step 2 and 3). When the L2 generates a write-back request for B, the block is missing and it generates a write-back miss. C is evicted, and B is written. If B had not been evicted, the write-back would have generate a hit. Note that in both cases, the latency remains the same. This slight difference between write-back hit and write-back miss is not perceptible in terms of performance, but the energy requirement is different.

Equations 1 and 2 reflect the dynamic energy consumption of the LLC:

\[ E_{rd} = P^h_r \times E_r + \left\{ (1 - P^h_r) \times (E_r + E_m) \right\} \]
\[ E_{wr} = P^h_w \times E_w + \left\{ (1 - P^h_w) \times (E_w + E_m) \right\} , \]

where \( E_{rd} \) and \( E_{wr} \) respectively denote the total dynamic energy consumption for reading and writing, \( E_r \), \( E_w \) and \( E_m \) the energy consumption for a single read, write or miss, and \( P^h_r \) and \( P^h_w \) the probability of read hit and write-back hit. Decreasing the number of write-fill means increasing \( P^h_r \). Hence, reading becomes less costly since one avoid the cost of a miss, \( E_m \). As previously mentioned, a side-effect of this choice could be to reduce \( P^h_w \). In such a case, writing becomes more costly since a miss has been added. In a write-fill -optimized scenario where \( P^h_r \) is increased and \( P^h_w \) is decreased, the dynamic energy consumption of the LLC could be greater than a scenario without write-fill optimization.

3.3.2 Difference between SRAM and STT-MRAM. Changes on the dynamic energy consumption of the LLC do not have the same effect according to the technology used for the cache. Equation 3 depicts the energy consumption of the LLC, regardless of the technology:

\[ E = \alpha(W \times T) + \beta(E_{rd} + E_{wr}) , \]

where \( W \) is the static power of the LLC, \( T \) the execution time, and \( \alpha \) and \( \beta \) two variables. \( \alpha(W \times T) \) and \( \beta(E_{rd} + E_{wr}) \) respectively denot the static part and the dynamic part of the energy consumption. A cache that uses SRAM technology has a total energy consumption mostly dominated by the static part, i.e., \( \alpha \gg \beta \). An aggressive optimization on write-fill would decrease the total execution time \( T \), and by corollary the static energy consumption. However, even if the dynamic power is increased, the relation \( \alpha \gg \beta \) remains true. Then, the overall energy consumption of the LLC is decreased.

This behavior is not observable with the STT-MRAM technology. One key property of STT-MRAM is its very low static power. Then, the relation between \( \alpha \) and \( \beta \) is reversed and \( \alpha \ll \beta \). In such a case, an increase of the dynamic energy
consumption is not masked by the static part as with the SRAM technology. Then, a cache that uses the STT-MRAM technology could suffer from a greater energy consumption when using optimization on write-fill than without.

3.4 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 Figure 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, it is well-known that LRU is not the most efficient policy [38] and is also far from the theoretical optimal that could be achieved [16]. The state-of-the-art Hawkeye [18] replacement policy has been developed as an attempt to bridge this gap. This policy is based on the theoretical MIN algorithm, a.k.a. Belady’s algorithm [5]. To the best of our knowledge, this is the most advanced replacement policy [3].

This strategy identifies instructions that often generate cache misses. For a certain number of cache accesses, a data structure called a predictor keeps in memory the result of each access, i.e., hit or miss, by using saturating counters. The program counter of the instruction that has generated the access is also saved. Hence, the memory of the predictor contains instructions that generate hits or misses. Upon each cache access, the predictor is consulted, a prediction is made and saturating counters are updated. Cache blocks, which are accessed by instructions generating cache misses have higher priority for eviction.

4 NON-VOLATILE MEMORY EXPLORATION

4.1 Leveraging the density of STT-MRAM

Large LLC memory capacity allows to store more data and avoid costly accesses to the main memory.\(^2\) As a result, LLC accesses become more expensive in terms of latency and energy, while the leakage power increases as the area is doubled.

For the same cache capacity, STT-MRAM requires a smaller silicon area footprint than SRAM thanks to its higher density. In other words, STT-MRAM provides larger cache memory capacity for the same silicon area. In this work, we exploit this feature, to enlarge the LLC capacity up to the silicon area of the reference SRAM LLC. Hence, the following constraint must be satisfied:

$$A_{\text{sram}} \geq A_{\text{stt}},$$

where $A_{\text{sram}}$ is the silicon area of the reference LLC in SRAM and $A_{\text{stt}}$ is the silicon area of the LLC in STT-MRAM. The reference LLC selected in our study is a SRAM cache with a storage capacity of 2MB for monocore systems and 4MB for multicore systems. We use NVSim [15] to determine that the STT-MRAM LLC size can be increased up to 8MB and 16MB within the reference cache area constraint respectively for monocore and multicore systems (see more details in Section 5). Each cache size above this limit breaks the constraint expressed by Formula 4.

\(^2\)Note that some applications may not benefit from this feature. When memory accesses have no spatial locality, the cache architecture cannot capture this behavior and all accesses would eventually miss, regardless of the cache size.
4.2 Design exploration with NVSim

Usually, authors use NVSim to extract a cache configuration and give no explanation on how do they obtain these results. In this paper, we propose an extensive NVM exploration in order to select the best cache configuration according to a given criterion. To the best of our knowledge, this kind of design space exploration has never been presented in a paper.

4.2.1 Methodology. Our exploration is conducted with NVSim and adopt the following method:

- A minimal definition of the cache configuration
- An automatized exploration based on this definition
- The definition of a criterion to select a configuration
- A visualization of the exploration space
- The application of our criterion to select an appropriate configuration

We set five parameters in the minimal definition of a cache: memory size, word size, associativity, technology and temperature. The automatized process of exploration relies on the exploration mode provided by NVsim. For all parameters without a pre-defined value, NVSim explores a large variety of possibilities and store all results in a CSV file.

4.2.2 Criterion selection. There is different criteria of design selection based for example on the latency or the energy costs. Each of these criterion is usually used according to the considered cache level. For instance, a first-level cache should treat CPU requests in a fast manner. Hence, the latency criterion is pre-dominant over the energy. On the contrary, a Last-Level Cache should be optimized for its static energy, which is correlated to its area. Table 1 summarizes six criteria we identified for a cache configuration selection.

Since STT-MRAM is known for its high delay and energy, we want to reduce these values as much as possible. Moreover, we set an area constraint on the LLC for our exploration (see Formula 4). Then, we choose to take into account the area in our choice and select the Area-Delay-Energy Product (ADEP) as our main discriminant. This is a balanced choice between all three parameters that we need.

4.2.3 Result visualization. Figure 5 shows the design space exploration for a 4MB 16-way cache. There is a total of 72192 possible designs. For the sake of visibility, we perform a zoom on the most interesting region of the plot. Each point denotes an energy value. The intersection of this value, the area value and the delay value gives one possible design. Black points are the Pareto design according to these three metrics. The point surrounded with a square is the selected ADEP design. We repeat this process for all STT-MRAM cache configuration used in this paper. Results are summarized in Table 2.
5 EXPERIMENTAL SETUP

5.1 Environment Setup

We describe the timing, area and power 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.

5.1.1 Memory Model. For both LLC models, we used 22 nm technology with a temperature of 350 K. The considered STT-MRAM model is provided with NVSim and assumes optimization for cell area, set/reset pulse duration and energy. The obtained parameter values are summarized in Table 2 and are compliant with state-of-the-art projection for this technology scaling [29].

The considered main memory model is based on a publicly available datasheet from Micron Technology [2]. We modeled a 4GB DDR3 with 1 DIMM, 8 ranks, 8 banks per ranks, organized with $16 \times 65536$ columns with 64B on each row. Thus, each bank contains 64MB of data, each rank 512MB, and the total is 4GB. The extracted latency parameters are given in Table 3. For multicore systems with 8GB of memory, we add another DIMM with the same characteristics.

5.1.2 Power models. Caches memory. For each cache level, we extract the energy cost of each memory operation, i.e., read, write and miss, and multiply it by the number of reads, writes and misses observed on this cache level, as follows:

\[
E^i = \{R^i \times E^i_R + W^i \times E^i_W + M^i \times E^i_M\} + \{T \times P^i_{\text{leak}}\},
\]

where $i$ is the $i^{th}$ cache level; $R^i$, $W^i$ and $M^i$ are respectively the numbers of reads, writes and misses; $E^i_R$, $E^i_W$ and $E^i_M$ are respectively the costs of a read, a write and a miss operation; $T$ is the execution time and $P^i_{\text{leak}}$ is the leakage power. The first part between braces represents the dynamic energy consumption and the second part the static energy consumption.

Main memory. In addition to the latency model, Table 3 contains information extracted from the Micron datasheet [2] to create a power model for the main memory. Here, $RD$ and $WR$ are respectively the unit cost per read and write; $PRE$...
Table 2. SRAM and STT-MRAM timing and area results configurations.

<table>
<thead>
<tr>
<th></th>
<th>SRAM</th>
<th>STT-MRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2MB 4MB 8MB 16MB</td>
<td>2MB 4MB 8MB 16MB</td>
</tr>
<tr>
<td>Read latency [ns]</td>
<td>1.43 2.10 3.74 6.53</td>
<td>4.43 4.45 5.05 5.55</td>
</tr>
<tr>
<td>Write latency [ns]</td>
<td>-</td>
<td>6.00 6.05 6.31 6.52</td>
</tr>
<tr>
<td>Read energy [nJ]</td>
<td>0.07 0.10 0.14 0.19</td>
<td>0.20 0.25 0.26 0.29</td>
</tr>
<tr>
<td>Write energy [nJ]</td>
<td>0.06 0.09 0.13 0.18</td>
<td>0.25 0.30 0.30 0.33</td>
</tr>
<tr>
<td>Miss energy [nJ]</td>
<td>0.001 0.002 0.003 0.004</td>
<td>0.12 0.12 0.12 0.12</td>
</tr>
<tr>
<td>Static power [mW]</td>
<td>3.0 6.2 12.4 23.3</td>
<td>1.8 4.4 8.9 16.7</td>
</tr>
<tr>
<td>Area [mm²]</td>
<td>1.5 3.0 5.8 11.2</td>
<td>0.4 0.8 1.5 2.9</td>
</tr>
</tbody>
</table>

Table 3. Main memory configuration parameters [2]

<table>
<thead>
<tr>
<th></th>
<th>tRP tRCD tCAS tRAS tRFC tCK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11 cycles 28 cycles 208 cycles 1.25ns</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>RD WR PRE ACT ACTBG REF TREF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.47nJ 0.47nJ 0.22nJ 0.38nJ 0.027W 46.33nJ 64ms</td>
</tr>
</tbody>
</table>

and ACT are respectively the cost of page pre-charge and page activation; ACTBG is the active background energy consumption; REF is the cost of a refresh and TREF is the self-refresh frequency of the main memory.

We use the following formula [27] to compute the energy consumption of the main memory:

\[
E_m = E_a + E_b + E_c ,
\]

\[
E_a = R \times RD + W \times WR ,
\]

\[
E_b = (R_M + W_M) \times (PRE + ACT) ,
\]

\[
E_c = \left(\frac{T}{T_{REF}}\right) \times REF + T \times ACT_{BG} ,
\]

where \(E_m\) is the total energy consumption of the main memory, \(E_a\) the energy consumption due to read and write, \(E_b\) the energy consumption due to row buffer miss which triggers pre-activation and activation of memory pages, and \(E_c\) is the static energy consumption due to page refresh and static power.\(^3\) For Equation 7, \(R\) and \(W\) are the number of read and write. For Equation 8, \(R_M\) and \(W_M\) are the number of read miss and write miss. For Equation 9, \(T\) is the execution time.

5.1.3 Simulation Environment. Our evaluation is conducted with the ChampSim simulator [1] used for the Cache Replacement Championship at ISCA’17 conference [3]. 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. For multicore systems, we set the number of cores to 4. L1 and L2 caches are private to each core and the LLC is shared between all cores. The setup is specified in Table 4.

\(^3\)There is no low-power mode in our model that could reduce \(ACT_{BG}\).
Table 4. Experimental setup configuration.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>L1 (I/D)</th>
<th>L2</th>
<th>L3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32KB, 8-way, LRU, Private, 4 cycles</td>
<td>256KB, 8-way, LRU, Unified, 8 cycles</td>
<td>Varying size/policy, 16-way, Shared</td>
</tr>
<tr>
<td>L3 size</td>
<td>2MB</td>
<td>4MB</td>
<td>8MB</td>
</tr>
<tr>
<td>L3 SRAM latency</td>
<td>20</td>
<td>23</td>
<td>30</td>
</tr>
<tr>
<td>L3 STT latency (R/W)</td>
<td>33/39</td>
<td>33/39</td>
<td>35/40</td>
</tr>
<tr>
<td>Hawkeye budget</td>
<td>28.2KB</td>
<td>58.7KB</td>
<td>114.7KB</td>
</tr>
<tr>
<td>CPU</td>
<td>1 or 4 core(s), Out-of-Order, 4GHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Main mem. size/latency</td>
<td>4GB or 8GB, hit: 55 cycles, miss: 165 cycles</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each trace represents an isolated region of interest of 1 billion instructions. Each core executes a single-threaded application during 1 billion instructions. The cache warm-up takes 200 millions instructions while the remaining 800 millions instructions are used to report execution statistics. For multicore platforms, we consider 20 mixes composed each by four SPEC CPU2006 applications. Mixes have been generated randomly from the 20 applications and are presented in Table 5. When a core finishes its 1 billion instructions, it continues to read the trace to simulate an activity on the memory hierarchy until all cores reach 1 billion of executed instructions. The extra activity related to this mechanism is not reported in the final results. We calculate the average performance, i.e., IPC, by applying a geometric mean on the IPCs measured for all applications, as previous work did [18, 19, 35].

Sixteen configurations are addressed in this study: 2MB LLC cache with SRAM and STT-MRAM; 4MB and 8MB LLC caches only with STT-MRAM; and each of these four caches is combined with either LRU or Hawkeye. This is the same setup for multicore platform except that it is shifted from 4MB to 16MB. For the sake of simplicity, we associate the prefixes T (for Tiny), S (for Small), M (for Medium) and B (for Big) together with technology names in order to denote respectively the 2MB, 4MB, 8MB and 16MB LLC configurations. The name of considered replacement policies, i.e., LRU and Hawkeye, are used as a suffix. For instance, T_stt_hwk denotes a 2MB STT cache, using the Hawkeye policy.

5.1.4 Latency Calibration. The total access time of LLC in SRAM is usually dominated by the transfer delay of the interconnect, and not by the cache access itself [21]. This mitigates the impact of the potential performance penalty resulting from the integration of STT-MRAM in LLC. Therefore, we define the total access time \( L_T \) for the LLC as follows:

\[
L_T = L_C + L_I
\]

where \( L_I \) is the interconnect latency and \( L_C \) the cache latency (see also Figure 6). Thus, the effective latency is the sum of the interconnect latency and the cache latency. Our evaluation framework is calibrated based on an Intel-i7 processor where the LLC latency for a 2MB SRAM cache is 5\( ns \), i.e., \( L_T = 5\times 10^{-9} s \). With NVSim, we extract the cache access latency of a 2MB SRAM cache and we obtained \( L_C = 1.34 \times 10^{-9} s \). Hence, we calculate the interconnect latency:

\[
L_I = L_T - L_C = 3.66 \times 10^{-9} s
\]

We set \( L_I \) to this value and use it as an offset to calculate the total access time for each cache configuration mentioned in Table 2. Then, we convert this latency in cycles w.r.t. the CPU frequency.
Table 5. Detail of the 20 mixes executed on multicore platforms

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>mix1 gobmk</td>
<td>libquantum</td>
<td>perlbench</td>
<td>xalanchmk</td>
</tr>
<tr>
<td>mix2 astar</td>
<td>bwaves</td>
<td>lbm</td>
<td>zeusmp</td>
</tr>
<tr>
<td>mix3 cactusADM</td>
<td>lbm</td>
<td>milc</td>
<td>perlbench</td>
</tr>
<tr>
<td>mix4 bwaves</td>
<td>cactusADM</td>
<td>sphinx3</td>
<td>wrf</td>
</tr>
<tr>
<td>mix5 astar</td>
<td>cactusADM</td>
<td>GemsFDTD</td>
<td>perlbench</td>
</tr>
<tr>
<td>mix6 cactusADM</td>
<td>GemDFTD</td>
<td>gobmk</td>
<td>soplex</td>
</tr>
<tr>
<td>mix7 astar</td>
<td>cactusADM</td>
<td>lesli3d</td>
<td>sphinx3</td>
</tr>
<tr>
<td>mix8 bwaves</td>
<td>libquantum</td>
<td>perlbench</td>
<td>sphinx3</td>
</tr>
<tr>
<td>mix9 cactusADM</td>
<td>gobmk</td>
<td>milc</td>
<td>soplex</td>
</tr>
<tr>
<td>mix10 bzip2</td>
<td>gobmk</td>
<td>lbm</td>
<td>perlbench</td>
</tr>
<tr>
<td>mix11 astar</td>
<td>gobmk</td>
<td>milc</td>
<td>soplex</td>
</tr>
<tr>
<td>mix12 gobmk</td>
<td>lesli3d</td>
<td>libquantum</td>
<td>perlbench</td>
</tr>
<tr>
<td>mix13 bwaves</td>
<td>bzip2</td>
<td>gobmk</td>
<td>wrf</td>
</tr>
<tr>
<td>mix14 gobmk</td>
<td>lbm</td>
<td>lesli3d</td>
<td>milc</td>
</tr>
<tr>
<td>mix15 cactusADM</td>
<td>gobmk</td>
<td>milc</td>
<td>perlbench</td>
</tr>
<tr>
<td>mix16 bwaves</td>
<td>bzip2</td>
<td>gobmk</td>
<td>lesli3d</td>
</tr>
<tr>
<td>mix17 astar</td>
<td>bzip2</td>
<td>lesli3d</td>
<td>xalanchmk</td>
</tr>
<tr>
<td>mix18 gobmk</td>
<td>libquantum</td>
<td>wrf</td>
<td>xalanchmk</td>
</tr>
<tr>
<td>mix19 gobmk</td>
<td>lbm</td>
<td>milc</td>
<td>zeusmp</td>
</tr>
<tr>
<td>mix20 milc</td>
<td>perlbench</td>
<td>wrf</td>
<td>zeusmp</td>
</tr>
</tbody>
</table>

\[ L_T = L_I + L_C \]

**Fig. 6.** Configuration of the LLC latency in ChampSim

### 6 EXPERIMENTAL RESULTS

This section presents performance results in a first time, and energy results in a second time. Both mon core and multicore platforms are discussed.

#### 6.1 Performance results

Performance results are presented 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. For mon core platforms, the reference is a 2MB SRAM LLC with LRU, i.e., \( T_{\text{sram}_\text{lu}} \). For multicore platforms, it is a 4MB LLC with LRU, i.e., \( S_{\text{sram}_\text{lu}} \).
6.1.1 Impact of Cache Size and Technology. Here, all configurations use the LRU replacement policy. The top of Figure 7a shows the MPKI improvement w.r.t. the reference configuration. We observe that the $T_{stt\_lr}$ configuration does not influence the MPKI since the cache size remains unchanged. Conversely, a reduction of MPKI is clearly visible with $S_{stt\_lr}$ and $M_{stt\_lr}$ 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 8MB. For this application, the MPKI is decreased by a factor of $0.56 \times$. This indicates that a large part of the working set now fits into the LLC. Similar results are observed on Figure 8a for a multicore system. The MPKI is not changed with the default $S_{stt\_lr}$ configuration, and is reduced with other configurations. Configuration $B_{stt\_lr}$ reduces the MPKI up to $0.60 \times$ with mixes 7 and 8.

The bottom part of Figure 7a shows the normalized IPC achieved by STT-MRAM configurations w.r.t the reference configuration. The $T_{stt\_lr}$ 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 $S_{stt\_lr}$ configuration gives on average the same
results than the reference, while the $M_{stt\_lr}$ configuration outperforms the reference on average by $1.06\times$. With $S_{stt\_lr}$ the IPC is degraded for sixteen applications, while it is only for nine applications with $M_{stt\_lr}$.

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.

Results on multicore platforms (Figure 8a) follow the same trend as on a monochrome system. Configuration $S_{stt\_lr}$ exhibits lower performance than the reference configuration $S_{sram\_lr}$ due to the STT-MRAM latency. However, a bigger cache size is more efficient. The $M_{stt\_lr}$ configuration provides a lower IPC for thirteen applications and $B_{stt\_lr}$ for only four applications. As a result, the average gain on IPC with the largest configuration is $1.07\times$.

Working document
On average, increasing the LLC size shows that STT-MRAM could achieve the same or better performance as SRAM under area constraint for mono and multicore systems.

6.1.2 Impact of Cache Replacement Policy. Here, all configurations use the Hawkeye replacement policy. Performance results for a monocore system are presented in Figure 7b. We observe the gains on the T_sram_hwk configuration, i.e., the Hawkeye reference. This configuration never degrades performances and achieves an average speedup of 1.05×. Larger STT-MRAM configurations, S_stt_hwk and M_stt_hwk, perform better than T_sram_hwk on average.

Thanks to the Hawkeye policy, T_stt_hwk and S_stt_hwk outperform the reference for lbm or mcf. This was not the case with LRU, as depicted in Figure 7a. Note that for a few applications such as bwaves, GemsFDTD or zeusmp, the T_sram_hwk configuration achieves a higher speedup than larger configurations with the same (or almost) 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 M_stt_hwk, which achieves a performance improvement of 1.10× on average over the T_sram_lru baseline.

As with the LRU replacement policy, results for multicore systems on Figure 8b follow the same trend as on monocore platforms. However, they are more valuable. The default configuration S_stt_hwk badly affects the IPC for only 3 applications (up to 1.2% for mix 20) while it is degraded for 15 applications on a monocore platform. As a consequence, the average gain for the IPC with configurations M_stt_hwk and B_stt_hwk is respectively 1.09× and 1.14×. Configuration S_stt_hwk is the only STT-MRAM configuration that achieves lower results than S_sram_hwk, while its results are improved w.r.t. to S_sram_lru.

![Fig. 9. Performance impact of Hawkeye normalized to LRU for a monocore](image)

6.1.3 Hawkeye versus LRU. In this section we focus only on results for a monocore system to avoid redundancy. However, similar results are observed with a multicore platform.

Figure 9 shows the effect of the Hawkeye policy over LRU. Results are normalized for each configuration to its counterpart with LRU. For example, M_stt_hwk is normalized to M_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 9 shows that the 8MB configuration is not as efficient as the 4MB configuration in terms of performance improvement. The average gain for the IPC for $M_{sram\_hwk}$ and $M_{stt\_hwk}$ is lower than $S_{sram\_hwk}$ and $S_{stt\_hwk}$. 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 Figure 7b shows that the 8MB 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 Figure 7a, results show that the MPKI is reduced for $S_{stt\_lru}$ and $M_{stt\_lru}$. Hence, increasing the cache size is efficient. Similarly, in Figure 7b, the MPKI is also reduced for the same configurations while replacing LRU by Hawkeye. However, the gains observed in Figure 9 show that Hawkeye increases the MPKI compared to LRU for a 8MB LLC. The reason is that Hawkeye made inadequate eviction decisions. Indeed, the Hawkeye predictor exploits 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 $M_{stt\_hwk}$ is still better than other configurations despite these inaccurate decisions.

6.2 Energy results

In this section, we present the energy consumption and energy-efficiency results with LRU and Hawkeye replacement policies. We first describe our results for a monocore platform, and secondly for a multicore platform. For the rest of the paper, the energy-efficiency is characterized as the Energy-Delay Product (EDP).

6.2.1 Entire system perspective. Figure 10a depicts results for the energy consumption (top) and the EDP (bottom) for a monocore system with LRU. On average, configurations $T_{stt\_lru}$, $S_{stt\_lru}$ and $M_{stt\_lru}$ reduce the energy consumption respectively by a factor $0.89\times$, $0.86\times$ and $0.83\times$. The application libquantum is the only application where the energy consumption is increased. This is explained by the higher execution time due to longer STT-MRAM latency (Figure 7a). The Energy-Delay Product is correlated to the energy consumption. On average, the gain w.r.t. the SRAM reference is $0.95\times$, $0.86\times$ and $0.79\times$ respectively for configurations $T_{stt\_lru}$, $S_{stt\_lru}$ and $M_{stt\_lru}$. However, some applications like bwaves, libquantum or milc badly affect the EDP compared to the SRAM reference $T_{sram\_lru}$. These applications suffer from a too important slowdown in terms of execution time, due to their insensibility to the LLC size (Figure 7a).

Results with the Hawkeye replacement policies on Figure 10b show that the SRAM configuration $T_{sram\_hwk}$ provides a small gain of $0.95\times$ in terms of energy consumption. The low power feature of the STT-MRAM technology increase this gain up to $0.85\times$, $0.81\times$ and $0.80\times$ respectively for configurations $T_{stt\_hwk}$, $S_{stt\_hwk}$ and $M_{stt\_hwk}$. With the Hawkeye replacement policy, gains are more noticeable and no degradation can be observed as with the LRU. The EPD is also improved, and only bwaves and libquantum applications exhibit a higher EDP w.r.t. $T_{sram\_lru}$. The best average EDP is provided by configuration $M_{stt\_hwk}$.

Working document
Fig. 10. Energy consumption (top) and EDP (bottom) for a monochrome platform with LRU and Hawkeye replacement policies.

Thanks to the low-power feature of the STT-MRAM, results for multicore platform show a decrease of the energy consumption for all considered mixes. However, one can observe on Figure 11a that for mixes 1, 12 and 18, the energy consumption increases as the cache size is scaled. This is due to a higher execution time related to the STT-MRAM latency that increases with cache size. On average, we observe an energy consumption improvement of 0.91×, 0.90× and 0.85× respectively for configurations $T_{stt\_lru}$, $S_{stt\_lru}$ and $M_{stt\_lru}$. Regarding the EDP, we observe a counter-performance for mix 1, 12, 18 and 8. For the first three, this is due to their higher execution time previously mentioned. Performance of mix 8 on Figure 8b show an improvement, up to 1.20× with $B_{stt\_lru}$ configuration, while the energy consumption reduction with this cache configuration is slight and similar to configuration $T_{stt\_lru}$. This is due to the sphinx3 application in the mix, that achieves an IPC improvement of 1.50×, while others applications suffers from IPC reduction of 0.98×, 0.91× and 0.99×. The geometric mean shows an average improvement while the global execution time of mix 8 is increased by 1.09×. This explain the low improvement in terms of EDP. Results with the Hawkeye replacement policy are presented by Figure 11b. They follow a similar trend as with LRU, with more significant reductions on the
energy consumption. The improvement is gradual as the cache size increases. The EDP results are correlated to this observation, and the best average EDP improvement is achieved by configuration $B_{stt\_hwk}$ by $0.79\times$.

![Energy consumption and EDP for a multicore platform with LRU](image-a)

![Energy consumption and EDP for a multicore platform with Hawkeye](image-b)

**Fig. 11.** Energy consumption and EDP for a multicore platform with LRU and Hawkeye replacement policies

### 6.2.2 Last-Level Cache perspective

In the previous section, we present the energy consumption results for the entire system. Here, we rather focus on the Last-Level Cache and the impact of the Hawkeye replacement policy. For the sake of simplicity, this analysis is conducted on a monocyte platform, but results for multicore platform are similar.

Figure 12 illustrates the impact of Hawkeye on the energy consumption. For each cache application, the energy consumption with Hawkeye is normalized to its counterpart with LRU. On average, the effect is positive and the LLC consumes less energy with Hawkeye. However, results show the opposite for few applications, like *wrf* or *GemsFDTD*. With these applications, the energy consumption is always greater with Hawkeye than with LRU. This situation is due to changes on transactions on the LLC, as discussed in Section 3.3.
Fig. 12. Energy impact of the Hawkeye replacement policy on a monocore platform. For each configuration with Hawkeye, result is normalized to its counterpart with LRU.

Table 6. Evolution of the number of transactions received by the LLC with the Hawkeye replacement policy

<table>
<thead>
<tr>
<th>Metric</th>
<th>Read hit</th>
<th>Read miss</th>
<th>Write-back hit</th>
<th>Write-back miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>GemsFDTD</td>
<td>0.98×</td>
<td>1.0×</td>
<td>0.44×</td>
<td>2400×</td>
</tr>
<tr>
<td>libquantum</td>
<td>2.10×</td>
<td>0.92×</td>
<td>0.17×</td>
<td>7.10^5×</td>
</tr>
<tr>
<td>milc</td>
<td>16×</td>
<td>0.96×</td>
<td>0.26×</td>
<td>6.10^7×</td>
</tr>
<tr>
<td>wrf</td>
<td>1.08×</td>
<td>0.98×</td>
<td>0.83×</td>
<td>450×</td>
</tr>
</tbody>
</table>

Table 6 gives the total amount of read hit, read miss, write-back hit and write-back miss for four applications with a 4MB cache: GemsFDTD, libquantum, milc and wrf. On one hand, this illustrates the positive effect of Hawkeye. Except for GemsFDTD, the total number of read hit is increased, while the number of read miss is identical or decreased. Considering Formula 1, variable $P_{\text{hr}}$ is increased and the energy cost due to misses is decreased. On the other hand, we can observe the negative impact of the Hawkeye strategy. The number of write-back hit is largely reduced, from 0.83× and up to 0.17×, while the number of write-back miss is drastically increased, from 450× and up to 7.10^5×. Considering Formula 2, one can observe that when $P_{\text{wr}}$ is decreased, then additional miss costs are added and writing is more expensive.

Upon a certain threshold, the energy gains obtained by decreasing the number of read miss become less important than additional costs due to an increasing number of write-back miss. Hence, the global energy consumption of the LLC is increased.

7 CONCLUSION

This paper evaluates the impact of a state-of-the-art replacement policy used along with STT-MRAM technology on monocore and multicore architectures. The high density feature of the STT-MRAM is used to increase the cache size, while the replacement policy is used to improve the overall performance. Moreover, the low-power aspect of the STT-MRAM allows a designer to decrease the energy consumption.

We analyze the two different types of write that exists in the memory hierarchy and showed that 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. However, pre-simulation
analysis and experimental results suggest that this policy could have a negative impact on the energy consumption of the LLC for a certain class of applications.

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 inadequate eviction decisions. The evaluation results showed that performance can be improved up to 10% and 14% respectively for monocore and multicore platform. This gain, combined with the drastic static energy reduction enabled by STT-MRAM, leads to increased energy-efficiency up to 26.3%× and 27.7% for monocore and multicore systems.

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


Working document


