Low-Energy Volatile STT-RAM Cache Design Using Cache-Coherence-Enabled Adaptive Refresh

JIANHUA LI, Hefei University of Technology
LIANG SHI, Chongqing University
QINGAN LI, Wuhan University
CHUN JASON XUE, City University of Hong Kong
YIRAN CHEN, University of Pittsburgh
YINLONG XU, University of Science and Technology of China
WEI WANG, Hefei University of Technology

Spin-Torque Transfer RAM (STT-RAM) is a promising candidate for SRAM replacement because of its excellent features, such as fast read access, high density, low leakage power, and CMOS technology compatibility. However, wide adoption of STT-RAM as cache memories is impeded by its long write latency and high write power. Recent work proposed improving the write performance through relaxing the retention time of STT-RAM cells. The resultant *volatile* STT-RAM needs to be periodically refreshed to prevent data loss. When volatile STT-RAM is applied as the last-level cache (LLC) in chip multiprocessor (CMP) systems, frequent refresh operations could dissipate significant extra energy. In addition, refresh operations could severely conflict with normal read/write operations to degrade overall system performance. Therefore, minimizing the performance impact caused by refresh operations is crucial for the adoption of volatile STT-RAM.

In this article, we propose Cache-Coherence-Enabled Adaptive Refresh (CCear) to minimize the number of refresh operations for volatile STT-RAM, adopted as the LLC for CMP systems. Specifically, CCear interacts with cache coherence protocol and cache management policy to minimize the number of refresh operations on volatile STT-RAM caches. Full-system simulation results show that CCear performs close to an ideal refresh policy with low overhead. Compared with state-of-the-art refresh policies, CCear simultaneously improves the system performance and reduces the energy consumption. Moreover, the performance of CCear could be further enhanced using small filter caches to accommodate the not-refreshed private STT-RAM blocks.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; C.1.0 [**Processor Architectures**]: General

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Spin-torque transfer RAM, embedded DRAM, nonvolatile memory, refresh, energy efficiency, cache coherence

A preliminary version of part of this work appeared in *Proceedings of the Design, Automation & Test in Europe Conference (DATE'13)*.
This work was supported by grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CityU 123811 and 123210) and, in part, by the National Nature Science Foundation of China (No. 61073038). It was also partially supported by the Fundamental Research Funds for the Central Universities.
Authors' addresses: J. Li (corresponding author) and W. Wang, Hefei University of Technology, P.R. China; L. Shi, Chongqing University, P.R. China; Q. Li, Wuhan University, P.R. China; C. J. Xue, City University of Hong Kong, Kowloon, Hong Kong; Y. Chen, University of Pittsburgh, Pittsburgh, PA; Y. Xu, University of Science and Technology of China, P.R. China; corresponding author's email: jianhual@mail.ustc.edu.cn.

DOI: http://dx.doi.org/10.1145/2534393

ACM Reference Format:
Li, J., Shi, L., Li, Q., Xue, C. J., Chen, Y., Xu, Y., and Wang, W. 2013. Low-energy volatile STT-RAM cache design using cache-coherence-enabled adaptive refresh. ACM Trans. Des. Autom. Electron. Syst. 19, 1, Article 5 (December 2013), 23 pages.
DOI: http://dx.doi.org/10.1145/2534393

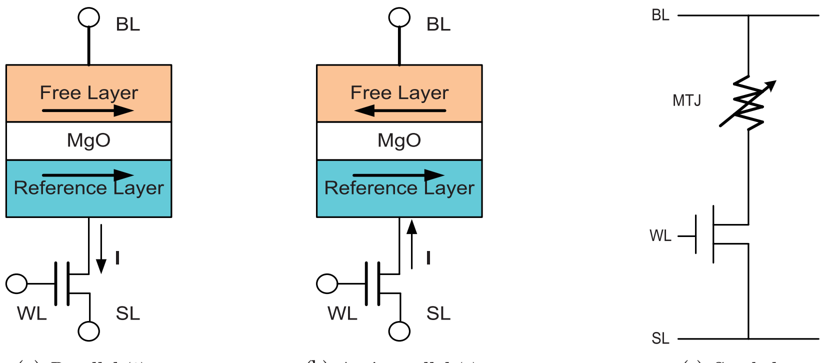
1. INTRODUCTION

As technology scaling continues, the leakage power of current on-chip SRAM cache memories gradually becomes the bottleneck of overall system performance [Kim et al. 2003; Meng et al. 2005]. Techniques, such as drowsy cache [Flautner et al. 2002; Meng et al. 2005], cache decay [Hu et al. 2002], and filter cache [Kin et al. 1997], can effectively reduce cache energy consumption from different aspects. However, these techniques are becoming less efficient under subthreshold technology node, because the transistor's leakage current exponentially increases as technology continues to scale down. Moreover, chip multiprocessor (CMP) has become the de facto design for modern high-performance computer systems, and the demand for larger on-chip cache memories makes the energy issue even more severe. Spin-Torque Transfer RAM (STT-RAM [Hosomi et al. 2005]) is an emerging nonvolatile memory technology. STT-RAM is promising to replace traditional SRAM as processor caches given its attractive features, such as fast read access, high storage density, low leakage energy, and unlimited endurance [Chen et al. 2010; Xue et al. 2011].

STT-RAM [Hosomi et al. 2005] is the second generation of Magnetic Random Access Memory (MRAM) [Tehrani et al. 2003], which settles the drawbacks of the first-generation, field-switched MRAM. Magnetic Tunnel Junction (MTJ) is the basic storage element in STT-RAM. Each MTJ consists of two ferromagnetic layers, known as reference layer and free layer, which are separated by one tunnel barrier (MgO). As shown in Figures 1(a) and (b), both reference layer and free layer possess their magnetization directions while the direction of the reference layer is pinned. Therefore, the relative magnetization directions can be parallel or anti-parallel through changing the magnetization direction of the free layer. In each cell, when two layers have parallel polarities, the resistance of MTJ is low, which indicates a logical value of 0. On the contrary, anti-parallel directions lead to high MTJ resistance that signifies a logical value of 1. The most universal structure of an STT-RAM memory cell is the 1T1MTJ cell structure which contains one MTJ as the storage element and one NMOS transistor as the access device shown in Figure 1(c).

Unfortunately, the widespread adoption of STT-RAM as the on-chip cache memory is impeded by two major obstacles, namely, its long write latency and high write energy. To tackle these issues, recent work [Smullen et al. 2011; Sun et al. 2011; Jog et al. 2012] proposed to improve the write performance of STT-RAM through relaxing the retention time of STT-RAM. The retention time of STT-RAM can be lowered through reducing the planar area of the MTJ, the saturation magnetization, the effective anisotropy, or the free layer thickness [Sun et al. 2011]. Through lowering the retention time of STT-RAM, the write current of STT-RAM is also reduced, resulting in improved write performance. For example, for 10 F^2 MTJ, the optimal read and write performance of STT-RAM is obtained when the retention time is relaxed to 56 μs, as illustrated by Smullen et al. [2011].

While volatile STT-RAM with relaxed retention time delivers comparable read and write performance with SRAM, the requirement of cache block refresh impedes the adoption such volatile STT-RAM as the large-capacity LLC for CMP systems. As an illustrative example, for the $56\mu s$ volatile STT-RAM [Smullen et al. 2011], refreshing one block needs seven cycles at 2GHz frequency. For a 512KB volatile STT-RAM with 64B line size, the total time to refresh all lines occupies more than half of the



(a) Parallel (0). (b) Anti-parallel (1). (c) Symbol.

Fig. 1. STT-RAM cell structure.

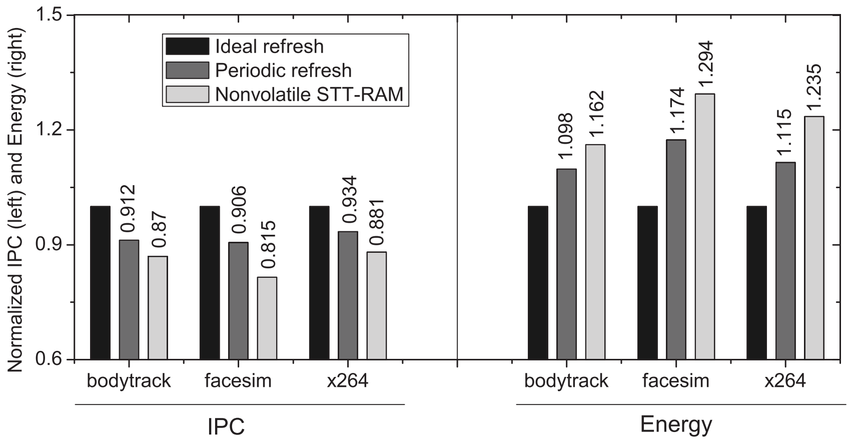


Fig. 2. Comparison of ideal refresh, nonvolatile STT-RAM, and periodic refresh.

retention period. Therefore, refresh operations could significantly conflict with the normal read/write accesses to the volatile STT-RAM. In the following, we present an example to show that the interference caused by the refresh operations could considerably degrade the overall system performance as well as bring about significant energy overhead.

Figure 2 shows the performance and energy comparison for a 16-core CMP system with an 8MB volatile STT-RAM-based LLC[1] using periodic refresh policy, [Smullen et al. 2011], an ideal refresh policy, and an 8MB nonvolatile STT-RAM-based LLC when running three LLC-intensive PARSEC programs [Bienia 2011]. The ideal refresh policy assumes that the refresh operation will never conflict with normal accesses and the refresh operations consume zero energy. As shown in Figure 2, due to the conflicts between refresh operations and normal accesses, IPC for *bodytrack*, *facesim*, and *x264* is reduced by 8.8%, 9.4%, and 6.6%, respectively, compared with the ideal refresh policy. More importantly, refresh operations also induce significant energy dissipation. For instance, the energy dissipation is increased by 17.4% for *facesim*. The goal of this article is to design an efficient refresh scheme to mitigate energy consumption through minimizing the refresh operations on volatile STT-RAM with minimum performance

[1]The detailed system setup is presented in Section 4.

degradation. As shown in Figure 2, the performance of the studied CMP system with nonvolatile STT-RAM LLC is the worst in the evaluated schemes. The results confirm the effectiveness of previous retention time relaxation schemes [Smullen et al. 2011; Sun et al. 2011; Jog et al. 2012].

In CMP systems, many blocks in the LLC are shared by multiple cores, and several copies of one block could be stored in multiple L1 caches. Cache coherence protocols are utilized to maintain coherent memory accesses on CMP systems by enforcing the Single-Writer, Multiple-Reader (SWMR) invariant [Sorin et al. 2011]. Moreover, many blocks in the LLC will never be reused and become dead after the eviction from upper-level caches. Based on these observations, in this article, we propose Cache-Coherence-Enabled Adaptive Refresh (CCear) to minimize the number of refresh operations in volatile STT-RAM caches. CCear cooperates with cache coherence protocols and cache management policies in CMP systems. Specifically, for shared blocks, CCear interacts with a modified cache coherence protocol to reduce the number of refresh operations. To mitigate the potential performance loss, a novel coherence state, W, is added into a baseline $MESI$ directory protocol to assist CCear. Filtered by L1 caches, many private blocks in LLC will never be reused or have a very long reuse distance after the eviction from upper-level caches [Khan et al. 2010b, 2010a; Liu et al. 2008; Jaleel et al. 2010]. CCear interacts with cache management policy to significantly cut down the number of refresh operations for private blocks.

The main contributions of this article are as follows.

—We present an analysis of the performance impact caused by refresh operations on volatile STT-RAM-based LLC and demonstrate the importance of minimizing the number of refresh operations.
—We propose CCear to minimize the number of refresh operations of shared and private cache blocks in volatile STT-RAM-based LLC through interacting with a modified cache coherence protocol and the cache management policy, respectively.
—Through full-system simulation, we show that volatile STT-RAM-based LLC with CCear yields comparable performance with the ideal refresh policy. Moreover, we propose to integrate small SRAM-based filter caches to further enhance the performance of CCear.

The remainder of this article is organized as follows. The related work is presented in Section 2. The proposed CCear policy is presented in Section 3, while the experiments and performance analysis are demonstrated in Section 4. Finally, Section 5 concludes.

2. RELATED WORK

In this section, we first present the literature of utilizing STT-RAM to design processor caches. Subsequently, related work about refresh operation on volatile memories is presented.

2.1. Design Caches Using STT-RAM

The stumbling block of adopting STT-RAM as processor caches is the inherent long write latency and high write energy. In the following, we primarily discuss the literature about optimizing the write performance of STT-RAM.

Reducing STT-RAM Write Activities. Dong et al. [2008] first evaluated the benefits of implementing caches with MRAM [Tehrani et al. 2003] in single-core architecture. It is found that the main stumbling block for the adoption of STT-RAM is the high energy and long latency of the write operations.

For the purpose of mitigating write energy of STT-RAM, Zhou et al. [2009] proposed the Early Write Termination (EWT) technique. The principle of EWT is to discriminatively update the bits whose new value is different from the old value. As a result, the energy of writing the unmodified bits can be eliminated with the EWT scheme. Rasquinha et al. [2010] proposed to reduce the number of write operations on STT-RAM caches through delaying the write-back operations from the high-level caches. Such a write-bias cache replacement policy modifies the locality information in cache sets, and therefore could degrade system performance. In addition, a write cache is added upon the STT-RAM with the purpose of coalescing write operations to STT-RAM caches.

Recent work [Wu et al. 2010; Sun et al. 2009] proposed integrating STT-RAM with SRAM to design hybrid cache memories. Wu et al. [2010] studied both inter-level and intra-level hybrid cache architectures which take advantage of the high density and nonvolatility of emerging memory technologies, including STT-RAM. Sun et al. [2009] proposed an SRAM-MRAM hybrid L2 cache for CMP systems, where a small SRAM is used to store write-intensive cache lines migrated from the STT-RAM. For the purpose of mitigating the performance impact of STT-RAM writes, write-intensive blocks are dynamically migrated from STT-RAM to SRAM in these hybrid caches to reduce the number of writes on STT-RAM. In order to support the migration operation, dedicated cache controller and access counters are needed in these schemes.

On the basis of the hybrid cache architecture [Sun et al. 2009], Li et al. [2011] proposed exploiting write nonuniformity among cache sets to improve the utilization of SRAM ways in hybrid caches. Jadidi et al. [2011] proposed migrating the write-intensive cache blocks to other cache lines in the same/different cache set to reduce the write frequency on STT-RAM ways. Recent work [Li et al. 2012] proposed a compilation technique to minimize the number of migrations in hybrid caches. Chen et al. [2012] proposed a reconfigurable hybrid cache by dynamically powering on/off cache ways on demand to optimize the power consumption.

Relaxing STT-RAM Retention Time. Differently from the preceding works, recent work [Smullen et al. 2011] proposed to relax the retention time of STT-RAM cells through shrinking the MTJ planar area. The resultant volatile STT-RAM has significant write performance improvement compared with the nonvolatile counterpart. For volatile STT-RAM LLC, the data is volatile and should be securely refreshed to prevent data losses. In addition, a periodic refresh policy was integrated in the relaxed STT-RAM [Smullen et al. 2011] to refresh the cells before data losses.

Sun et al. [2011] proposed to further relax the retention time of STT-RAM cells and evaluated the performance of the resultant STT-RAM as L1 caches. For ultra-low retention time STT-RAM caches, the cells have to be frequently refreshed wherein the refresh could conflict with normal read and write accesses and consume extra energy. To tackle this issue, a counter-based refresh scheme is proposed [Sun et al. 2011] to mitigate the overhead of refresh. Each cache block needs a counter to indicate the lifetime of the block, and the counter needs to be frequently updated by a global clock. Jog et al. [2012] found that the average interval between two successive writes to the same L2 cache block is around 10ms. Based on this observation, a 10ms retention time STT-RAM is designed and evaluated as the L2 cache. Differently from Smullen et al. [2011] and Sun et al. [2011], the expiring blocks are dynamically written to a small buffer.

Low retention time (μs level) STT-RAM exhibits better performance in terms of access latency, energy dissipation, and storage density. When utilizing such volatile STT-RAM as the large LLC for CMP systems, the refresh operation will be frequently scheduled, which in turn leads to severe conflicts with normal accesses. This article proposes techniques to minimize the number of refresh operations of volatile STT-RAM caches through exploiting the inherent features of LLC in CMP systems.

2.2. Refresh on Volatile Memory

Similar to DRAM, volatile STT-RAM [Smullen et al. 2011] needs to be periodically refreshed to prevent data loss. In the following, we present the literature of refresh schemes for volatile memories.

The refresh scheme is well known in DRAM, which is mainly used as main memory. Two basic means of performing refreshes are distributed refresh and burst refresh. For distributed refresh, the refresh operations of rows in DRAM (cache blocks for STT-RAM cache) are evenly distributed within the retention interval, typically 64 *ms* for DRAM. For burst refresh, rows are refreshed one by one sequentially. Note that normal accesses are not allowed during the refreshing period.

The read operations in DRAM are self-destructive, thus a DRAM row that was recently read or written to by the processor does not need to be refreshed again by the periodic refresh operation. Ghosh and Lee [2007] proposed exploiting this property to mitigate the refresh overhead in DRAM. By comparison, the read operation in STT-RAM is nondestructive, and the Smart Refresh [Ghosh and Lee 2007] cannot be directly applied to reduce the refresh operations in volatile STT-RAM. Recent work [Liu et al. 2012] proposed to use the information about variability in DRAM cell retention times to reduce the required refresh operations. This technique is orthogonal to the technique proposed in this article. For the purpose of mitigating the interference caused by refresh operations, Stuecheli et al. [2010] proposed predicting remaining idle periods for DRAM and scheduling refresh operations during these predicted periods. However, the total number of refresh operations is not changed. Thus the refresh energy is not substantially affected. More recently, Valero et al. [2013] proposed to exploit the reuse information of cache blocks to selectively refresh eDRAM cache blocks according to their MRU-Tour [Valero et al. 2012] information.

In Sun et al. [2011], a counter-based refresh scheme is proposed to reduce the refresh overhead in volatile STT-RAM-based L1 cache. The principle of the counter-based refresh is that a recently written cache block does not need to be refreshed again by the periodic refresh operation. For a small L1 cache, it is not difficult to ensure that one *write-instead-refresh* block will be refreshed again in the next periodic refresh operation. However, for large-capacity volatile STT-RAM-based LLC in CMP systems, the refresh period for each cache block will be significantly shortened under a given cell retention time. Therefore, it is not trivial to securely refresh a single LLC block which is refreshed by a write operation in the previous periodical refresh operation.

3. CACHE COHERENCE ENABLED ADAPTIVE REFRESH

In this section, we first illustrate the architecture of CMP systems which integrates volatile STT-RAM as the LLC. Subsequently, we present the refresh policy for shared and private blocks under CCear, respectively. Finally, filter cache is introduced to enhance the performance of CCear.

3.1. System Architecture

Figure 3 shows the architecture of a 16-core tiled CMP system. Under tiled CMP architecture [Taylor et al. 2002], each tile typically consists of a core, private L1 instruction and data caches, and a shared L2 slice. As shown in Figure 3, the 16 tiles are connected through on-chip network [Dally and Towles 2001]. In order to reduce leakage power and maintain high performance, the L2 cache data store is implemented with volatile STT-RAM [Smullen et al. 2011], while the L2 cache tag store and directory are implemented using low-power SRAM cells. Differently from the data store, the tag store and the directory are less heavily written, even for write-intensive workloads. Therefore, utilizing SRAM to store the tag and directory information could eliminate

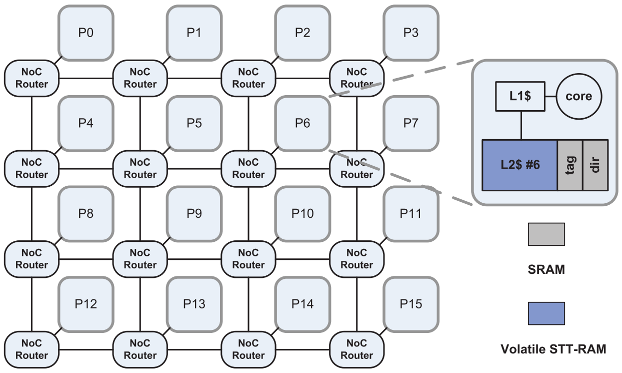


Fig. 3. Multicore architecture with volatile STT-RAM-based LLC.

the refresh operations which are energy consuming. We configure the L2 cache such that it is sequentially accessed (i.e., the tag and data are accessed sequentially). Under sequential access mode, the dynamic power of accessing the data array can be saved in case of cache misses.

3.2. CCear Policy for Shared Blocks

Under CCear, the access patterns of cache blocks are exploited to minimize the refresh operations in volatile STT-RAM caches. The LLC under CMP systems is typically shared among cores, thus each LLC block could have multiple replicas[2] in the upper-level caches. Cache coherence protocol [Sweazey and Smith 1986] is utilized to ensure the consistency between each LLC block and its replicas in the cache hierarchies. We propose two refresh policies, Not Refresh (NR) and Fixed Times Refresh (FTR), which exploit the inclusion property of cache hierarchies to reduce the number of refresh operations in volatile STT-RAM caches. In the following, the refresh policies are first illustrated. Subsequently, we present the method to maintain cache coherence.

3.2.1. Refresh Polices.

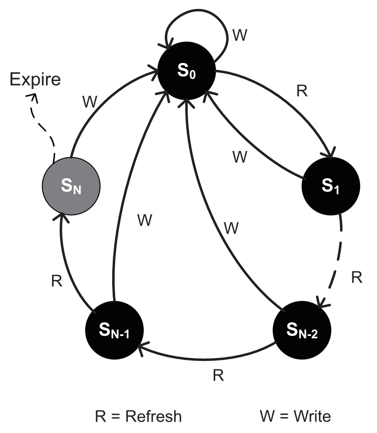
Not Refresh. This refresh policy is opposite to the always-refresh policy adopted in the periodic refresh. Under NR, when a refresh operation is activated on a shared block, CCear ignores the refresh command, and the data in the corresponding block will expire. To implement the NR policy, each cache block is configured with one extra bit, denoted as *valid_bit*. This bit is stored in the directory entry to indicate whether the data block indexed by this entry is valid or not. Upon receiving a refresh command to a shared block, the *valid_bit* of this block is first checked. If the block is valid (the value of this bit is 1), CCear will set the *valid_bit* to 0, which indicates that the block is not refreshed and the data in the block will be lost soon. Otherwise, CCear will do nothing because the data in the block has expired.

Fixed Times Refresh. Under this refresh policy, each shared block is refreshed for a fixed number of times. Specifically, each shared block will be refreshed N times after loading from main memory or writing back from L1 cache. The principle of FTR is to extend the valid period for shared blocks at the cost of more refresh operations. Compared with NR policy, the potential benefit of the FTR policy is that more coherence requests could directly obtain the target data from the LLC. Under FTR policy, each

[2]In this work, we consider the inclusive cache hierarchies which are simple in management and prevalent in both academia and industry.

```
if block(addr).valid_bit ==1
    if block(addr).ref_counter > 1
        refresh(block(addr));
        block(addr).ref_counter--;
    else
        block(addr).valid_bit = 0;
        return;
else
    return;
```

(a) Pseudocode.



(b) State transition diagram.

Fig. 4. CCear policy for shared blocks.

cache block is associated with a counter denoted as *ref_counter*. Initially, the *ref_counter* for each shared LLC block is set to N. N is a user-defined parameter. A larger N indicates that more data requests will directly obtain the target block at LLC. However, a larger N also indicates more energy-consuming refresh operations which also degrade system performance.

Figure 4(a) presents the pseudocode for FTR policy. When a periodic refresh command is activated for one block whose address is *addr*, FTR will first check the validity of the block, as shown in Figure 4(a). If the block is not valid, FTR will ignore the refresh command. Otherwise, FTR then checks the *ref_counter* of the block. If the block has not been refreshed N times, the block will be refreshed, and its *ref_counter* will be reduced by one. Otherwise, FTR chooses not to refresh the block and resets the valid bit of this block. NR policy can be identified as a special case for FTR policy where N is set to zero.

Figure 4(b) shows the state transition diagram for shared blocks under the FTR policy. There are a total of N states, where S_M indicates that the corresponding block has been refreshed M times. For each shared block loaded from off-chip memory or written back from upper-level caches, its state is S_0. If a shared block is continuously refreshed M times, its state will be switched from S_0 to S_M, as shown in Figure 4(b). For the block which has been refreshed N times, no further refresh will be carried out. Thus, the block will become expired, and the data in the block will be lost. As shown in Figure 4(b), the block state is switched to S_0 if the block is accessed by a write operation. Because STT-RAM is not self-destructive, a block which is recently accessed by a write operation does not need to be refreshed again by the periodic refresh operation. We suppose that the block will be written again in the near future and set the block to the original state (S_0) to extend the validity period of the block.

3.2.2. Maintain Cache Coherence. Through refreshing the shared blocks by zero or a fixed number of times, CCear can effectively mitigate the energy dissipation of refresh operations. However, the data in the not-refreshed block will expire, which makes conventional cache coherence protocol unable to ensure the consistency in cache hierarchies. For example, when another core issues a GetS request to one expired block, the requestor cannot get the correct data because the data stored in the block has expired. In order to maintain consistent access to the expired block, one approach is to write back the block before the data in the block expires. However, this approach could lead

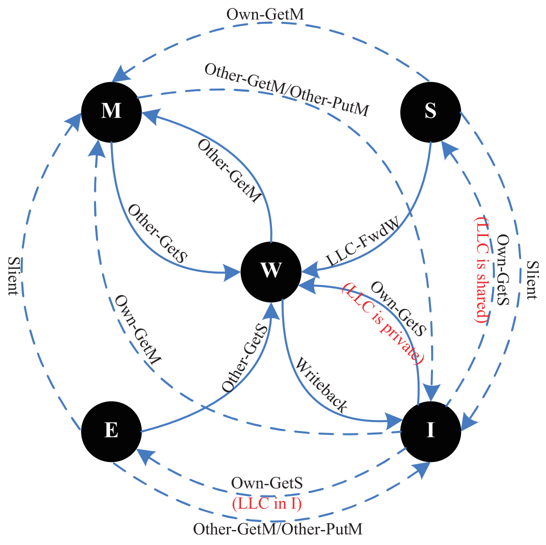


Fig. 5. Coherence state transition for L1 cache.

to extra on-chip traffic, because all the sharers of the block in the upper-level caches need to be invalidated. Moreover, writing back blocks could degrade system performance, because subsequent accesses to these blocks have to re-fetch them from off-chip memory.

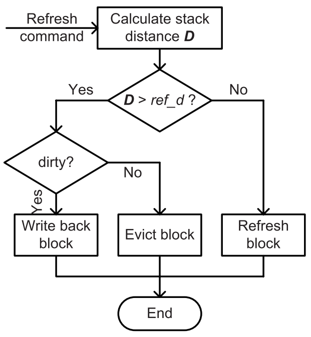
In this work, we propose to maintain coherent accesses to not-refreshed blocks using the metadata stored in SRAM tags and directory entries. In the CMP architecture shown in Figure 3, the tag and directory are implemented with SRAM. Therefore, the metadata is still valid even though the data has expired. We modify the cache coherence protocol to maintain cache coherence even though the data stored in the block has expired. Specifically, when the directory controller receives coherence requests for shared blocks with expired data, the corresponding requests will be handled using three-stage routing. Coherence requests are forwarded to the nearest L1 owner according to the directory information stored in the SRAM. Therefore, using SRAM to store tag and directory information not only reduces energy consumption, but also facilitates the maintenance of cache coherence. In the following, we present the modified cache coherence protocol.

For the purpose of maintaining coherent accesses to expired blocks, a novel coherence state W is added for L1 caches to indicate that there are replicas of the block in L1 caches. Different from the S state, L1 block in the W state is not allowed to be silently evicted. The L1 block in the W state should be explicitly written back by the cache management policy. On one hand, the L2 cache can still obtain the correct copy in case that the corresponding data in the L2 cache has vanished. On the other hand, the other requestors can still obtain the correct copy from this L1 block in the W state.

The extended coherence protocol is called $MWESI$ protocol for short. Several coherence state transitions between the W state and other states are added, as indicated by the solid arrows in Figure 5. In addition, the state transitions, $M \rightarrow S$ and $E \rightarrow S$, are removed from conventional $MESI$ protocol [Sorin et al. 2011]. In the modified coherence protocol, an L1 block in the I state will be transitted to the S state only if the block has one or multiple replicas in the other L1 caches, as depicted in Figure 5. If there are no replicas in L1 caches (privately owned by L2 cache), the state will be changed to the W state. The L1 data request will be served with three-hop coherence mechanism

```
if getDistanceToMRU(block(addr)) > ref_d
    if block(addr).dirty == true;
        write_back(block(addr));
        return;
    else
        silient_evict(block(addr));
        return;
else
    refresh(block(addr));
    return;
```

(a) Pseudocode.



(b) Flow chart.

Fig. 6. Static approach for private blocks.

if the target L2 data has expired. Note that the tag and directory information stored in SRAM will ensure correct cache coherence operations even though the corresponding data has disappeared.

Note that the introduced *W* state is different from the *O* state in *MOESI* protocol. A block in the *W* state must be clean while a block in the *O* state could be dirty [Sorin et al. 2011]. The dash arrows in Figure 5 represent the state transitions in conventional *MESI* protocol while the solid arrows indicate the state transitions between the *W* state and other states. Under CCear, each write back from the L1 cache will reset the *valid_bit* of the corresponding L2 block to 1 in addition to the coherence state transition. Through such reset operations, the subsequent requests to this block before the next refresh operation can directly obtain the data from the L2 cache without three-hop coherence handling.

The objective of adding the *W* state is to make sure that the L2 block with expired data will eventually receive the correct data through writing back the L1 replica in the *W* state. In conventional *MESI* protocol, the L1 block in the *S* state is allowed to be silently evicted without noticing L2 cache. If all of the replicas in L1 caches are evicted silently, the L2 cache will lose the correct data permanently, and the corresponding L2 block will be in a faulty state. CCear only allows one L1 block in the *W* state. On the condition that the L1 block in the *W* state is written back to L2, the L2 cache controller will forward a control message (indicated with $LLC - FwdW$) to the nearest L1 replica if replicas of this block exist in L1 caches. Upon receiving the control message, the corresponding L1 cache controller will switch the state of the replica to the *W* state, as shown in Figure 5. Subsequently, L2 cache can obtain the correct data from the L1 cache with the nearest replica.

3.3. CCear Policy for Private Blocks

Filtered by L1 caches, many cache blocks in LLC will never be reused or have a very long reuse distance after loading from main memory [Khan et al. 2010b, 2010a, Liu et al. 2008; Jaleel et al. 2010], and these blocks will become private blocks. These private blocks will be dead eventually or have a very long dead time until the next reuse. For the purpose of mitigating the unnecessary refreshes to private blocks, we propose to leverage the LRU information of each cache set to refresh private L2 blocks. The detailed refresh policies for private blocks under CCear are illustrated in Figure 6. The principle of refreshing private blocks is as follows.

Each L2 cache slice is associated with a counter, represented by *ref_d*. This counter is utilized by CCear to refresh private blocks, as shown in Figure 6. Upon receiving a refresh command to a private block with address *addr*, CCear will first calculate the distance between this block and the MRU position of the current cache set, as depicted in Figure 6. CCear can obtain the distance through interacting with the cache management policy. If the distance is larger than *ref_d*, CCear will write back or silently evict the block according to its coherence state indicated in lines 1∼7. Otherwise, the block will be refreshed, and subsequent requests can directly find the block in the L2 cache. If *ref_d* of each L2 slice is set with appropriate value, the unnecessary refreshes to private blocks could be effectively eliminated to save energy and reduce the conflicts to normal accesses. In this article, we propose one static approach and one dynamic approach to set the *ref_d* counter. The static approach will assign *ref_d* counter with a predefined value while the dynamic approach will dynamically set the *ref_d* counter to one of two predefined values.

3.3.1. Static Approach. A straightforward way is to set *ref_d* to a fixed value. The fixed value should be prudently chosen. On one hand, a small *ref_d* will make CCear evict private blocks near the MRU position, as demonstrated in Figure 6. Such evictions may be premature and detrimental to performance because subsequent requests to these blocks have to re-fetch the data from main memory. On the other hand, a large *ref_d* will refresh many private blocks even though these blocks are already dead. In the static approach, we choose to set *ref_d* to the value equal to half of the cache associativity. The static approach works well for most of the workloads, as shown by the experimental results in Section 4.

3.3.2. Set-Dueling-Based Dynamic Approach. The advantage of the static approach is its simplicity in implementation. However, different workloads and different program phases typically exhibit varied locality. The static approach is not adaptable to the varied workloads and may lead to either premature or overdue private block evictions. In order to mitigate the performance impact caused by inappropriate private block refresh decisions, we propose to utilize the set-dueling mechanism [Qureshi et al. 2008] to dynamically tune *ref_d* along with the varied workloads. For each L2 slice using set-dueling, the value of *ref_d* is adaptively set to one of the two predefined values stored in *ref_d0* and *ref_d1*. In the following, the details of setting *ref_d* using set-dueling mechanism are illustrated.

Figure 7 shows the architecture of the proposed set-dueling mechanism for a sample L2 slice with 16 cache sets. Here, the purpose of set-dueling is to adjust the value of *ref_d* in the CCear refresh policy for private block between two choices, *ref_d0* and *ref_d1*. As shown in Figure 7, there are four cache sets (set 0, 5, 10, and 15) dedicated to CCear with *ref_d0*. Another four cache sets (set 3, 6, 9, and 12) are dedicated to CCear with *ref_d1*. In addition to these eight cache sets illustrated, the remainder cache sets are called *follower sets*. The *ref_d* for the follower sets is determined by set-dueling and the detailed mechanism is as follows.

One miss to a not-refreshed private block in a cache set dedicated with *ref_d0* will increase the saturating counter, PSEL [Qureshi et al. 2008], by one. By comparison, one miss to a not-refreshed private blocks in cache sets dedicated with *ref_d1* will decrease PESL by two. Here, the method to decrease the PESL counter is different from the original method [Qureshi et al. 2008]. On one hand, we only consider the misses to the not-refreshed private blocks in this work, while the general cache miss statistics are utilized in Qureshi et al. [2008]. On the other hand, PESL is decreased by two instead of one, as shown in Figure 7. If the MSB (most significant bit) of PSEL is 1, the follower sets will apply *ref_d0* for CCear. Otherwise, the value of *ref_d* for the follower sets will be set to *ref_d1*.

As just illustrated, PSEL is modified by the miss on previous not-refreshed private blocks. This is different from the set-dueling in Qureshi et al. [2008], which utilizes

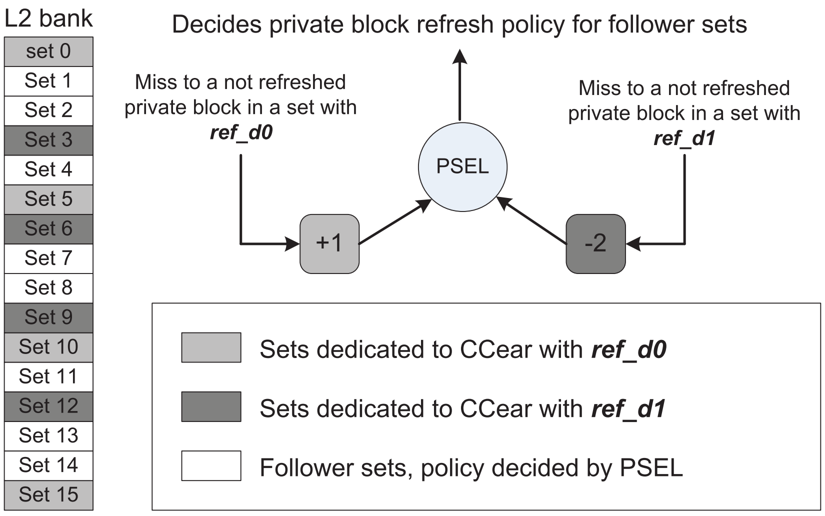

L2 bank
set 0
Set 1
Set 2
Set 3
Set 4
Set 5
Set 6
Set 7
Set 8
Set 9
Set 10
Set 11
Set 12
Set 13
Set 14
Set 15
Decides private block refresh policy for follower sets
Miss to a not refreshed private block in a set with ref_d0
Miss to a not refreshed private block in a set with ref_d1
PSEL
+1
-2
Sets dedicated to CCear with ref_d0
Sets dedicated to CCear with ref_d1
Follower sets, policy decided by PSEL

Fig. 7. Dynamic approach using set-dueling.

the normal miss as the activation event to modify PSEL. In the architecture shown in Figure 3, the tag array in the L2 slice is based on SRAM. When evicting the not-refreshed private L2 blocks, the state of this block is set to NP (not present). However, the tag information for this block is still valid until updated by next write operation to this block. Thus, when a miss occurs, we can determine whether or not the miss is on a previous not-refreshed block through comparison of the tags.

Taking a 16-way set associative cache for example, the potential values for *ref_d* is from 0 to 15. To mitigate the premature eviction of private blocks caused by overly small *ref_d*, we set *ref_d0* to 4. *ref_d1* is always set to a larger value than *ref_d0*, such as 8. Initially, the *ref_d* counter in each L2 slice is set to *ref_d0*. If such setting causes plenty of premature private block evictions, the set-dueling mechanism will detect the excess evictions and reset *ref_d* to a larger value indicated by *ref_d1*.

3.4. Enhance CCear Using Filter Cache

Under CCear, private cache blocks in the LLC could still be prematurely evicted or written back in both the static and dynamic approaches. Subsequent requests to these dropped blocks have to fetch them again from main memory. In order to minimize the costly off-chip memory accesses, we propose to cache the not-refreshed private blocks using a small SRAM-based filter cache. Figure 8 shows the architecture of the tile configured with a fully-associative filter cache. The small filter cache is utilized to mitigate the potentially long latency memory accesses caused by fetching the non-refreshed private cache blocks.

The filter cache is dedicated to accommodate private LLC blocks which are not refreshed by CCear. If CCear does not refresh a private volatile STT-RAM block, the block will be evicted to the filter cache. Note that the private blocks evicted by the cache management policy will be directly written back to the main memory. The filter cache is based on SRAM and managed with LRU replacement policy. The filter cache will be searched if cache access misses in the LLC. The costly main memory access can be eliminated if the corresponding cache block is found in the filter cache. Our objective of utilizing the filter cache is to obtain high performance. If application is not critical on performance, the filter cache can be turned off to save energy.

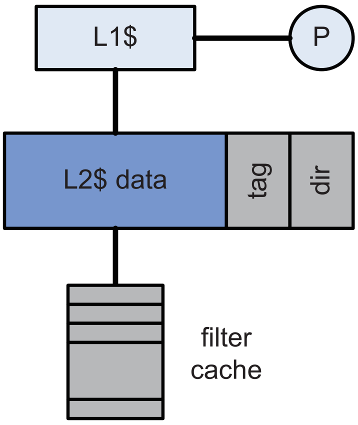


Fig. 8. Tile architecture with filter cache.

4. EXPERIMENTS AND ANALYSIS

We implement an architectural-level simulation to evaluate the CCear scheme presented in Section 3. First, we present the experimental setup, including the simulation infrastructure and the evaluated workloads. Subsequently, the performance of CCear and the corresponding analysis are presented. Afterwards, the efficiency of the filter cache will be evaluated. Then, we present an evaluation of CCear on CMP system with eDRAM based LLC. Finally, the implementation overhead of CCear will be discussed.

4.1. Experimental Setup

We model a 2GHz CMP system, similar to the architecture shown in Figure 3, with 16 in-order cores using Simics [Magnusson et al. 2002], which is an execution-driven full system simulator. Each core is configured with 32KB private instruction/data cache and a shared 512KB L2 slice. We adopt the parameters presented in Smullen et al. [2011] for the volatile STT-RAM as the data store. We have modified CACTI [Muralimanohar et al. 2007] to model cache banks with volatile STT-RAM-based data store and SRAM-based directory and tag store. The adopted SRAM is based on a 65nm technology node provided by CACTI. We modified the MESI directory protocol in GEMS framework [Martin et al. 2005] to support CCear and ensure the consistency of cache hierarchies. We assume the main memory has a fixed round-trip latency of 200 cycles. The memory subsystem is simulated with GEMS [Martin et al. 2005], and the on-chip network energy dissipation is calculated with Orion [Kahng et al. 2009] under 65nm technology node. The details of our simulator parameters are shown in Table I.

We compared the proposed CCear scheme with the ideal refresh policy, the periodic refresh policy used in Smullen et al. [2011], and Cache Revive [Jog et al. 2012]. We utilize energy dissipation and instruction per cycle (IPC) as the performance metrics for the evaluated schemes. For the ideal refresh policy and periodic refresh policy, the LLC bank is implemented using pure volatile STT-RAM. For CCear, the LLC bank is heterogeneous, as shown in Figure 3. The characteristics of the two types of LLC banks are presented in Table II. For the Cache Revive policy, each LLC bank is configured with a buffer of 238 entries [3] The buffer is implemented using low-retention STT-RAM [Jog et al. 2012] which needs to be periodically refreshed to preserve the stored cache blocks.

[3]In Jog et al. [2012], a buffer with 1,900 entries for a 4MB LLC bank yields optimal performance for Cache Revive. In our evaluation, the 512KB LLC bank is configured with a buffer of 238 entries by scaling.

Table I. Main Simulation Parameters

Parameter	Value
Processor	16 UltraSPARC III+ in-order cores 2GHz, Operating System: Solaris 10
L1 I/D Cache	Private, 32/32KB, 2-way, 2-cycle latency 64 bytes block size, write-back
Last-Level Cache	8MB, 16 banks 16-way, 64B block, write-back
Coherence mechanism	*Modified MESI directory protocol*
Main memory latency	200 cycles (round-trip)
Interconnect	4×4 mesh, 128-bit links, 1-cycle link latency 2-cycle router latency

Table II. Characteristics of LLC Bank

Bank type	Read latency	Write latency	Read energy	Write energy	Leakage power	Area
Pure STT-RAM	0.75 ns	2.25 ns	0.28 nJ	1.71 nJ	71 mW	1.16 mm^2
Heterogeneous	0.73 ns	2.21 ns	0.3 nJ	1.65 nJ	78 mW	1.2 mm^2

Table III. Benchmark Characteristics

Workload	Read-Write ratio	Data sharing	Application domain
blackscholes	1.9 : 1	low	Financial analysis
bodytrack	4.1 : 1	high	Computer vision
canneal	2.0 : 1	high	Engineering
facesim	2.3 : 1	low	Computer animation
raytrace	1.2 : 1	high	Rendering
streamcluster	71 : 1	low	Data mining
swaptions	4.8 : 1	low	Financial Analysis
x264	3.3 : 1	high	Media processing

We simulated a set of PARSEC applications [Bienia 2011] with different intensities of read/write operations. As depicted in Table III, the selected workloads cover various application areas, including financial analysis, computer vision, etc. In the evaluation, all the workloads are executed with *simmedium* input sets. We fast forward each workload to the parallel region. We evaluated the whole parallel region, known as region of interest (ROI), of the selected workloads.

4.2. CCear Performance and Analysis

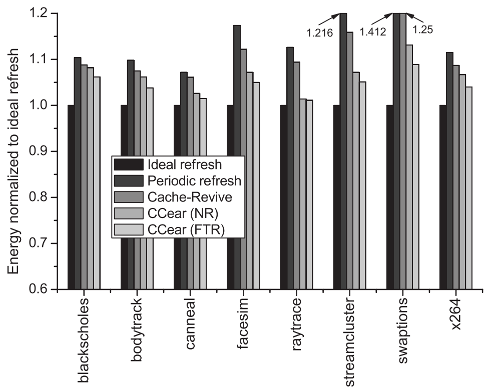
In this section, we first present the energy consumption and IPC of all the evaluated schemes. Subsequently, we present the sensitivity analysis of CCear in FTR policy with different refresh settings. Afterwards, we demonstrate the sensitivity analysis of set-dueling-controlled private block refresh with different *ref_d1* values. Finally, we present an evaluation of periodic refresh policy with $MOESI$ protocol and compare it with CCear based on $MWESI$ protocol.

$$E_{total} = E_{dynamic} + E_{static}; \tag{1}$$

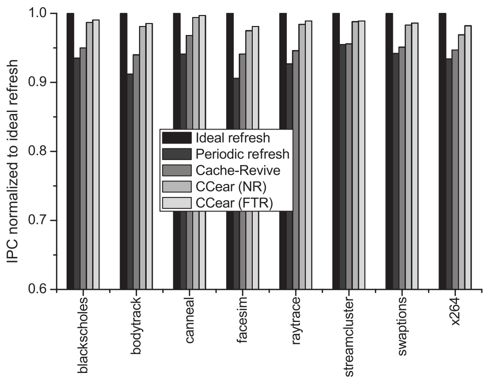
$$E_{dynamic} = N_{read} * E_{read} + N_{write} * E_{write} + NoC_{dynamic}; \tag{2}$$

$$E_{static} = LLC_{static} + NoC_{static}. \tag{3}$$

4.2.1. Energy. We evaluated the energy of LLC, including the on-chip network connecting the cache banks. The energy is calculated using Equation (1). In the proposed CCear policy, there are two refresh schemes for shared blocks (NR and FTR policy)



(a) Impact on energy.



(b) Impact on IPC.

Fig. 9. CCear performance.

and private blocks (static and dynamic approach), respectively. Therefore, there are totally four configurations for CCear. Figure 9(a) shows the energy dissipation of all evaluated schemes normalized to the ideal refresh policy for all the evaluated schemes. For CCear, both the NR and FTR policy are evaluated to show their effectiveness of minimizing refresh operations for shared blocks. In addition, static approach is utilized to refresh private blocks. The fixed number of times N in FTR policy in this evaluation is set to four. The static approach could prematurely evict the private blocks and lead to extra off-chip memory accesses. The extra number of the off-chip memory accesses is recorded. In the evaluation, we assume the main memory is based on DDR3 memory and the estimated read/write energy is 3nJ [Micron Technology 2007]. The energy dissipation of the extra off-chip memory accesses is calculated through multiplying the extra number of off-chip accesses by the read/write energy. The reported energy for CCear shown in Figure 9(a) has incorporated the energy of extra off-chip memory accesses.

Appropriate refresh policies can make better trade-offs between the whole system performance and the system energy dissipation. The always-refresh mechanism with periodic refresh policy consume much more energy compared with the ideal refresh policy. As shown in Figure 9(a), the energy dissipation of the periodic refresh policy is 7.2% to 41.2% more than that of the ideal refresh policy for the evaluated workloads. On average, the periodic refresh policy consumes 16% more energy compared with the ideal refresh policy. For *swaptions*, the energy consumption of periodic refresh policy is significantly increased by 41.2% compared with the ideal refresh policy. This is because the actual energy dissipation of the volatile STT-RAM LLC for *swaption* is significantly lower than the energy consumed in other workload due to fewer accesses to LLC which also represents little on-chip routing energy. Compared with the periodic refresh policy, Cache Revive effectively reduces the energy dissipation, as shown in Figure 9(a). Using the ideal refresh policy as baseline, Cache Revive reduces the energy consumption by 1.6% to 15.9% compared with periodic refresh policy.

As shown in Figure 9(a), both CCear configurations outperform the periodic refresh policy for all the evaluated workloads. Compared with the ideal refresh policy, CCear consumes 1.4% to 13% and 1.1% to 8.9% more energy, respectively; for the evaluated workloads when NR and FTR are utilized. However, compared with the periodic refresh policy, CCear with NR policy reduces the energy dissipation by 2.5% to 28%, taking the ideal refresh policy for reference. CCear with FTR policy further reduces the energy consumption by 3.2% to 32.3% compared with the periodic policy. For CCear, many unnecessary refresh operations to both shared and private blocks are eliminated. This is the key to the energy reduction yielded by CCear against the periodic refresh policy. These two CCear configurations also outperform Cache Revive for all workloads, as demonstrated in Figure 9(a). Compared with Cache Revive, CCear reduces the energy consumption by 5.2% and 7.5% on average, respectively, when the NR and FTR policy are utilized. Cache Revive can also effectively reduce the refresh operations. However, Cache Revive needs to invalidate replicas of the expired shared blocks which could lead to significant invalidation traffic. In addition, the buffer in each LLC bank needs to be refreshed to reserve data. By comparison, CCear can exploit coherence protocol to eliminate these overhead. Moreover, CCear does not need the buffer to store the expired blocks. As a result, CCear saves more energy compared with Cache Revive.

NR policy is simple in design and implementation. It can be considered as a special case of FTR policy. As depicted in Figure 9(a), CCear with FTR policy consumes less energy compared with the NR policy. Many LLC accesses with short reuse distance could directly find the shared blocks if the target blocks are refreshed by the FTR policy. Therefore, compared with NR, fixed times refresh can eliminate many long-latency and energy-consuming three-hop coherence handling.

4.2.2. IPC. Figure 9(b) presents the IPC for the evaluated schemes. Two CCear configurations perform close to the ideal refresh policy for most of the evaluated workloads, as indicated in Figure 9(b). Both CCear configurations outperform periodic refresh policy and Cache Revive for all workloads. The average IPC for periodic refresh policy is 0.93 for the evaluated workloads compared with the baseline. By comparison, the average IPCs for Cache Revive and two CCear configurations are 0.95, 0.982, and 0.988, respectively. The main incentive to better performance of CCear is the reduced conflicts between refresh operations and normal cache accesses. This can be confirmed through checking the IPC of LLC-intensive workloads which have more conflicts. For *facesim*, the IPC of the periodic refresh policy is 0.906, which is normalized to the ideal policy. By comparison, the normalized IPC of two CCear configurations is 0.975 and 0.98, respectively.

CCear with the NR and FTR policy performs better than Cache Revive, as shown in Figure 9(b). Under Cache Revive, not all LLC blocks are refreshed and written back to the buffer or main memory. Thus, subsequent requests to the expired blocks have to re-fetch them from either the buffer or the main memory. In contrast, under CCear, many expired shared blocks have replicas in upper-level caches. Subsequent requests to these shared blocks can obtain the data with low-overhead on-chip routing. Moreover, private blocks with good locality are refreshed by CCear. Thus, subsequent requests can directly access the data in volatile STT-RAM with low latency.

4.2.3. Sensitivity on FTR Times. In order to analyze the impact of the fixed refresh times (N) in FTR policy to system performance, we evaluated CCear with FTR policy under different refresh times using three LLC-intensive workloads. For the purpose of eliminating the interference from private block refreshes, we choose to always refresh the private blocks.

Figure 10 shows the energy dissipation and IPC, normalized to the ideal refresh policy, for the FTR policy under different numbers of refresh times for shared blocks. As shown in Figure 10(a), the energy dissipation increases continually as N becomes larger. Large N signifies more refresh operations as well as more conflicts between normal accesses and refresh operations. Therefore, the energy caused by refresh and on-chip routing will correspondingly increase. In the IPC side, for the evaluated three workloads, refreshing the shared blocks by five to six times yields optimal performance, as shown in Figure 10(b). For *facesim*, the optimal IPC is obtained when N is set to six.

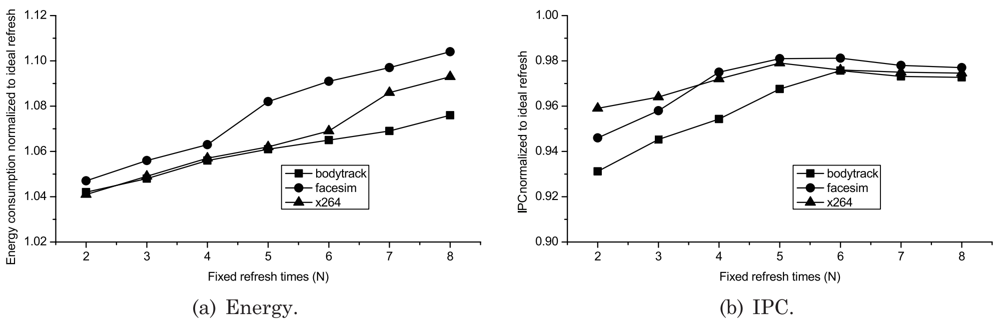


Fig. 10. CCear performance under different FTR settings.

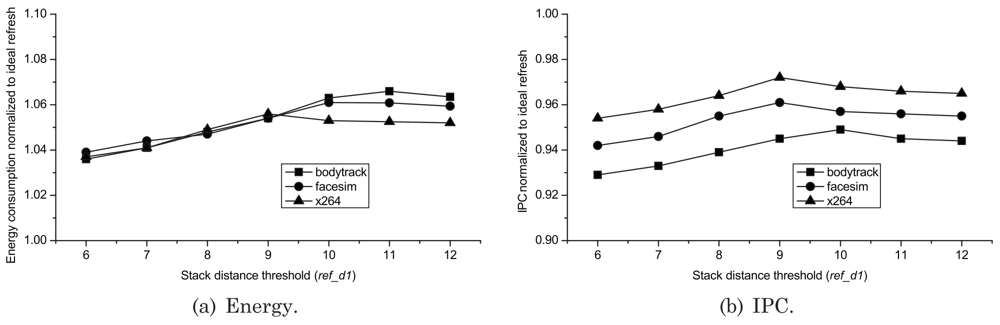


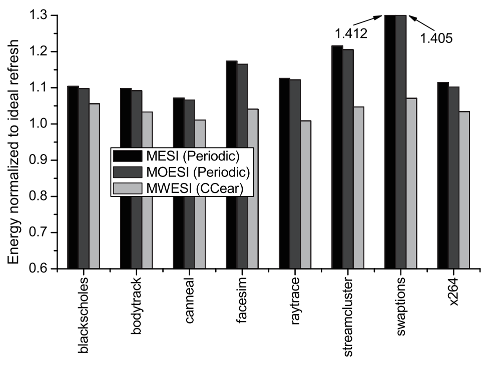
Fig. 11. CCear performance under different *ref_d1* settings.

For energy efficiency, large N should be avoided, because each increment to N represents plenty of increment to energy-consuming shared block refreshes. As illustrated in Sections 4.2.1 and 4.2.2, CCear with FTR policy using four times refresh effectively improved the overall system performance compared with the periodic refresh policy. Combined with the performance trend shown in Figure 10, four to six is the appropriate setup choice for the FTR policy.

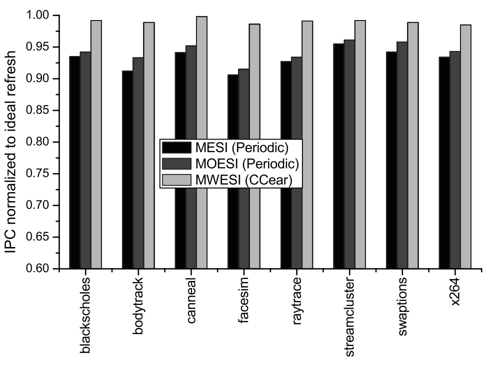
4.2.4. Sensitivity on ref_d1. We also conduct the sensitivity analysis of the *ref_d1* under the dynamic approach. Specifically, we evaluate the performance of CCear with the dynamic approach for private blocks when *ref_d1* is set to different values. During the evaluation, we utilize the NR policy to refresh shared blocks to mitigate the potential interference. Figure 11 depicts the results of the evaluation with three LLC-intensive workloads illustrated previously.

As observed from Figure 11(b), the IPC for the evaluated workloads exhibits two different variation trends as the value of *ref_d1* increases. Specifically, the IPC continually increases along with the increasing *ref_d1* in the first stage. The increasing IPC confirms that the dynamic approach can effectively improve the performance of CCear. In the following stage, IPC smoothly decreases along with the continually increasing *ref_d1*. As shown in Figure 11(b), CCear in the evaluated settings obtains the best IPC for *bodytrack*, *facesim*, and *x264* when the value of *ref_d1* is set to 10, 9, and 9, respectively. The main reason for such variation trend of IPC is as follows.

The number of misses to the blocks which are not refreshed by CCear will decrease as *ref_d1* becomes larger. In addition, the PSEL counter update policy in the set-dueling mechanism adopted in this article is different from the original policy [Qureshi et al. 2008]. The PSEL counter is decreased by two instead of one, as illustrated in Qureshi et al. [2008]. Therefore, for small *ref_d1*, CCear could make the follower sets dedicated



(a) Energy.



(b) IPC.

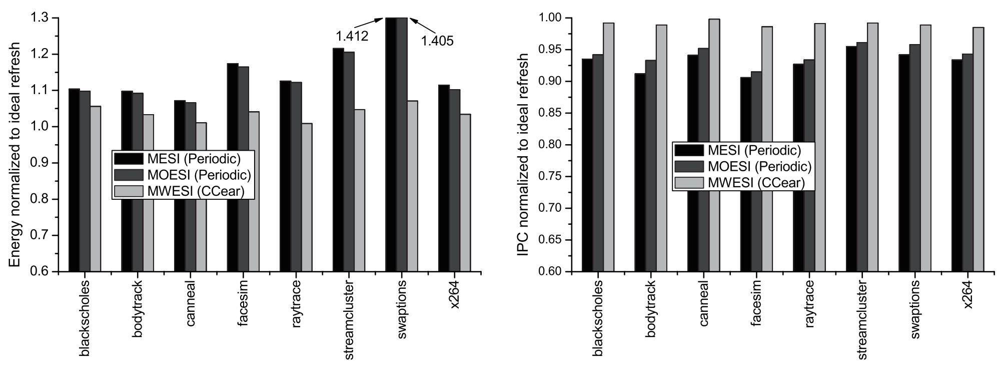
Fig. 12. Compared with MOESI protocol.

to *ref_d1* policy if there are enough misses to the vanished blocks under the given program phase. When *ref_d1* is large enough, most of the private blocks in each cache set will be refreshed by CCear. As a result, the number of misses to the vanished block is usually small, and the set-dueling approach seldom makes the follower sets dedicated to *ref_d1* policy. Compared with smaller *ref_d1*, the number of off-chip accesses to re-fetch the vanished private blocks is more when the *ref_d1* is large. This is the main reason for the degradation of the IPC in the second trend.

The energy dissipation under different *ref_d1* settings exhibits similar two stage trends, as shown in Figure 11(a). CCear consumes the highest energy for *bodytrack* when *ref_d1* is set to 11. For *facesim* and *x264*, the inflection points of energy dissipation are 10 and 9, respectively. The main reason for the increasing trend in energy dissipation is due to the increased refresh operations when the follower sets are dedicated to *ref_d1* policy through set-dueling mechanism. However, when *ref_d1* is large enough, the set-dueling rarely make the follower sets dedicated to *ref_d1*, and the overall energy therefore exhibits slight degradation, as shown in Figure 11(b).

4.2.5. Comparison with MOESI Protocol. The introduced W state is similar to the O state in the MOESI coherence protocol. We conduct an evaluation of periodic refresh with the MOESI protocol and compare it to CCear with the extended MWESI protocol. Figure 12 shows the evaluation results. For the evaluated workloads, CCear outperforms the periodic refresh policy when MOESI protocol is adopted. CCear reduces the energy consumption by 4.2% to 33.4% compared with periodic refresh. On average, CCear reduces the energy consumption by 11.9%. CCear with MWESI protocol has an average IPC of 0.99, while the average IPC of the periodic refresh policy with MOESI protocol is 0.94.

As shown in Figure 12, the periodic refresh policy under the MOESI protocol yields slightly better performance compared with the MESI protocol. This is because the O state can reduce many write-back operations of dirty blocks. As a result, the energy consumption is reduced correspondingly. Moreover, the O state allows a processor to supply the modified data directly to the other processor. This is beneficial under the CMP system wherein the communication latency and bandwidth between two cores are significantly better than the latency and bandwidth between cores and main memory. Therefore, the periodic refresh policy with the MOESI protocol has better IPC compared with the MESI protocol. However, the O state cannot reduce the number of refresh operations for cache blocks, while the main roadblock for volatile memories is the refresh operations.



(a) Energy. (b) IPC.

Fig. 13. CCear performance with filter cache.

4.3. Efficiency of Filer Cache

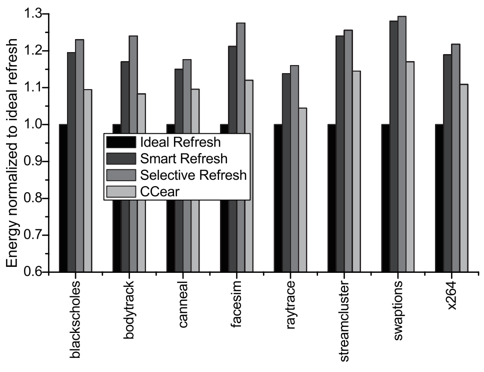
Storing private blocks that are not refreshed by CCear in a filter cache has the potential to reduce the costly off-chip memory access. We implement a set of experiments to evaluate the efficiency of the filter cache. In the evaluation, each LLC bank is configured with a 64-entry fully-associative filter cache which is based on nonvolatile STT-RAM. CCear here use the NR policy for shared blocks and the static approach for private blocks.

Figure 13(b) presents the performance of CCear configured with a filter cache. It is observed that the filter cache can effectively enhance the performance of CCear. With the filter cache, the IPC of the evaluated workloads under CCear further approaches the performance of the ideal refresh policy. This small filter cache improves the IPC of CCear by 1.15% on average, taking the ideal refresh policy as the baseline. At the same time, the filter cache brings about extra energy dissipation. Figure 13(a) shows the energy dissipation of CCear configured with the filter cache. On average, CCear with the filter cache consumes 0.6% more energy compared with CCear without the filter cache. The filter cache can reduce energy consumption by reducing the off-chip main memory accesses. However, the filter cache itself also consumes power to save the expiring private blocks. As indicated by the evaluation results, the energy consumption by filter cache is more than the energy yielded by itself. Therefore, if the application does not have a strict performance requirement, the filter cache could be switched off to save energy.

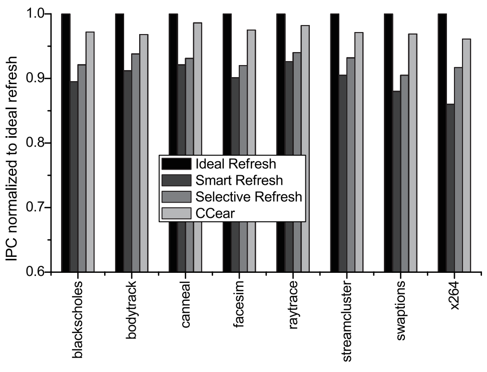
4.4. CCear Performance on eDRAM-Based LLC

Embedded DRAM (eDRAM) has higher storage density compared with SRAM. It is also a promising candidate to replace SRAM as LLC in future high-performance processors. A 32MB eDRAM LLC is utilized in an IBM Power7 [Kalla et al. 2010] processor. In this section, we present an evaluation of CCear on eDRAM-based LLC. The system configuration in the evaluation is similar to Table I, except that the LLC is based on eDRAM. The parameters of eDRAM are obtained using CACTI [Muralimanohar et al. 2007]. We adopt eDRAM cells at 105°C with a retention time of 40 μs reported by Barth et al. [2008]. We compare CCear with Smart Refresh [Ghosh and Lee 2007] and Selective Refresh [Valero et al. 2013]. Figure 14 presents the evaluation results for 16-core CMP with eDRAM LLC.

As shown in Figure 14(a), CCear effectively reduces the energy consumption compared with Smart Refresh and Selective Refresh, Compared with Smart Refresh, CCear yields 5.4% to 11% reduction in energy consumption compared with Smart Refresh,



(a) Energy.



(b) IPC.

Fig. 14. Performance of CCear on eDRAM-based LLC.

taking ideal refresh as the baseline. Moreover, CCear reduces the energy consumption by 8% to 15.7% compared with Selective Refresh. As for IPC, CCear outperforms both Smart Refresh and Selective Refresh, as shown in Figure 14(b). The normalized IPCs for Smart Refresh Selective Refresh, and CCear are 0.9, 0.926, and 0.973 on average, respectively.

Smart Refresh only reduces the refresh operations to the blocks which are recently accessed by either read or write operations. Differently from DRAM with *ms*-level retention time, for low-retention (μs level) eDRAM, the opportunities provided for Smart Refresh to reduce the refresh energy are very limited, because within the retention period, the eDRAM LLC block could be read or written with little probability. In contrast, CCear can effectively reduce the refresh operation through interaction with coherence protocol and cache management policy. For Selective Refresh, the blocks which have only one MRUT will expire. The blocks with more than one MRUT (reused by more than one times) will always be refreshed. In other words, Selective Refresh could guarantee that requests to frequently reused blocks could directly obtain the target data from LLC. This is the reason for its better performance compared to Smart Refresh. However, the performance is obtained at the cost of more refresh operations indicated by more energy consumption compared to Smart Refresh.

4.5. CCear Overhead Analysis

In order to support the proposed CCear policy, the cache coherence protocol needs to be modified. A novel state, W, is added into the baseline $MESI$ directory protocol. If the original bits used for coherence states in the baseline protocol cannot accommodate the new state, one more bit is needed for each entry. The extended $MWESI$ protocol has similar complexity with $MOESI$ protocol. Under NR and FTR policy, three-hop coherence handling may be utilized to fetch the valid data from other L1 caches when the data in corresponding the LLC block has expired. Compared with direct LLC access on the condition that the blocks are refreshed, three-hop coherence handling typically causes more traffic and consumes more energy. However, if the reuse distance of the LLC block is long enough, the benefit gained from the reduced refresh operations can outperform the traffic as well as the energy overhead of three-hop routing.

For the NR policy, each LLC block needs to be associated with an extra bit to indicate whether the block is expired or not. For the FTR policy, besides the valid bit, each LLC block needs a counter to indicate how many times the block has been refreshed. As discussed in Section 4.2, four to six refresh times are appropriate for near-optimal performance. Therefore, four bits are enough to encode the block valid information as the refresh information. The typical block size for last-level caches is 64 B, and the storage overhead of is less than 0.8%.

For the static approach, one counter is needed for each cache bank to indicate ref_d, the predefined distance to MRU position of cache sets. For the LLC with 16 banks adopted in this article, 80 bits are needed. For the dynamic approach, each cache bank needs three counters, as illustrated in Qureshi et al. [2008]. For the 32-bit PSEL counter, the storage overhead in the dynamic approach is 752 bits.

For systems with strict performance requirement, CCear should be configured with filter caches to provide robust performance. Our experimental results show that 64-entry filter cache for each LLC bank could effectively enhance the performance of CCear. For the cache configurations in the proceding evaluation, the storage overhead of the filter caches is 64 KB.

5. CONCLUSION

In this article, we propose an adaptive refresh policy called CCear for volatile STT-RAM last-level caches. Through modifications to cache coherence protocols, CCear can adaptively and effectively minimize the refresh operations to shared LLC blocks. Moreover, for private LLC blocks, CCear interacts with cache management policy to minimize the refresh operations. Experimental results of full-system simulation show that CCear approaches the performance of an ideal refresh policy for a set of PARSEC workloads. In addition, CCear outperforms the periodic refresh policy and Cache Revive [Jog et al. 2012] for all the studied workloads. Moreover, CCear works effectively and efficiently for self-destructive eDRAM-based LLC. Finally, the performance of CCear can be further enhanced using small filter caches to accommodate private blocks.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their tremendous feedback as well as members of our research group for their thoughts and comments on this work at various stages.

REFERENCES

Barth, J., Reohr, W. R., Parries, P., Fredeman, G., Golz, J., Schuster, S. E., Matick, R. E., Hunter, H., Tanner, C. C., Harig, J., Kim, H., Khan, B. A., Griesemer, J., Havreluk, R. P., Yanagisawa, K., Kirihata, T., and Iyer, S. S. 2008. A 500 MHz random cycle, 1.5 ns latency, SOI embedded DRAM macro featuring a three-transistor micro sense amplifier. *IEEE J. Solid-State Circ. 43*, 1, 86–95.

Bienia, C. 2011. Benchmarking modern multiprocessors. Ph.D. Dissertation. Princeton University, Princeton, NJ.

Chen, E., Apalkov, D., Diao, Z., Driskill-Smith, A., Druist, D., Lottis, D., Nikitin, V., Tang, X., Watts, S., Wang, S., Wolf, S. A., Ghosh, A. W., Lu, J. W., Poon, S. J., Stan, M., Butler, W. H., Gupta, S., Mewes, C., Mewes, T., and Visscher, P. B. 2010. Advances and future prospects of spin-transfer torque random access memory. *IEEE Tran. Magnet. 46*, 6, 1873–1878.

Chen, Y.-T., Cong, J., Huang, H., Liu, B., Liu, C., Potkonjak, M., and Reinman, G. 2012. Dynamically reconfigurable hybrid cache: An energy-efficient last-level cache design. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*. 45–50.

Dally, W. J. and Towles, B. 2001. Route packets, not wires: On-chip inteconnection networks. In *Proceedings of the 38th Annual Design Automation Conference (DAC'01)*. 684–689.

Dong, X., Wu, X., Sun, G., Xie, Y., Li, H., and Chen, Y. 2008. Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement. In *Proceedings of the 45th Annual Design Automation Conference (DAC'08)*. 554–559.

Flautner, K., Kim, N. S., Martin, S., Blaauw, D., and Mudge, T. 2002. Drowsy caches: Simple techniques for reducing leakage power. In *Proceedings of 29th Annual International Symposium on Computer Architecture (ISCA'02)*. 148–157.

Ghosh, M. and Lee, H.-H. S. 2007. Smart Refresh: An enhanced memory controller design for reducing energy in conventional and 3D die-stacked DRAMs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'40)*. 134–145.

HOSOMI, M., YAMAGISHI, H., YAMAMOTO, T., BESSHO, K., HIGO, Y., YAMANE, K., YAMADA, H., SHOJI, M., HACHINO, H., FUKUMOTO, C., NAGAO, H., AND KANO, H. 2005. A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-ram. In *Proceedings of the IEEE International Electron Devices Meeting (IEDM'05)*. 459–462.

HU, Z., KAXIRAS, S., AND MARTONOSI, M. 2002. Let caches decay: Reducing leakage energy via exploitation of cache generational behavior. *ACM Trans. Comput. Syst. 20*, 2, 161–190.

JADIDI, A., ARJOMAND, M., AND SARBAZI-AZAD, H. 2011. High-endurance and performance-efficient design of hybrid cache architectures through adaptive line replacement. In *Proceedings of the 17th IEEE/ACM International Symposium on Low-Power Electronics and Design (ISLPED'11)*. 79–84.

JALEEL, A., THEOBALD, K. B., STEELY, S. C. JR., AND EMER, J. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. 60–71.

JOG, A., MISHRA, A. K., XU, C., XIE, Y., NARAYANAN, V., IYER, R., AND DAS, C. R. 2012. Cache revive: Architecting volatile STT-RAM caches for enhanced performance in CMPs. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*. 243–252.

KAHNG, A. B., LI, B., PEH, L.-S., AND SAMADI, K. 2009. ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition*. 423–428.

KALLA, R., SINHAROY, B., STARKE, W. J., AND FLOYD, M. 2010. Power7: IBM's next-generation server processor. *IEEE Micro 30*, 2, 7–15.

KHAN, S. M., JIMÉNEZ, D. A., BURGER, D., AND FALSAFI, B. 2010a. Using dead blocks as a virtual victim cache. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. 489–500.

KHAN, S. M., TIAN, Y., AND JIMENEZ, D. A. 2010b. Sampling dead block prediction for last-level caches. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'43)*. 175–186.

KIM, N. S., AUSTIN, T., BAAUW, D., MUDGE, T., FLAUTNER, K., HU, J. S., IRWIN, M. J., KANDEMIR, M., AND NARAYANAN, V. 2003. Leakage current: Moore's law meets static power. *Computer 36*, 12, 68–75.

KIN, J., GUPTA, M., AND MANGIONE-SMITH, W. H. 1997. The filter cache: An energy efficient memory structure. In *Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture (MICRO'97)*. 184–193.

LI, J., SHI, L., XUE, C. J., YANG, C., AND XU, Y. 2011. Exploiting set-level write non-uniformity for energy-efficient NVM-based hybrid cache. In *Proceedings of the 9th IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia)*. 19–28.

LI, Q., LI, J., SHI, L., XUE, C. J., AND HE, Y. 2012. MAC: Migration-aware compilation for STT-RAM based hybrid cache in embedded systems. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED'12)*. 351–356.

LIU, H., FERDMAN, M., HUH, J., AND BURGER, D. 2008. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'41)*. 222–233.

LIU, J., JAIYEN, B., VERAS, R., AND MUTLU, O. 2012. RAIDR: Retention-aware intelligent DRAM refresh. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*. 1–12.

MAGNUSSON, P. S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HALLBERG, G., HOGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. 2002. Simics: A full system simulation platform. *Computer 35*, 2, 50–58.

MARTIN, M. M. K., SORIN, D. J., BECKMANN, B. M., MARTY, M. R., XU, M., ALAMELDEEN, A. R., MOORE, K. E., HILL, M. D., AND WOOD, D. A. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Architect. News 33*, 4, 92–99.

MENG, Y., SHERWOOD, T., AND KASTNER, R. 2005. On the limits of leakage power reduction in caches. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*. 154–165.

MICRON TECHNOLOGY. 2007. Calculating Memory System Power for DDR3. 2007. http://download.micron.com/pdf/technotes/ddr3/TN41_01DDR3Power.pdf.

MURALIMANOHAR, N., BALASUBRAMONIAN, R., AND JOUPPI, N. 2007. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07)*. 3–14. http://www.hpl.hp.com/research/cacti/.

QURESHI, M. K., JALEEL, A., PATT, Y. N., STEELY S. C. JR., AND EMER, J. 2008. Set-dueling-controlled adaptive insertion for high-performance caching. *IEEE Micro 28*, 1, 91–98.

RASQUINHA, M., CHOUDHARY, D., CHATTERJEE, S., MUKHOPADHYAY, S., AND YALAMANCHILI, S. 2010. An energy efficient cache design using spin torque transfer (STT) RAM. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED'10)*. 389–394.

SMULLEN, C. W., MOHAN, V., NIGAM, A., GURUMURTHI, S., AND STAN, M. R. 2011. Relaxing non-volatility for fast and energy-efficient STT-RAM caches. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*. 50–61.

SORIN, D. J., HILL, M. D., AND WOOD, D. A. 2011. *A Primer on Memory Consistency and Cache Coherence*. Morgan and Claypool.

STUECHELI, J., KASERIDIS, D., HUNTER, H. C., AND JOHN, L. K. 2010. Elastic refresh: Techniques to mitigate refresh penalties in high density memory. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'43)*. 375–384.

SUN, G., DONG, X., XIE, Y., LI, J., AND CHEN, Y. 2009. A novel architecture of the 3D stacked MRAM L2 cache for CMPs. In *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture (HPCA'09)*. 239–249.

SUN, Z., BI, X., LI, H. (HELEN), WONG, W.-F., ONG, Z.-L., ZHU, X., AND WU, W. 2011. Multi retention level STT-RAM cache designs with a dynamic refresh scheme. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*. 329–338.

SWEAZEY, P. AND SMITH, A. J. 1986. A class of compatible cache consistency protocols and their support by the IEEE futurebus. *SIGARCH Comput. Archit. News 14*, 2, 414–423.

TAYLOR, M. B., KIM, J., MILLER, J., WENTZLAFF, D., GHODRAT, F., GREENWALD, B., HOFFMAN, H., JOHNSON, P., LEE, J.-W., LEE, W., MA, A., SARAF, A., SENESKI, M., SHNIDMAN, N., STRUMPEN, V., FRANK, M., AMARASINGHE, S., AND AGARWAL, A. 2002. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro 22*, 2, 25–35.

TEHRANI, S., SLAUGHTER, J. M., DEHERRERA, M., ENGEL, B. N., RIZZO, N. D., SALTER, J., DURLAM, M., DAVE, R. W., JANESKY, J., BUTCHER, B., SMITH, K., AND GRYNKEWICH, G. 2003. Magnetoresistive random access memory using magnetic tunnel junctions. *Proc. IEEE 91*, 5, 703–714.

VALERO, A., SAHUQUILLO, J., PETIT, S., AND DUATO, J. 2013. Exploiting reuse information to reduce refresh energy in on-chip eDRAM caches. In *Proceedings of the 27th International ACM Conference on Supercomputing (ICS'13)*. 491–492.

VALERO, A., SAHUQUILLO, J., PETIT, S., LÓPEZ, P., AND DUATO, J. 2012. Combining recency of information with selective random and a victim cache in last-level caches. *ACM Trans. Archit. Code Optim. 9*, 3, 16:1–16:20.

WU, X., LI, J., ZHANG, L., SPEIGHT, E., RAJAMONY, R., AND XIE, Y. 2010. Design exploration of hybrid caches with disparate memory technologies. *ACM Trans. Archit. Code Optim. 7*, 3, 15:1–15:34.

XUE, C. J., ZHANG, Y., CHEN, Y., SUN, G., YANG, J. J., AND LI, H. 2011. Emerging non-volatile memories: opportunities and challenges. In *Proceedings of the 7th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'11)*. 325–334.

ZHOU, P., ZHAO, B., YANG, J., AND ZHANG, Y. 2009. Energy reduction for STT-RAM using early write termination. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers (ICCAD'09)*. 264–268.

Received December 2012; revised June, August 2013; accepted September 2013