Valero Bresó, A.; Petit Martí, SV.; Sahuquillo Borrás, J.; Kaeli, DR.; Duato Marín, JF. (2015). A reuse-based refresh policy for energy-aware eDRAM caches. Microprocessors and Microsystems. 39(1):37-48. doi:10.1016/j.micpro.2014.12.001.

A Reuse-Based Refresh Policy for Energy-Aware eDRAM Caches

Alejandro Valero[1], Salvador Petit[1], Julio Sahuquillo[1],
David R. Kaeli[2], and José Duato[1],
[1]Department of Computer Engineering, Universitat Politècnica de València[a,*],
[2]Department of Electrical and Computer Engineering, Northeastern University[b]

[a]*Camí de Vera s/n, Valencia 46022, Spain. Tel: (+34) 963877577 ext. 75738*
[b]*360 Huntington Ave., Boston, MA 02115, USA*

Abstract

DRAM technology requires refresh operations to be performed in order to avoid data loss due to capacitance leakage. Refresh operations consume a significant amount of dynamic energy, which increases with the storage capacity. To reduce this amount of energy, prior work has focused on reducing refreshes in off-chip memories. However, this problem also appears in on-chip eDRAM memories implemented in current low-level caches. The refresh energy can dominate the dynamic consumption when a high percentage of the chip area is devoted to eDRAM cache structures.

Replacement algorithms for high-associativity low-level caches select the victim block avoiding blocks more likely to be reused soon. This paper combines the state-of-the-art MRUT replacement algorithm with a novel refresh policy. Refresh operations are performed based on information produced by the replacement algorithm. The proposed refresh policy is implemented on top of an energy-aware eDRAM cache architecture, which implements bank-prediction and swap operations to save energy.

Experimental results show that, compared to a conventional eDRAM design,

*Corresponding author: Alejandro Valero
Email address: alvabre@gap.upv.es (
[1]Department of Computer Engineering, Universitat Politècnica de València)

the proposed energy-aware cache can achieve by 72% refresh energy savings. Considering the entire on-chip memory hierarchy consumption, the overall energy savings are 30%. These benefits come with minimal impact on performance (by 1.2%) and area overhead (by 0.4%).

Keywords: On-chip caches, Reuse information, Selective refresh

---

1. Introduction

Capacitors in Dynamic Random-Access Memory (DRAM) cells store data as different levels of charge, and this charge leaks out over time. The elapsed time since the capacitor was last charged until data contents are lost is referred to as the *retention time*. To avoid data loss due to capacitive discharge, DRAM cell contents are periodically read out and written back in a process known as refresh. Refresh operations consume a significant amount of dynamic energy and can negatively impact performance, since refresh requests compete for memory with regular processor read and write requests. This overhead associated with refresh is expected to grow larger in future technologies given their growing memory densities. For instance, refresh energy consumption is expected to reach nearly half the total consumption of future 64Gb DRAM devices [1].

Prior work has concentrated on reducing refresh energy by avoiding issuing *unnecessary* refresh accesses in off-chip DRAM devices. Regular memory accesses implicitly trigger a refresh operation since DRAM contents are written back after they are read. Prior work [2] has shown we can exploit this behavior by delaying periodic refreshes of frequently requested data. Related work considered inter-cell variation in retention time in order to adapt the refresh period to each memory row [1, 3, 4]. Finally, Error Correcting Codes (ECC) have been also used to recover data lost due to extended refresh periods [5].

Fairly recently, DRAM cells started to be embedded in CMOS technology [6]. These logic-based cells are referred to as embedded DRAM or simply eDRAM cells. Compared to 6T cells implemented with Static RAM (SRAM) technology, eDRAM cells are slower, but they provide much higher density and minimal

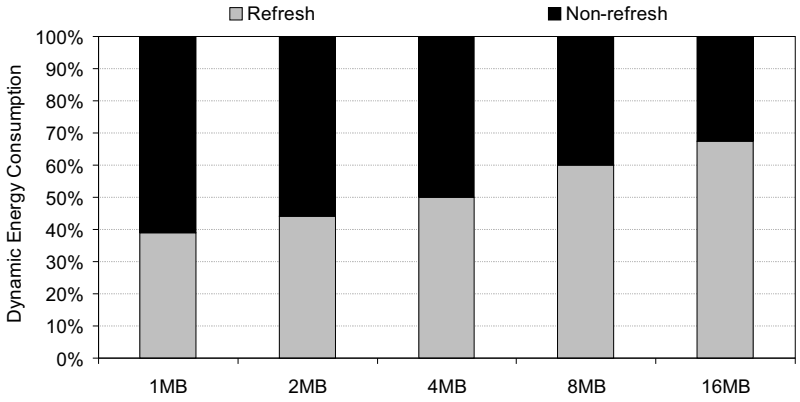


Figure 1: Dynamic energy split into expenses due to refresh and non-refresh operations in eDRAM technology, varying the aggregate L2 capacity.

leakage energy. These features can be better exploited in low-level caches, which represent a massive amount of storage in current and future high-end multicore processors. For example, the aggregated data capacity of second-level (L2) caches in the IBM Power8 [7] is around 6MB. This value is expected to grow larger beyond 16MB in Intel's Knights Landing prototype [8].

Similar to off-chip DRAM devices, refresh operations in eDRAM technology represent an important fraction of the total dynamic energy, as shown in Figure 1[1]. As observed, refresh energy increases with the growth in aggregated cache capacity implemented with eDRAM and grows by up to 67% for an overall capacity of 16MB. Considering all the L2 processor caches, more lines have to be refreshed during the same retention time, which in eDRAM caches is typically thousand times shorter than in off-chip DRAM memories [9]. Nevertheless, eDRAM refresh consumption continues to be significant in embedded single-core processors such as the Samsung S5L8900 used in Apple's iPhone devices [10], or even in devices with relatively lower eDRAM capacities (i.e., 2MB), such as

[1]Results have been obtained with the machine parameters and methodology described in Section 5.

the Sony's PlayStation Portable (PSP) [11].

The overhead of performing refresh in eDRAM caches has been recently noted by other researchers. A common approach to reduce this overhead is to lower the impact of inter-cell variability on refresh energy [12, 9, 13]. Other work has considered using time-based dead-block predictors [14] and cache block state [15] for filtering refresh requests.

It is generally accepted that only a small subset of L2 cache lines are reused (see Section 3), which means that data locality in L2 caches is much lower than in L1 caches. This behavior has been successfully exploited in recently proposed smart replacement strategies [16, 17, 18, 19]. These mechanisms perform better than Least Recently Used (LRU) replacement, selecting a victim block that is not likely to be reused again, which is normally chosen (e.g., randomly) from a set of candidates. In other words, this class of algorithms speculatively identifies those blocks within a cache set that exhibit poor locality and can be considered as candidates for eviction.

This paper describes a novel strategy that reduces refresh energy in low-end single-core processors. Our approach uses the same information that is used by replacement policies to discern whether a block should be refreshed or not in the L2 cache. In this context, the proposed selective refresh policy is integrated with a state-of-the-art Most Recently Used-Tour (MRUT) replacement algorithm [18].

To further increase energy savings, the refresh policy is evaluated with our proposed energy-aware cache, hereafter referred to as the *enaw* architecture. As part of this strategy, we allow blocks in different banks to be swapped. This enables the MRU blocks to be placed in the same bank. Cache lines stored in this bank are always refreshed and accessed first by using a technique referred to as *bank-prediction*.

Experimental results show that, compared to a conventional eDRAM cache using a typical refresh mechanism, the proposed *enaw* approach with selective refresh reduces refresh energy on average by 72%, whereas the overall on-chip memory hierarchy energy savings are up to 30% on average. These benefits come

at the cost of minimal performance degradation and area overhead. Moreover, compared to an energy-aware phased eDRAM cache with typical refresh, the proposed *enaw* cache reduces the refresh energy consumption by more than 50%, while also improving performance. Finally, the proposed cache architecture achieves the best Energy-Delay-squared Product (ED^2P) among the studied schemes.

The remainder of this paper is organized as follows. Section 2 provides related research. Section 3 presents the MRUT replacement algorithm. Section 4 introduces the *enaw* eDRAM cache architecture with the selective refresh policy. Section 5 analyzes our experimental results, including energy, performance, ED^2P, and area. Finally, Section 6 summarizes the paper and discusses directions for future work.

2. Related Work

Refresh performance for on-chip eDRAM caches is not as easy to optimize as compared to off-chip DRAM devices since we cannot adopt most existing off-chip refresh techniques. First, the access time of external Dual In-Line Memory Modules (DIMMs) is at least 6 orders of magnitude faster (from ns to ms) than the next level of the hierarchy (e.g., disks). Thus, less aggressive techniques should be used since a very long disk access is required on a misspeculation. Second, main memory is not organized as a cache, so optimizations such as way-prediction cannot be applied. Third, external DRAM memory works at a coarser (row or page) granularity, where the size is typically several KBytes.

The refresh overhead problem in on-chip eDRAM caches has been previously addressed, taking into account inter-cell feature variations [12, 9, 13]. This prior work pursued solutions that are orthogonal to our proposed selective refresh. In [12], the authors propose to learn the appropriate refresh period from each cache set via a regressive process. Initially, this process assumes the worst-case refresh period for the entire cache. Then, refresh periods are increased step-by-step until ECC detects data losses. In this way, the proper refresh period for

each cache set is selected.

In [9], an ECC optimization approach is proposed to identify expired data in enlarged refresh periods. This approach provides both single-bit and multi-bit failure detection. The single-bit error can be corrected, while those cache sections with multi-bit errors are disabled to avoid the high latency and complexity of multi-bit error correction.

The Mosaic [13] architecture minimizes the number of refresh operations required by exploiting the fact that cell retention times of eDRAM caches exhibit spatial correlation. The cache is divided into regions with different retention time requirements, and the contents of each region are refreshed at different rates using counters in the cache controller.

Chang et al. [14] describe a mechanism that skips refresh operations in those blocks marked as *dead* by a time-based dead-block predictor. The refresh mechanism requires four control bits per cache block and three additional bits per cache set. Blocks are refreshed depending on the predictor accuracy, which is controlled by a state machine. If the accuracy for a given block is low, then the elapsed time from the last access to the block until the time it is considered useless is increased. On the other hand, high accuracy means that a reasonable elapsed time has been reached. In contrast to this work, instead of making decisions based on the elapsed time from the last access, our refresh policy exploits reuse information in terms of how many times a block has been accessed, and does not require any predictor assistance nor state machine.

In [15], periodic refreshes are delayed taking into account the implicit refresh incurred by regular accesses. This mechanism is orthogonal to our refresh policy. Prior work has also proposed a refresh policy that is aware of the block state. *Refrint* requires a 5-bit counter per cache block. The counter is set to its maximum value when a block is accessed or written back, and it is decremented on each periodic refresh to that block. When the counter reaches zero, the block is written back and refreshed if dirty, or it is invalidated in case it is clean.

The 3T1D-based L1 data cache [20] makes use of the information given by the LRU replacement algorithm to save refresh energy. This approach modifies

the cache controller to allocate the MRU data blocks in those lines identified with the longest retention time.

The Cache Decay [21] scheme was proposed for L1 SRAM caches to reduce leakage energy. This approach turns off those blocks that are not being used according to a given value (threshold) of processor cycles. The mechanism works based on the fact that due to L1 data locality, hits in a cache block concentrate in bursts of accesses, which are followed by a long period where the block is not accessed until it is evicted. However, this behavior is uncommon in lower-level caches because L1 caches filter many accesses to L2. Unlike this Cache Decay, the goal of our work is to reduce refresh energy in eDRAM caches that minimize leakage by design.

Finally, The Drowsy Cache [22] scheme also focuses on L1 SRAM cache lines. This approach reduces the supply voltage of selected cache lines while preserving their state. This technique is periodically applied in those lines that have not been accessed during a sampling period of time (measured in processor cycles). In contrast, in our proposed adaptive refresh policy, the sampling period is established by tracking cache misses, while the decision to skip the refresh of a line is determined based on the number of misses that hit in the tag array and by the specific information given by the MRUT replacement algorithm.

3. Exploiting Reuse Information: MRU-Tour

Reuse information has been widely investigated in the past to improve cache performance, especially in L1 data caches [23, 24, 25]. This section describes the MRUT replacement policy that was originally devised for highly associative SRAM low-level caches [18]. This policy is used in this work to drive refresh decisions, that is, to discern which blocks do not need to be refreshed.

The MRUT algorithm works on the MRUT concept, which is defined as the number of times that a block becomes the MRU while it is in cache. Figure 2 illustrates this concept for a generic block A during its generation time. This time starts when A is fetched into the cache ($t1$), and finishes when the block

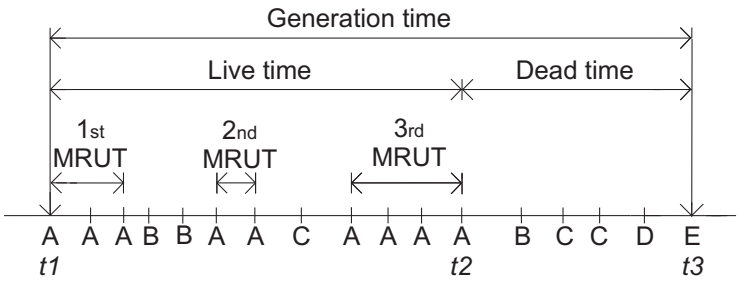


Figure 2: Generation time of the cache block A.

is replaced ($t3$). The generation time consists of live and dead times. The live time refers to the elapsed time since the block is placed in the cache until its last access at time $t2$ before replacement, while the dead time comprises the time from its last access until eviction. The first MRUT period of A starts when it is first brought into the cache, continues while the block is being accessed in the MRU position, and ends when another block, say B, is accessed and becomes the MRU block, forcing A to leave this location. In the example, after accessing B block A is accessed again, so it returns to the MRU position and starts its second MRUT period.

Based on the observation that most of the blocks exhibit a single MRUT at the time they are evicted when managed by a typical LRU algorithm, the MRUT policy achieves a cost-efficient replacement strategy. The MRUT policy works as follows. It requires a control bit per cache block, referred to as the MRUT-bit, to indicate if the block has experienced one or multiple MRUTs. Each time a block is fetched into the cache, its MRUT-bit is set to zero to reflect that an MRUT period has started. If the block is accessed while it is in a non-MRU position, it returns to the MRU location and a new MRUT period starts. This is indicated by setting the MRUT-bit to one, meaning that the block possesses good locality (multiple MRUTs). The victim block is randomly selected among those that have their MRUT-bit set to zero, excluding the last y accessed blocks. This filters stale information, using only recent information, since the LRU stack order is kept for just these y blocks. In case that all the blocks in the same

```
Algorithm: MRUT
Cache hit in block x:
  if(x is not the MRU block)
    set the MRUT-bit of x to 1
Cache miss:
  a) select the block to be replaced
    if (there are candidates with MRUT-bit=0)
       randomly among candidates (except the last y referenced blocks)
    else
       randomly among all blocks (except the last y referenced blocks)
  b) set the MRUT-bit of the incoming block to 0
End Algorithm
```

Figure 3: MRUT replacement algorithm.

set exhibit multiple MRUTs, the victim block is selected among all the blocks (except the last y referenced blocks). Figure 3 shows our implementation of this algorithm. In addition, to adapt the replacement algorithm to dynamic changes in the working set, the MRUT-bits of all the cache blocks are periodically set to zero at runtime in those applications exhibiting few cache misses (e.g., MPKI < 10). Please refer to [18] for further details.

4. *Enaw* Cache Architecture

This section describes the proposed *enaw* eDRAM architecture, which relies on two main design features to save energy: i) bank-prediction cache and ii) selective refresh. In addition we consider swap operations of blocks in different banks to make bank-prediction more performance effective. These enhancements are discussed below.

4.1. Bank-Prediction

Parallel access to all the tags and cache ways of the selected set helps enhance performance, but can be energy inefficient, especially in the context of highly-

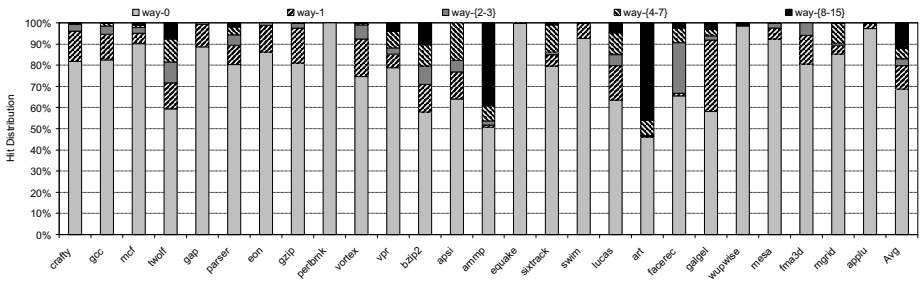


Figure 4: Percentage of cache hits across the ways of a 128B-line 2MB-16way L2 cache using LRU replacement.

associative eDRAM caches where all the ways must be written back on each access after a (destructive) read. Previous work has addressed this problem by predicting which way should be accessed first. Some of these mechanisms [26, 27, 28] have been deployed in commercial microprocessors. These schemes, especially when applied in L1 caches, suffer minor performance losses since data locality is high in these memories [29]. However, data locality is much less predictable in L2, so way-prediction schemes lead to unacceptable performance losses. To deal with this drawback, we propose a method to predict a group of ways (instead of only one) to be accessed first.

In order to discern how many cache ways should be accessed (i.e., predicted) in a first step, we analyzed the hit distribution across the cache ways for a 2MB-16way L2 cache with the LRU algorithm. Figure 4 shows the results. Labels *way-0* and *way-1* refer to the cache way storing the MRU block and the following MRU block, respectively. Label *way-15* refers to the way holding the LRU block. Several ways have been grouped together for illustrative purposes.

On average, 69% of the cache hits land in *way-0* due to the temporal locality present in L2 caches. This percentage is significantly lower than the hit ratio found in L1 caches, where this value is generally above 90% [29]. This means that, unlike way-predictors for L1 caches, a predictor for L2 caches accessing first the cache way containing the MRU data of the target set would introduce an unacceptable performance drop. However, this problem does not remain a

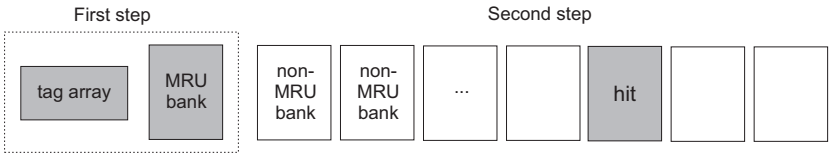


Figure 5: Diagram of the *enaw* cache access.

barrier if we predict the access to more cache ways. Effectively, the percentage of hits increases to 80% when considering an additional way (considering both *way-0* and *way-1*). Using two ways, the hit percentage is above 90% in 15 of 26 benchmarks, though only 6 benchmarks have a 90% or higher hit ratio in the case where only a single way is considered. We analyzed the benefits of predicting an increasing number of ways, but each additional way provides only a marginal benefit as seen in Figure 4. For instance, including *way-2* and *way-3* only increases the hit ratio by 3%. Based on these results, we can conclude that accessing both *way-0* and *way-1* as an initial step should lead to better performance in L2 caches (see Section 5.2).

For analysis purposes, this work assumes that each cache bank of the data array consists of a pair of ways. For example, a 16-way cache has 8 banks. The key idea behind this bank distribution is to keep the pair of ways associated with the MRU and second MRU blocks in the same bank (we will refer to this bank as the *MRU bank*), which is accessed first on every cache access. The bank-prediction technique works as follows. On each cache access, both the tag array and the MRU bank of the data array are accessed in parallel in a first step. Therefore, on a hit in the MRU bank, the access time of the cache is the same as that of a conventional cache. Otherwise, in case of a tag hit associated with any other bank, only the bank containing the target block is accessed in the second step, which starts right after the tag comparison. On a cache miss, just the MRU bank is accessed during the first step and the second step is skipped. Figure 5 plots both steps. Gray color refers to the accessed components in the first and second steps (if any).

Finally, the tag array is assumed to be built with SRAM technology, since implementing this structure with *slow* eDRAM would negatively affect the performance when performing the second step and on a cache miss. Besides, as the tag array is much smaller than the data array, much less leakage and area savings can come from it.

4.2. Maintaining the Block Order Only in the MRU Bank and Swap Operation Details

Our bank distribution scheme produces banks containing the two MRU blocks. This design simplifies replacement logic. The MRUT algorithm maintaining just the order of two ways (*way-0* and *way-1*, -i.e., $y = 2$-) shows roughly the same hit distribution across cache ways as the LRU algorithm. For instance, the percentage of hits in *way-0* is on average by 68%. This percentage grows up to 79% when considering both *way-0* and *way-1*. The MRUT algorithm can be implemented as follows in the *enaw* design. The stack order is maintained just for the blocks stored in the MRU bank, which are not selected for replacement to leverage the freshness of the information. The remaining blocks are considered as candidates for eviction in case they experienced a single MRUT (MRUT-bit=0). Hardware complexity is significantly reduced with respect to the LRU algorithm, since blocks in the MRU bank only require a pair of control bits per cache block: the MRU-bit (to track the position in the LRU stack) and the MRUT-bit (to indicate whether the block has multiple MRUTs or not), whereas blocks stored in *non-MRU banks* do not require any status bit apart from the MRUT-bit. Figure 6 depicts a block diagram of the proposed cache with a possible set of values of the control bits used by the MRUT replacement algorithm.

In order to keep the pair of ways that hold the MRU data blocks of each set in the same bank, the cache controller is enhanced to implement the swap of blocks between different banks. Figure 7(a) and Figure 7(b) illustrate the data transfers performed during the swap operation that arise in non-MRU hits and cache misses, respectively. On a cache hit in a non-MRU bank, the

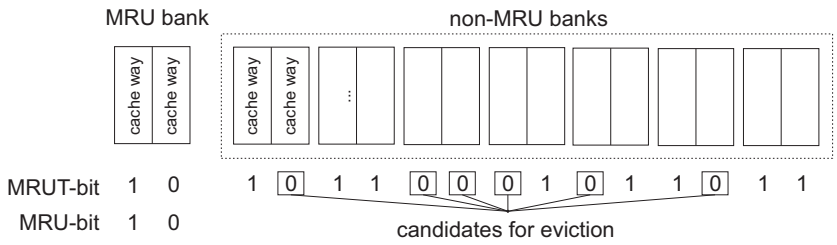
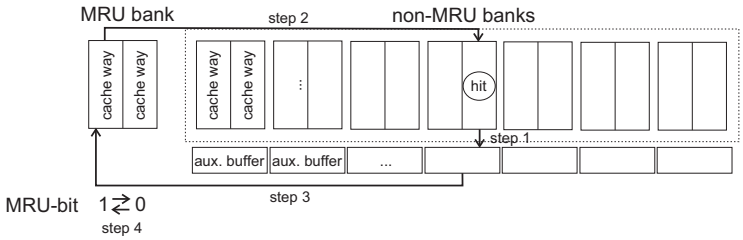


Figure 6: Diagram of the data array of the *enaw* cache with the control bits of the MRUT algorithm.

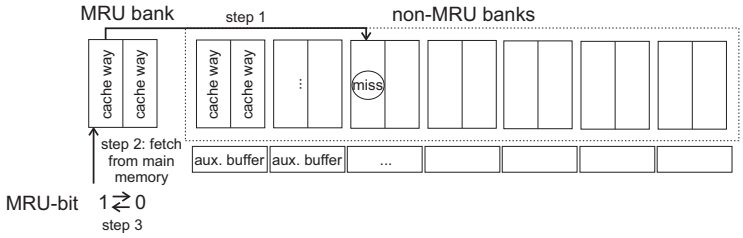
target block is temporarily stored in an auxiliary buffer associated with the non-MRU bank[2] as shown in the first step of Figure 7(a). The LRU block of the MRU bank (second MRU block) is transferred to this non-MRU bank (step 2). Then, the target block is moved from the intermediate buffer to the MRU bank, and becomes the MRU block (step 3). Notice that after this step the LRU stack must be updated accordingly (step 4). On a cache miss, the LRU block of the MRU bank moves to the non-MRU bank which contains the victim block according to the MRUT algorithm (step 1 of Figure 7(b)), whereas the requested block is fetched from main memory and stored in the MRU bank (step 2). Finally, the LRU stack is updated (step 3). On a hit in the MRU bank, no data movement between ways or banks is performed, instead the LRU stack is updated if necessary.

The design assumes that tags are not swapped. For 16-way caches, four status bits per tag are required to maintain the mapping between tags and cache ways, which is similar to the technique used in [30]. Note that the access to these status bits is not in the critical path since they are read together with the tag array and the MRU bank in the first step of the *enaw* cache access (see Figure 5). The area overhead introduced by these control bits, as well as the auxiliary buffers, is minimal as analyzed in Section 5.3.

[2]The proposed design includes as many buffers as non-MRU banks to permit bank parallelism.



a) Non-MRU hit



b) Cache miss

Figure 7: The different steps involved in the swap operation.

4.3. Refresh Mechanism

The cache memory accesses and swap operations implicitly refresh the eDRAM contents. However, as the storage cells lose their charge over time, if some form of refresh is not performed, rarely accessed data will be lost. In such a case, if the data are later requested, a clean copy can be fetched from the main memory (assuming a write-through policy), but this negatively impacts both performance and energy consumption. To avoid data losses, refresh cycles are scheduled using either a distributed or burst method in typical DRAM memories [31]. This work assumes a distributed refresh method as the baseline since it is the most commonly used method.

Refresh operations are performed at a cache block granularity, so that the refresh period can be established by computing the shortest charge retention time divided by the number of blocks in the cache. This guarantees that all the blocks are refreshed ahead of losing their contents. This work simulates logic-

compatible eDRAM technology using 10fF trench capacitors [32] in CACTI [33], which gives a retention time of 190K processor cycles for a 3GHz processor clock and 45nm technology. Notice that such a retention time value is reasonable, since it increases from a $1.3x$ to a $2.1x$ factor with respect to the retention times assumed in previous work [13, 15, 9]. In order to reduce bank conflicts and contention, refresh operations are interleaved across cache banks and follow a round-robin pattern.

In addition to the baseline distributed refresh policy (referred to as *Alw*), two additional selective refresh policies exploiting the MRUT concept have been devised for saving energy. We will refer to these as i) *Cond* and ii) *Adp*. These policies work as follows:

- ***Alw***. This policy always refreshes the target block regardless of the value of the MRUT-bit and the LRU-stack position of the block.

- ***Cond***. Refresh is applied whenever the following condition is satisfied: the target block is stored in the MRU bank or its MRUT-bit='1' (multiple MRUTs). Otherwise the block is marked as invalid and written back to main memory if dirty (*early* writeback).

- ***Adp***. This policy dynamically adapts between *Alw* and *Cond* at runtime.

The *Cond* policy aims to reduce energy waste due to refresh with respect to *Alw*. This policy allows data losses in those blocks less likely to be accessed again (i.e., these blocks are not refreshed). Since *Cond* is a speculative approach, it can lead to performance degradation caused by misspeculation (a block is requested after capacitive discharge). The block must then be fetched from the main memory. This policy uses the position in the LRU stack and the MRUT-bit to decide whether the block should be refreshed or not. If the prediction accuracy is high, the *Cond* policy can achieve substantial energy savings with minimal performance loss. However, if the prediction accuracy is low, this policy can severely impact performance.

Because of *Cond* can threaten performance in the case of low prediction accuracy, the *Adp* policy has been devised to deal with this drawback. The *Adp* policy dynamically selects which of the previous policies will be applied depending on the previous cache behavior. It uses a pair of counters for the whole cache to track the number of standard cache misses (*miss* counter), as well as the number of misses that hit in the tag array when the associated data line has lost its contents (*tag-hit-data-miss* counter). These misses occur when the requested block has been previously invalidated when working under the *Cond* policy, and they are a subset of the number of standard misses.

Initially, the *Cond* policy is selected by default, both counters are reset to zero, and a sampling period starts. This period finishes when the *miss* counter reaches a given value (e.g., 128). At that point, if the *tag-hit-data-miss* counter exceeds a given threshold (e.g., 8), the *Alw* refresh policy is selected during the next sampling period. Otherwise the *Cond* policy continues to be applied. Then, both counters are reset and a new sampling period starts. In the case where the *Alw* policy is used during a given sampling period and the *tag-hit-data-miss* counter does not surpass its threshold, the *Cond* policy is chosen for the next sampling period.

Threshold values of *miss* and *tag-hit-data-miss* counters are defined as a power of two, in order to make hardware simple. That is, it is enough to check just a single bit of each counter to choose the policy to be applied during the next sampling period. Note that the sampling period is independent of the refresh period, which is determined by the retention time and number of blocks.

Figure 8 presents an example that illustrates how the *Cond* policy works[3] in a cache set consisting of two and four MRU and non-MRU ways, respectively. Initially, the cache set does not store any valid data (represented by the mark "–"). Then, blocks A and B are accessed and consequently placed in the two MRU ways. After that, the first periodic refresh is triggered, affecting the MRU way where block A resides. Note that this refresh operation is performed

[3]Of course, it also includes the behavior of *Adp* when the selected policy is *Cond*.

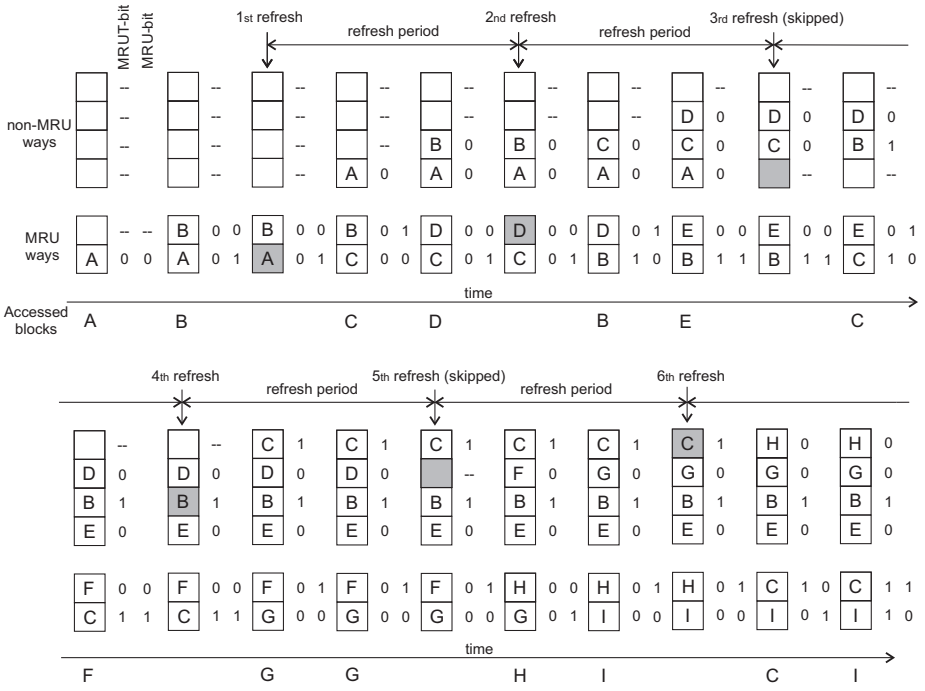


Figure 8: Example of periodic refresh operations performed by *Cond* in the devised energy-aware cache.

regardless of the value of the MRUT-bit corresponding to block A, since the contents of the two MRU ways are always refreshed. Similarly, block D is refreshed in the second periodic refresh.

The third periodic refresh corresponds to a non-MRU way that contains the block A with its MRUT-bit not set, so the refresh operation is skipped and the block is invalidated. This situation also occurs in the fifth periodic refresh, resulting in the invalidation of block D. In contrast, blocks B and C are refreshed (in the fourth and sixth periodic refreshes, respectively) because their MRUT-bit is set. Recall that the MRUT-bit is set whenever a block is accessed while it is not in the MRU position, which occurs for blocks B, C, and I in their second access. Finally, according to the round-robin policy followed by the refresh mechanism, the seventh periodic refresh to this cache set (not shown) would correspond to the first MRU way (assuming in this example that

each cache bank implements a single way in the cache).

5. Experimental Evaluation

Next, we present the simulation environment used to evaluate the studied cache schemes. An extended version of the SimpleScalar simulation framework [34] has been used to model the MRUT replacement algorithm and the devised *enaw* architecture with the selective refresh policies. Leakage and dynamic energy were estimated from the execution time and the required memory events (i.e., cache hits, misses, writebacks, swaps, and refreshes) of the benchmarks, respectively. Bank conflicts and contention due to all these memory events has also been considered. Accesses to different banks can be issued concurrently, but an access to a given bank must wait until the previous access to the same bank finishes. On a swap operation, the banks involved cannot be accessed until the data migration finishes. Access time, leakage currents, dynamic energy per memory event, and area were calculated with CACTI 5.3 [33, 35] assuming a 3GHz processor clock and 45nm technology. The overall energy was estimated by combining the results of the detailed architectural simulator and the energy estimations.

Experiments have been performed for the Alpha ISA with the *ref* input set and running the SPEC CPU benchmark suite [36]. Results were collected simulating 500M instructions after skipping the initial 1B instructions. Table 1 summarizes the main architectural parameters.

5.1. Impact on Dynamic Energy

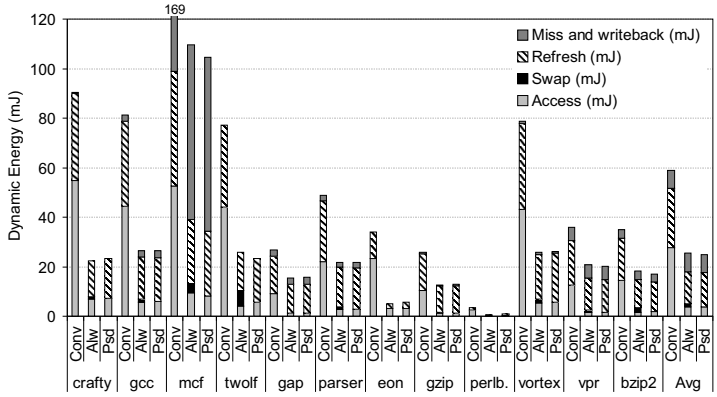
We want to evaluate the dynamic energy savings of the proposed *enaw* eDRAM cache and the refresh policies applied to it. Dynamic consumption has been divided into four major components, according to the type of operation: i) accesses to the L2 cache (*Access* consumption), ii) swap operations between banks in the *enaw* architecture (*Swap* consumption), iii) refresh operations of the L2 data (*Refresh* consumption), and iv) fetched blocks to the L2 cache from

Microprocessor core	
Issue policy	Out of order
Branch predictor type	Hybrid gshare/bimodal: gshare has 14-bit global history plus 16K 2-bit counters, bimodal has 4K 2-bit counters, and choice predictor has 4K 2-bit counters
Branch predictor penalty	10 cycles
Fetch, issue, commit width	4 instructions/cycle
ROB size (entries)	256
# Int/FP ALUs	4/4
Memory hierarchy	
L1 instruction cache	16KB, 2-way, 64B-line, 2-cycle
L1 data cache	16KB, 2-way, 64B-line, 2-cycle
L2 unified cache	2MB, 16-way, 128B-line, 8 banks SRAM tag array: 2-cycle eDRAM data array: 12-cycle MRUT replacement algorithm
Main Memory	100-cycle

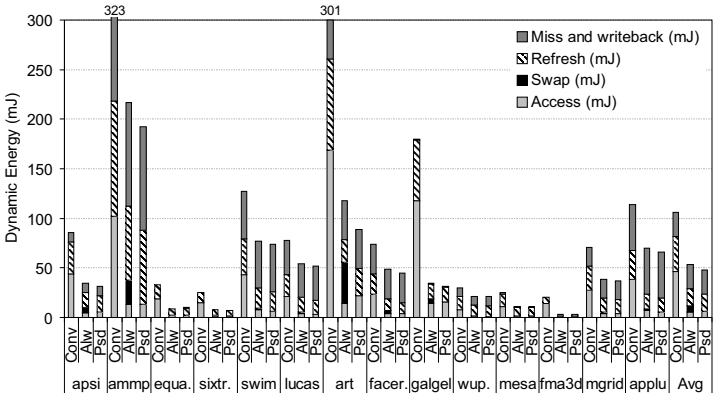
Table 1: Architectural machine parameters.

the main memory and L2 writebacks to the main memory (*Miss and writeback* consumption). The first component covers the energy spent in the access to the L2 cache, including the tag array, data array, and cache controller logic (e.g., decoders, multiplexers, and sense amplifiers). Remember that a swap operation arises in each non-MRU hit. Its consumption has been obtained as the sum of the cost of a read access to the MRU bank, a read access to the target non-MRU bank, a write access to that bank (i.e., step 2 in Figure 7(a)), and a write access to the MRU bank (step 3). The unidirectional transfer performed on each cache miss (step 1 in Figure 7(b)) has also been included in this category as a write access to a non-MRU bank. The L2 refresh consumption considers the energy cost due to refreshing the contents after reads and periodic refresh operations. Finally, a commodity 2GB DRAM main memory is assumed to estimate energy due to L2 misses and L2 writebacks. The number of these L2 memory events differs among refresh policies since they result in different prediction accuracies.

First, we analyze the effects of the devised bank-prediction technique and swap operations. Figure 9(a) and Figure 9(b) plot the dynamic energy (in



a) Int benchmarks



b) FP benchmarks

Figure 9: Dynamic energy (in mJ) of the conventional cache (*Conv*), the *enaw* cache (*Alw*), and the phased cache (*Psd*), all using the baseline refresh policy.

mJ) consumed, for Integer (Int) and Floating-Point (FP) benchmarks, respec-
420 tively. We are evaluating the *enaw* cache working with the previously described *Alw* policy (labelled as *Alw* in the graph). Two additional cache schemes have been considered for comparison purposes. Label *Conv* refers to a conventional eDRAM cache that accesses the tag array in parallel with all the data cache

ways, while *Psd* refers to a phased eDRAM cache that serializes accesses to the tag array and the data array [30]. Thus, the access time includes the tag array latency plus the data array latency, and only the target bank is accessed after the tag comparison. For the sake of fairness, it is assumed that these caches are implemented with the same technology and replacement policy as the proposal. That is, the tag array is built with SRAM technology and the the MRUT replacement algorithm is used. In addition, both cache schemes implement the baseline *Alw* refresh policy.

Results differ wildly across benchmarks for a given refresh policy due to two main factors. First, all the analyzed types of consumption depend on the number of accesses to L2 or main memory. The higher the number of accesses, the higher the energy consumption. Second, applications have different execution times. Those benchmarks with longer execution times significantly increase the refresh energy (e.g., *mcf* and *ammp*).

Compared to *Conv*, the consumption of the *enaw* cache is significantly reduced. Note that in some benchmarks such as *gcc* and *vortex*, only the *Access* consumption of the conventional cache exceeds the total dynamic energy consumption of the proposed *enaw* approach. This is due to a reduced number of accesses and refresh operations are carried out in the *enaw* cache thanks to the bank-prediction technique, which enables the proposed cache to access just the target non-MRU bank after the tag comparison. In addition, although *enaw* wastes energy in swap operations, this consumption is minimal because most of the hits concentrate in the MRU bank (see Figure 11). In fact, the swap energy only represents on average a 11% of the overall dynamic consumption. The highest swap energy overhead can be found in *art*, which is the benchmark with the highest number of non-MRU hits.

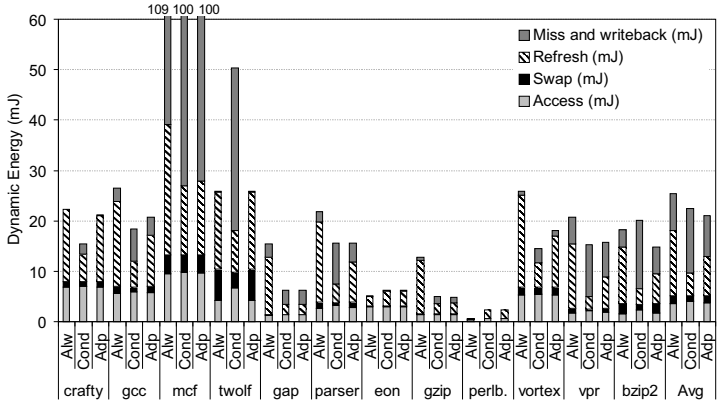
Compared to *Psd*, the *enaw* cache slightly increases the overall dynamic energy. Such energy waste, which can be observed in some of the applications like *mcf*, comes from the bank-prediction inaccuracy (useless MRU bank accesses) that produce a subsequent swap operation. On the other hand, *Psd* consumes slightly more refresh energy than *Alw* because the target bank must be refreshed

on each cache access (*Psd* does not include swaps) and it increases the execution time (see Section 5.2). Finally, notice that for a given benchmark, the *Miss and writeback* energy is the same regardless of the cache design. This is due to the fact that all the studied schemes prevent data loss by refreshing all the cache blocks.

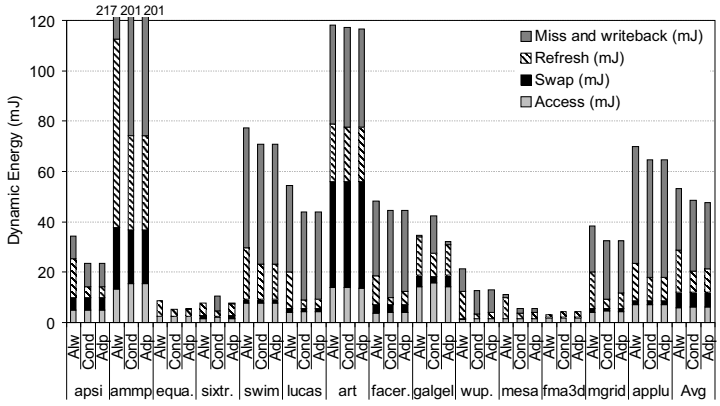
Figure 10(a) and Figure 10(b) compare the effects of the devised refresh policies used in the *enaw* architecture. Both figures show that the *Cond* policy largely increases the consumption in the *Miss and writeback* component with respect to *Alw* due to extra cache misses and *early* writebacks caused by the non-refreshed blocks. Compared to *Alw*, the increase in *Misses and writebacks* observed in *Cond* is balanced out by the associated refresh energy savings. On the other hand, *Adp* slightly increases the refresh energy with respect to *Cond*, but it considerably reduces the *Miss and writeback* energy. Since *Adp* reduces the number of accesses to the main memory, it performs much better than *Cond* (see Section 5.2). Regarding swap energy, the *Cond* policy barely reduces this consumption. This effect can be clearly appreciated in *twolf*. However, these savings are compensated for by the *Access* energy, which increases due to extra misses (requests to non-refreshed blocks and subsequent fetches to L2). The sum of both *Access* and *Swap* costs are uniform across the studied refresh policies and benchmarks. Overall, *Adp* is the most energy-efficient refresh method.

The refresh energy savings of *Alw* and *Adp* are on average 50% and 72%, respectively, as compared to *Conv*. The refresh reduction of *Alw* comes from bank-prediction, while savings of *Adp* are due to both bank-prediction and selective refresh. Taking into account the four components, the overall dynamic energy savings of *Alw* are on average 52% with respect to *Conv*, while the *Adp* method reduces the overall energy by 58%. Compared to *Alw* and *Psd*, *Adp* reduces the refresh energy consumption by 43% and 46%, respectively, which confirms that the proposed selective refresh technique is an effective way to attack the refresh overhead.

Finally, different threshold values for *miss* and *tag-hit-data-miss* counters were analyzed for *Adp*. Results were obtained with a threshold of 128 and 8



a) Int benchmarks



b) FP benchmarks

Figure 10: Dynamic energy (in mJ) of the *enaw* cache using the baseline refresh policy (*Alw*) and the proposed *Cond* and *Adp* selective refresh policies.

for *miss* and *tag-hit-data-miss*, respectively, since this pair is the most energy-efficient[4].

[4]Increasing the *tag-hit-data-miss* threshold reduces the *Refresh* consumption, but the *Miss and writeback* energy increases much more due to additional induced misses and writebacks. In contrast, a smaller threshold value makes the behavior of *Adp* closer to that of *Alw*. For

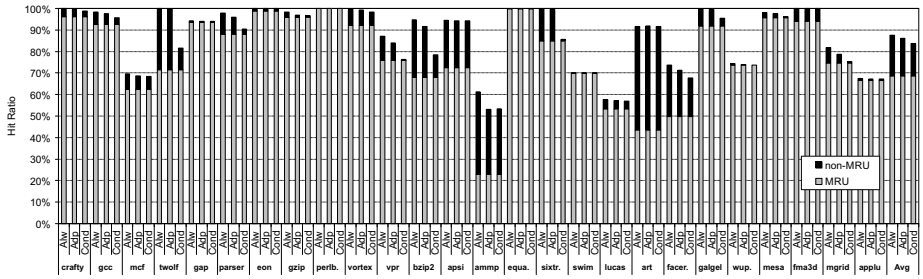


Figure 11: Hit ratio (%) of the *enaw* cache, divided into hits in MRU and non-MRU banks for the refresh policies considered.

5.2. Performance

Figure 11 presents the cache hit ratio of the *enaw* design, with the hit ratio in MRU and non-MRU banks reported individually. Notice that the hit ratio of *Alw* matches the hit ratio of conventional and phased caches, since both refresh methods avoid data losses.

The MRU hit ratio is on average 70%, while this percentage is above 80% in half of the applications. These results illustrate the effectiveness of the bank-prediction and the swap mechanism, since most of the cache accesses hit the MRU bank at the first step.

An interesting observation is that the MRU hit ratio remains constant for each benchmark regardless of the refresh policy used. This is due to the refresh methods considered here assume the same placement/replacement strategies and blocks stored in the MRU bank are always refreshed.

Regarding the non-MRU hit ratio, it is lower for the *Cond* policy than in the *Alw* policy. This is because *Cond* refreshes a small percentage of non-MRU blocks that are reused later. For instance, in a number of the benchmarks (9 of

the *miss* threshold, larger values allow the *Cond* refresh to be applied for a long period, which may lead to severe performance losses. On the contrary, smaller values shorten the sampling period in such a way that the mechanism does not have enough information to decide which is the most appropriate policy for the next period.

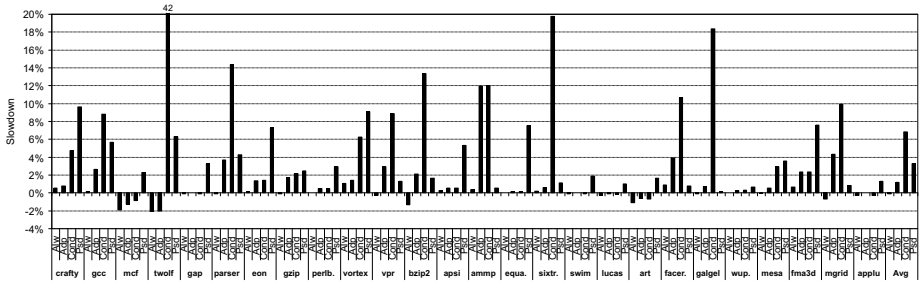


Figure 12: Slowdown (%) of the phased cache and the *enaw* cache for the proposed refresh policies with respect to the conventional cache.

26) such as *parser* or *galgel* experience noticeable non-MRU hit ratio differences, which results in unacceptable performance, as shown in Figure 12. In contrast, the non-MRU hit ratio of *Alw* and *Adp* are quite similar, with the only exception of *ammp*. Moreover, in other benchmarks like *apsi* and *art*, the non-MRU hit ratio remains almost constant (regardless of the refresh policy), which suggests that most of the blocks exhibit multiple MRUTs so that they are refreshed.

Figure 12 plots the slowdown (lower is the better) of the *enaw* scheme using *Alw*, *Adp*, and *Cond* refresh policies and the phased cache scheme (*Psd*) with respect to the conventional cache (*Conv*). On average, the performance loss of *Alw* is roughly the same as that of *Conv*, although differences can be found in some specific benchmarks. For example, in applications such as *crafty* and *vortex*, *Conv* outperforms *Alw*. This is mainly due to the fact that the latter scheme has to wait for the tag comparison before accessing the target non-MRU bank. In other benchmarks like *mcf* and *bzip2*, *Alw* obtains better results than the conventional cache. This is because the latter approach experiences higher bank contention, since on each cache access, all the cache banks are accessed in parallel. Thus, a given memory request may be stalled by a previous access, independent of the target bank. In *mcf* and *art*, this factor severely impacts *Conv*, since both selective refresh policies obtain better performance.

As explained above, the performance differences between *Alw* and the selective policies are mostly due to L2 misses caused by misspeculation. Notice that

employing a cache decay technique that does not refresh any block significantly impacts performance as compare to *Cond*. The slowdown associated with this policy is on average 6.9% higher with respect to the conventional cache. This percentage can be significantly reduced by using a less-aggressive *Adp* policy (only by 1.2%). These results lead us to conclude that: i) having the *enaw* cache access a single MRU bank first (a couple of cache ways) allows us to achieve negligible slowdown and ii) the swap operation does not substantially impact performance.

The *Psd* cache negatively impacts execution time with respect to *Alw* and *Conv*, since it waits for the tag comparison before accessing the target bank on each cache access. Notice too that the *enaw* cache using the *Adp* policy improves the performance with respect to *Psd* in almost all of the applications (i.e., 20 of 26 benchmarks). This means that, even though the misspeculation of the selective refresh can lead to performance loss, the benefits due to bank prediction can counteract the losses. In fact, even the *Cond* policy outperforms *Psd* in most of the benchmarks (e.g., *crafty*, *equake*, and *fma3d*); however, *Cond* experiences more slowdown on average because it is severely penalized by poorly performing applications such as *twolf* and *sixtrack*. Overall, *Adp* improves performance on average by 2.1% with respect to *Psd*.

5.3. On-Chip Memory Hierarchy Energy, Energy-Delay-Squared Product, and Area

This section quantifies the total on-chip memory hierarchy consumption, Energy-Delay-squared Product (ED^2P), and area for the studied L2 caches and both SRAM-based L1 data and L1 instruction caches. Table 2 summarizes results for the Int and FP applications together.

Leakage has been accounted for cycle by cycle during each benchmark execution. It comprises the tag array, data array, intermediate buffers (if any), and remaining cache controller logic. Dynamic energy is the sum of the *Access*, *Swap*, and *Miss and writeback* categories defined above. Total energy refers to the sum of leakage, dynamic, and refresh consumption. The area values (in

	L1 caches		L2 caches				
	Data	Insn.	*Conv*	*Psd*	Enaw		
					Alw	*Cond*	*Adp*
Leakage (mJ)	2.0	2.0	33.1	34.3	33.1	35.6	33.6
Dynamic (mJ)	9.7	30.8	54.3	21.6	25.4	29.9	26.8
Refresh (mJ)	0	0	29.9	15.6	15.0	6.6	8.5
Total energy (mJ)	11.7	32.8	117.3	71.5	73.5	72.1	68.9
ED^2P	–	–	997499	650925	624909	708007	600213
Area (mm^2)	0.27	0.27	8.20	8.20	8.23	8.23	8.23

Table 2: Total energy, ED^2P, and area of the L1 and L2 studied caches.

mm^2) include all the components considered for leakage, plus the overhead of the status bits required to keep the mapping between tags and data blocks, the MRU-bits, and the MRUT-bits. The ED^2P values for each cache architecture were obtained multiplying the corresponding total energy (in mJ) by the squared execution time (in ms).

Compared to the L1 caches, despite the proposed L2 caches are eDRAM-based, the amount of leakage current significantly increases in the eDRAM caches mainly due to their larger capacity. Both conventional and the *enaw* schemes using *Alw* consume the same amount of leakage energy, closely followed by *Adp*. Leakage differences appear due to this energy is proportional to the execution time, and longer execution time (see Figure 12) implies higher consumption. Depending on the cache architecture, the ratio of leakage with respect to the overall energy varies from 28% (Conv) to 49% (*enaw* caches with selective refresh).

The dynamic energy also increases with the storage capacity, although the proposed caches save energy by applying bank-prediction. Compared to the L1 data cache, the L1 instruction cache consumes a larger amount of dynamic energy because it is more frequently accessed. As discussed above, compared to *Conv*, the proposed cache reduces dynamic energy by implementing bank-prediction and swap operations. In addition, the selective policies considerably minimize the refresh costs. This allows *Adp* to be the most energy conservative cache among all the studied schemes. Overall, this approach reduces the total

energy by 41% with respect to the conventional cache. Taking into account all the on-chip memory hierarchy, this percentage is by 30%.

The *enaw* cache using the *Adp* policy is also the best cache design choice from the perspective of the ED^2P (lower is the better). Despite the fact that *Adp* increases the execution time when compared to *Conv* and *Alw*, its overall energy savings allow this scheme to obtain the lowest ED^2P among all the studied caches, even though this metric gives more weight to performance than to energy. Note that although *Alw* consumes more energy than *Psd* (mainly due to the savings of predicting the access to the MRU bank), *Alw* also reduces the ED^2P compared to *Psd* because it performs better.

Finally, the scant area overhead (0.4%) that we see in the *enaw* caches with respect to both conventional and phased approaches is due to the use of intermediate buffers and status bits. Regarding the additional hardware incurred by the proposed *Adp* refresh method, remember that it only requires two counters for the entire cache, and the policy to be applied is selected by checking just a single bit of each counter, yielding negligible overhead.

6. Conclusions and Future Work

This paper has shown that the information used by recent replacement algorithms for highly-associative caches can help designers to efficiently reduce the refresh overhead of eDRAM caches. Selective refresh mechanisms relying on a state-of-the-art MRU-Tour replacement algorithm have been studied, which leverages reuse information to identify useless blocks exhibiting poor locality and considers them as candidates for eviction. The proposed refresh policy prevents these blocks from being refreshed, reducing the overall energy consumption.

To further obtain energy benefits, the refresh policies have been evaluated on an energy-aware (*enaw*) cache architecture, which reduces dynamic energy by applying bank-prediction and swap operations. Compared to a conventional cache with the same storage capacity, the *enaw* cache reduces refresh energy by 72% and dynamic energy by 58%, which translates into a 30% reduction of the

overall on-chip memory hierarchy (leakage and dynamic) consumption. These energy benefits are achieved with minimal impact on performance and area.

Compared to an enegy-aware phased cache, the devised *enaw* cache reduces the refresh energy consumption in half, while improving the performance. Finally, the energy-delay-squared product analysis further supports that the *enaw* cache with selective refresh is the best design option among the studied schemes.

Our evaluation here has focused on L2 caches of single-core processors. For future work we plan to extend the selective refresh design to much largered shared low-level caches and multithreaded workloads in chip multi-processors.

Acknowledgments

This work was supported by the Spanish *Ministerio de Economía y Competitividad* (MINECO) and FEDER funds under Grant TIN2012-38341-C04-01. Additionally, it was also supported by the Intel Early Career Honor Programme Award and the Intel Doctoral Student Honor Programme Award. Prof. David Kaeli was supported in part by an NSF Award, CNS-1319501.

References

[1] J. Liu, B. Jaiyen, R. Veras, O. Mutlu, RAIDR: Retention-Aware Intelligent DRAM Refresh, in: Proceedings of the 39th Annual International Symposium on Computer Architecture, 2012, pp. 1–12. doi:10.1109/ISCA.2012.6237001.

[2] M. Ghosh, H.-H. S. Lee, 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, 2007, pp. 134–145. doi:10.1109/MICRO.2007.38.

[3] T. Ohsawa, K. Kai, K. Murakami, Optimizing the DRAM Refresh Count for Merged DRAM/Logic LSIs, in: Proceedings of the International Symposium on Low Power Electronics and Design, 1998, pp. 82–87.

[4] R. K. Venkatesan, S. Herr, E. Rotenberg, Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM, in: Proceedings of the 12th International Symposium on High-Performance Computer Architecture, 2006, pp. 155–165. doi:10.1109/HPCA.2006.1598122.

[5] J. Kim, M. C. Papaefthymiou, Dynamic Memory Design for Low Data-Retention Power, in: Proceedings of the 10th International Workshop on Integrated Circuit Design, Power and Timing Modeling, Optimization and Simulation, 2000, pp. 207–216. doi:10.1007/3-540-45373-3_22.

[6] R. E. Matick, S. E. Schuster, Logic-based eDRAM: Origins and rationale for use, IBM Journal of Research and Development 49 (1) (2005) 145–165. doi:10.1147/rd.491.0145.

[7] J. Stuecheli, POWER8, Hot Chips.

[8] D. Kanter, Knights Landing Details, Real World Technologies, available online at http://www.realworldtech.com/knights-landing-details/.

[9] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, S.-L. Lu, Reducing Cache Power with Low-Cost, Multi-bit Error-Correcting Codes, in: Proceedings of the 37th Annual International Symposium on Computer Architecture, 2010, pp. 83–93. doi:10.1145/1815961.1815973.

[10] A. Pinto, A Platform-based Approach to Communication Synthesis for Embedded Systems, ProQuest LLC, 2008.

[11] Y. Kurose, M. Okabe, K. Seno, H. Ozawa, T. Wada, K. Taniguchi, H. Hokazono, T. Hirano, I. Kumata, H. Hanaki, K. Hasegawa, S. Horiike, S. Arima, K. Ono, T. Hiroi, S. Takashima, A 90nm Embedded DRAM Single Chip LSI with a 3D Graphics, H.264 Codec Engine, and Reconfigurable Processor, Hot Chips 16.

[12] P. G. Emma, W. R. Reohr, M. Meterelliyoz, Rethinking Refresh: Increasing Availability and Reducing Power in DRAM for Cache Applications, IEEE Micro 28 (6) (2008) 47–56. doi:10.1109/MM.2008.93.

[13] A. Agrawal, A. Ansari, J. Torrellas, Mosaic: Exploiting the Spatial Locality of Process Variation to Reduce Refresh Energy in On-Chip eDRAM Modules, in: Proceedings of the 20th International Symposium on High-Performance Computer Architecture, 2014, pp. 84–95. doi:10.1109/HPCA.2014.6835978.

[14] M.-T. Chang, P. Rosenfeld, S.-L. Lu, B. Jacob, Technology Comparison for Large Last-Level Caches (L3Cs): Low-Leakage SRAM, Low Write-Energy STT-RAM, and Refresh-Optimized eDRAM , in: Proceedings of the 19th International Symposium on High-Performance Computer Architecture, 2013, pp. 143–154. doi:10.1109/HPCA.2013.6522314.

[15] A. Agrawal, P. Jain, A. Ansari, J. Torrellas, Refrint: Intelligent Refresh to Minimize Power in On-Chip Multiprocessor Cache Hierarchies, in: Proceedings of the 19th International Symposium on High-Performance Computer Architecture, 2013, pp. 400–411. doi:10.1109/HPCA.2013.6522336.

[16] J. Albericio, P. Ibáñez, V. Viñals, J. M. Llabería, Exploiting Reuse Locality on Inclusive Shared Last-Level Caches, ACM Transactions on Architecture and Code Optimization 9 (4) (2013) 38:1–38:19. doi:10.1145/2400682.2400697.

[17] A. Jaleel, K. B. Theobald, S. C. Steely Jr., J. Emer, High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP), in: Proceedings of the 37th Annual International Symposium on Computer Architecture, 2010, pp. 60–71. doi:10.1145/1815961.1815971.

[18] A. Valero, J. Sahuquillo, S. Petit, P. López, J. Duato, Combining Recency of Information with Selective Random and a Victim Cache in Last-Level Caches, ACM Transactions on Architecture and Code Optimization 9 (3) (2012) 16:1–16:20. doi:10.1145/2355585.2355589.

[19] C. Zhang, B. Xue, Divide-and-Conquer: A Bubble Replacement for Low Level Caches, in: Proceedings of the 23rd International Conference on Supercomputing, 2009, pp. 80–89. doi:10.1145/1542275.1542291.

[20] X. Liang, R. Canal, G.-Y. Wei, D. Brooks, Process Variation Tolerant 3T1D-Based Cache Architectures, in: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007, pp. 15–26. doi:10.1109/MICRO.2007.40.

[21] S. Kaxiras, Z. Hu, M. Martonosi, Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power, in: Proceedings of the 28th Annual International Symposium on Computer Architecture, 2001, pp. 240–251. doi:10.1109/ISCA.2001.937453.

[22] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, T. Mudge, Drowsy Caches: Simple Techniques for Reducing Leakage Power, in: Proceedings of the 29th Annual International Symposium on Computer Architecture, 2002, pp. 148–157. doi:10.1109/ISCA.2002.1003572.

[23] T. L. Johnson, W.-M. W. Hwu, Run-time Adaptive Cache Hierarchy Management via Reference Analysis, in: Proceedings of the 24th Annual International Symposium on Computer Architecture, 1997, pp. 315–326. doi:10.1145/264107.264213.

[24] J. A. Rivers, E. S. Davidson, Reducing Conflicts in Direct-Mapped Caches with a Temporality-based Design, in: Proceedings of the 25th International Conference on Parallel Processing, 1996, pp. 154–163. doi:10.1109/ICPP.1996.537156.

[25] G. S. Tyson, M. Farrens, J. Matthews, A. R. Pleszkun, A Modified Approach to Data Cache Management, in: Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture, 1995, pp. 93–103. doi:10.1109/MICRO.1995.476816.

[26] B. Calder, D. Grunwald, J. Emer, Predictive Sequential Associative Cache, in: Proceedings of the 2nd International Symposium on High-Performance Computer Architecture, 1996, pp. 244–253.

[27] R. E. Kessler, R. Jooss, A. Lebeck, M. D. Hill, Inexpensive Implementations of Set-Associativity, ACM SIGARCH Computer Architecture News 17 (3) (1989) 131–139. doi:10.1145/74926.74941.

[28] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, K. Roy, Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping, in: Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture, 2001, pp. 54–65. doi:10.1109/MICRO.2001.991105.

[29] S. Petit, J. Sahuquillo, J. M. Such, D. Kaeli, Exploiting Temporal Locality in Drowsy Cache Policies, in: Proceedings of the 2nd Conference on Computing Frontiers, 2005, pp. 371–377. doi:10.1145/1062261.1062321.

[30] Z. Zhu, X. Zhang, Access-Mode Predictions for Low-Power Cache Design, IEEE Micro 22 (2) (2002) 58–71. doi:10.1109/MM.2002.997880.

[31] Various Methods of DRAM Refresh, Micron Technology Inc., Technical Note N-04-30 (1999) 1–4.

[32] B. Keeth, R. J. Baker, B. Johnson, F. Lin, DRAM Circuit Design. Fundamental and High-Speed Topics, John Wiley and Sons, Inc., Hoboken, New Jersey, USA, 2008.

[33] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, N. P. Jouppi, CACTI 5.1, Hewlett-Packard Development Company, Palo Alto, CA, USA. Technical Report.

[34] D. Burger, T. M. Austin, The SimpleScalar Tool Set, Version 2.0, ACM SIGARCH Computer Architecture News 25 (1997) 13–25. doi:10.1145/268806.268810.

[35] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, N. P. Jouppi, A Comprehensive Memory Modeling Tool and its Application to the Design and Analysis of Future Memory Hierarchies, in: Proceedings of the 35th Annual International Symposium on Computer Architecture, 2008, pp. 51–62. doi:10.1109/ISCA.2008.16.

[36] Standard Performance Evaluation Corporation, available online at http://www.spec.org/.