Opportunistic Refreshing Algorithm for eDRAM Memories

Amit Kazimirsky and Shmuel Wimer, *Member, IEEE*

***Abstract*—Embedded DRAM (eDRAM) is an alternative technology that can replace the area and power consumed by SRAM cache memories. eDRAM consumes half the area and an order of magnitude less power than SRAM, but has the drawback of access blockage caused by its periodic data refreshing. This paper presents an opportunistic refreshing algorithm along with the appropriate memory architecture and skim control logic. This architecture takes advantage of the access idleness of the internal partitions of the memory and enables most of the refreshing operations to run concurrently with the ordinary R/W access. This eliminates the refreshing burden almost completely. The algorithm was simulated with industrial DSP access traces, and outperformed in a wide range of eDRAM technologies and internal memory architectures.**

***Index Terms*—eDRAM, L1 cache, low-power cache, refreshing algorithms, refreshing control.**

I. Introduction

EMBEDDED dram (eDRAM) is a well-known memory technology designed to replace the conventional six-transistor (6 T) SRAM by a smaller area and lower power consuming memory bit-cell (cell for short) [1], [2]. eDRAM cells are also less sensitive to process variations than SRAM [3]. There are two eDRAM cell types. The ordinary eDRAM cell, known as 1T1C, comprises one capacitor and one transistor. It is widely used in processors, but requires a special and expensive process technology. It also has the disadvantage of charge destruction at read operation. Hence an immediate and power consuming operation of refreshing for data restoration is required. This read-refresh is needed in addition to ordinary periodic refreshing due to charge leakage over time. The second cell type known as the *gain-cell* (GC), was designed to achieve a smaller area and power advantages of eDRAM while maintaining a low manufacturing cost and avoiding the read-refresh.

There are several types of GC comprising two [4], three [5] and four transistors [6], which have an order of magnitude less leakage-power and occupy less than half of the area of a 6 T-SRAM cell. Unlike 1T1C cells that have single port for both read and write, GCs have two separate ports. This enables the design of memories supporting simultaneous read and write [7].

Manuscript received May 7, 2016; revised June 25, 2016 and July 25, 2016; accepted August 11, 2016. Date of publication September 19, 2016; date of current version October 25, 2016. This work was supported by the Israeli Chief Scientist under the HiPer consortium of the MAGNET program. This paper was recommended by Associate Editor Y. Pu.

The authors are with the Engineering Faculty, Bar-Ilan University, Ramat-Gan 52900, Israel (e-mail: kaziamit@gmail.com; wimers@biu.ac.il).

Digital Object Identifier 10.1109/TCSI.2016.2600538

The main drawback of eDRAMs compared to SRAMs is their refreshing requirement. Unlike the SRAM cell, which retains its state permanently, eDRAM cell requires periodic refresh due to leakage at its storage node. The eDRAM *Data Retention Time* (DRT) can vary from a few to hundreds of microseconds [8], depending on the memory cell structure and the technology in use. During the refreshing operation the eDRAM is blocked for system access, thus causing performance loss.

There are two types of refreshing algorithms. The most commonly used and straightforward is periodic algorithm [9], [10], named *administrative* refreshing in [11]. It is also known as *global refreshing* because it refreshes the entire memory sequentially *row-by-row*. (We use the term row and line interchangeably.) Its main drawback is the blockage to system R/W access during the refreshing period. Methods to reduce its power consumption and system access blockage are referred in [12] and [13].

Global refreshing uses the same ports as ordinary memory access. These are blocked for normal cache accesses when used for refreshing. Due to the separate read and write ports of GC eDRAM, the read and write of distinct rows can take place simultaneously. Since refreshing requires a read and write operation, a sequence of consecutive refreshing accesses can be pipelined. Global refreshing degrades performance because of memory blockage. The cache implementation in [3] reported about 8% performance loss. The authors suggested but did not elaborate on the notion that the L1 under-utilization due to R/W access idleness can be leveraged to hide the refresh operations and reduce performance loss. Another solution to avoid performance degradation is to add extra R/W refreshing dedicated ports, making it independent of the ordinary R/W accesses. This however comes at the cost of considerable L1 area and power growth.

Another type of algorithm works in an *on-the-fly* manner, in which every row is monitored individually for a refreshing alert. There are two alert methods. In the first, each row is supplemented with a replica cell monitored for charge leakage, which indicates when its associated row of cells needs refreshing [14]. The second method monitors each row of cells with a counter that counts the clock cycles elapsed since the most recent write, which is when the counter is reset [3]. One idea is to set these counters individually during silicon testing according to the retention time of the corresponding lines. Once a counter reaches the retention time, its line must either be refreshed or evicted, depending on different refreshing schemes. To this end [3] proposed several refreshing policies combined with cache replacement policies, including no-refreshing and partial-refreshing. In no-refreshing, an attempt to read data which have been expired (corrupted) is treated as a cache miss. These techniques are tightly coupled to the cache replacement policy and maintenance of data consistency across memory hierarchies. While write-through can be handled simply, write-back involves very complex control hardware.

On-the-fly refreshing methods suffer from several drawbacks. With retention time in the range of microseconds and a nanosecond clock cycle, every per-line counter requires ten bits or more, which is considerable overhead. In addition, once the row's monitor alerts, a row refreshing is enforced. While increasing somewhat the memory availability to the system's R/W access, on-the-fly algorithms require complex control logic. Moreover, the system's R/W blockages are unpredictable. They may be introduced sporadically and randomly by line behavior that avoids the system using the memory blockage periods for other purposes.

The viability of eDRAM was examined in-depth for *Large Last-Level caches* (L^3Cs) [15]. The authors introduced the notion of *dead-line prediction* to avoid unnecessary refreshing. A line is dead throughout the period from the latest access and while awaiting eviction. For 32 MB L^3C this time may reach 60% of the lifetime in the cache. Dead-line prediction was first used to reduce the leakage power in SRAM L1 and L2 by reducing the supply voltage [16]. In the eDRAM cache, line refreshing is skipped if it is predicted to be dead, thereby saving energy.

The work in [17] described a concurrent eDRAM refreshing regime that was claimed to lead to low degradation in memory availability. This was achieved by dividing the cache into 16 banks. For a random access, the authors showed 96% to 99% availability in memories of banks comprising 512 to 128 lines, respectively. In this method each bank is supplemented with an independent line counter that generates the address of the currently refreshed line. While a certain bank is addressed for ordinary system access, one or two other banks, defined in a circular manner, are refreshed and increase their corresponding counter. This refresh uses internal R/W ports, with the underlying assumption that banks are individually accessed. By contrast, here we assumes that all banks are accessed simultaneously, thus simplifying the cache. The authors of [17] did not describe what happens when a certain bank is addressed repeatedly for an arbitrarily long period, thus prohibiting refresh for that period. There is probably some other control enforcing the refreshing of this bank when the retention time expires. Since the authors used the entire retention time for the refreshing period, their refreshing as described would not be sufficient to ensure proper data retention, as proven in Section III.

A more recent refreshing algorithm, called *versatile refresh*, was described in [18], and claimed to yield near-optimal throughput. In this case, the algorithm traverses the memory banks in a round-robin order. The refreshing within banks takes place row-by-row, skipping banks accessed for R/W. To resolve the problem of deficit refreshing in blocked banks, the algorithm maintains a global refresh history bitmap of a time-sliding window, which uses a history shift register. This versatile algorithm has an inherent problematic tradeoff. To guarantee high memory availability; i.e., low R/W access blockages, the size of the bitmap must be on the order of the retention period, which requires thousands of bits. A long alternating access-idle memory pattern can impose the toggling of all the bits in the bitmap. This represents huge power consumption, far larger than the power consumed by the entire memory array itself. To reduce this power overhead, a small bitmap of a few entries is used. This, however, defeats the more general aim of a retention period, and results in considerable memory blockages by unnecessary refreshing. Our work avoids this refreshing history bitmap.

Using eDRAM in a GPGPU across its entire memory hierarchy has lately been proposed in [10] and [19] by using the bubble (idle) memory cycles for refreshing. A 4-bit counter per cache entry was suggested, implying considerable over-refreshing and power overhead. For better efficiency the usage of only two counters per bank was considered, for which the determination of the optimal refreshing period is delicate, but this is not addressed. Any opportunistic refreshing must also guarantee that if bubbles do not occur for a long period, data will remain valid. This has a crucial impact on the overall memory availability for R/W, which was ignored. A different type of hiding the refreshing penalty for DRAM was proposed in [20]. It employed the refreshing in parallel to the write operation. The parallel refreshing took place at the bank granularity, too coarse for L1 purpose.

We suggest overcoming the memory availability problem by refreshing in an *opportunistic* manner, without intervening in the memory access. Refreshing takes place concurrently with the R/W operations, by utilizing temporal and local memory idleness. The refreshing architecture takes advantage of computational processes where access to memory addresses varies from cycle to cycle. On the other hand, to overcome the probability that the memory addresses may sometimes be locally stuck for a long period, thus preventing the refresh of this memory portion, initiative refreshing can also be generated. We show that this algorithm demonstrates considerable improvement in the memory availability over conventional periodic refreshing. An analytic model of the availability is presented, supported by simulations with real applications. The main contributions of this work are the following:

- Bridging the performance gap between eDRAM and L1 cache with small area and power overhead.
- A presentation of an eDRAM accurate refreshing model, supported by probabilistic analysis.
- A provably correct formulation of what is the universally optimal opportunistic refreshing period.
- A proposal of effective, robust and simple for hardware implementation, opportunistic refreshing algorithm, which exploits the above optimality conditions.
- An exploration of memory access availability, power and hardware complexity tradeoffs, showing conclusive relations between these factors.
- Good matching of the analytic model with the results obtained from industrial memory access simulations.

The rest of the paper is organized as follows. Section II describe the physical structure of a GC-based eDRAM and related L1 physical architecture. Section III presents the opportunistic refreshing algorithm, which performance is analyzed

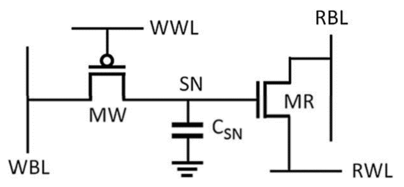


Fig. 1. Two-transistor gain-cell (2 T GC). Write takes place through the Memory Write transistor (MW) and read takes place through the Memory Read transistor (MR). The storage node (SN) is in between.

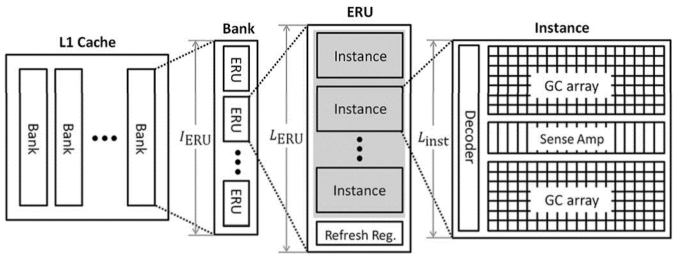


Fig. 2. L1 physical hierarchy.

in Section IV. Section V optimizes the elementary building block of the eDRAM, followed by implementation details in Section VI. Power analysis of a complete memory is discussed in Section VII. Section VIII presents experimental results and Section IX concludes the discussion.

II. The Gain Cell and L1 Cache

We subsequently describe basic gain cell L1 architecture. It is the building block on top of which the opportunistic algorithm is implemented to control.

A. The Gain Cell eDRAM

The GC-eDRAM (GC for short) is a high density alternative to the 6 T-SRAM cell, which is realizable with 2–4 transistors, and is manufactured by standard process technology [7]. Several studies have shown that GC consumes less standby power than a SRAM cell and can be operated at low supply voltages [21]. The GC uses its internal node capacitance to store data. Its unique structure decouples the read and the write ports, thus enabling simultaneous read and write operation of two distinct GCs. Though this work emphasizes the structural and architectural aspects of the memory array independently of the GC type, we recall briefly the basic properties of a 2 T GC below. An example of a GC is shown in Fig. 1 [4].

The write operation takes place through the *Memory Write* transistor (MW). The *Write Bit-Line* (WBL) is set to the value to be stored. To open the MW the *Write Word-Line* (WWL) switches to V_{gnd}, until the write operation is completed; i.e., when the *Storage Node* (SN) consisting of the gate and the diffusion parasitic capacitance C_{SN} is fully charged. The WWL then turns off, sufficiently before WBL gets a new value, since otherwise the data stored in C_{SN} may be corrupted.

The read operation takes place through a *Memory Read* transistor (MR). First, the *Read Bit-Line* (RBL) is pre-charged to V_{dd}, and then is disconnected from the supply voltage. To start the read operation the *Read Word-Line* (RWL) switches to V_{gnd}. If SN is high, MR turns on and RBL is discharged, thus "0" is read. Otherwise, RBL stays high and "1" is read. Notice that the read value is the opposite value of SN. Since the 2 T GC is not connected to a supply voltage, due to current leakage the SN value is destroyed. To sustain its data, it requires refreshing. GC DRT is the time that elapses between writing data into the SN until it cannot be retrieved. This depends not only on the GC structure, but also on system specifications such as its operation frequency and supply voltage [22]. Such considerations are beyond the scope of this work.

Refreshing of the memory array usually takes place row-by-row, and hence requires two independent operations. The first involves reading the data from the SN to an external register, and then writing it back to the SN. The separation of the R/W ports in GC enables simultaneous reading and writing of the data in different rows. Therefore, although the refresh latency of a row is two clock cycles, the refresh throughput is one row per cycle.

B. GC-Based L1 Cache Physical Architecture

This work is focused on the internal physical structure of L1. L1's parameters, such as total size, block size, associativity, and all the other related logical parameters are usually defined by technology, application, architectural and performance considerations. These are beyond the scope of this paper and are orthogonal to the physical aspects discussed below. The eDRAM L1 physical structure is hierarchical, as illustrated in Fig. 2. Our physical architecture below decomposes banks into smaller units, called *Elementary Refreshable Units* (ERUs). Their impact on the entire cache performance will be discussed and analyzed in later sections. ERUs are composed of elementary cache building blocks called *instances*. A typical instance comprises a few hundred rows, whose number L_{inst} is defined by the capacitive load limit the bit-line can drive, and the cache operation frequency. An instance comprises its own peripheral decoders and sense amplifiers.

The ERU comprises I_{inst} instances. An ERU is thus made up of $L_{\text{ERU}} = I_{\text{inst}} \times L_{\text{inst}}$ rows and C_{bank} columns, for a total of $I_{\text{inst}} \times L_{\text{inst}} \times C_{\text{bank}}$ bit-cells. An ERU is a self-contained refreshable entity, possessing all the required control hardware. It contains a *refreshing register*. A refreshing operation first writes the contents of a row into the refreshing register, and then stores it back in that row in the subsequent cycle. The EDA tool dubbed the *memory compiler* [23] assembles and connects ERU macros to logical blocks (banks) and obtains the entire L1.

The primary eDRAM factor defining the *refresh time period*, denoted by T_{DRT}, is its GC DRT, which is the time from writing a datum into a GC until that datum is corrupted (by leakage). An ERU has a single refresh controller, shared by all its I_{inst} instances. Given a memory target clock cycle T_{CLK}, the *retention period* $N_{\text{DRT}} = T_{\text{DRT}}/T_{\text{CLK}}$ dictates the largest possible L_{ERU}. The largest number of I_{inst} is therefore bounded by

$$I_{\text{inst}} \le \left\lfloor \frac{N_{\text{DRT}}}{L_{\text{inst}}} \right\rfloor. \tag{1}$$

The analysis below is two-staged. It first assumes that the ERU size L_{ERU} is given, for which we present the opportunistic refreshing controller. The impact of the L_{ERU} on memory availability for R/W access is then addressed. The tradeoff between two design goals is then presented and involves minimizing the refreshing control HW overhead, thus maximizing L_{ERU}, and maximizing the memory availability, thus minimizing L_{ERU}. These conflicting goals require seeking the optimal L_{ERU}, and since $L_{\mathrm{ERU}} = I_{\mathrm{inst}} \times L_{\mathrm{inst}}$, this also defines I_{inst}.

III. ERU Refreshing Control Algorithm

This section first presents the opportunistic and initiative aspects of the refreshing. It then derives analytically the maximal (hence optimal) refreshing period (interval) between two successive refreshing of a line, such that the validity of the stored data is always guaranteed. Knowing that period, an algorithm and its hardware implementation which exploits the optimality conditions is described.

A. The Behavior of Opportunistic Refreshing

One straightforward refresh algorithm commonly used by dynamic memories [9] is called *ordinary periodic* refreshing. It applies a refresh round in which the L_{ERU} rows are sequentially and contiguously refreshed one-by-one. During the refreshing period the memory is blocked for any R/W access. A successful refresh must ensure that the time elapsed between successive GC refreshing does not exceed the retention period N_{DRT}. An appropriate refreshing controller requires some hardware to monitor the continuous fulfillment of the N_{DRT} requirement. The ordinary periodic refreshing is skim and robust, but system performance may significantly be degraded due to L_{ERU} contiguous cycles when the memory is refreshed; namely the period when the memory is blocked for R/W accesses.

The introduction mentioned two *on-the-fly* refreshing methods satisfying the N_{DRT} requirement: a per-row replica-based and a per-row counter-based. Though they may reduce the amount of row refreshing compared to ordinary periodic refreshing, both methods are expensive. They require considerable hardware, and may also degrade system performance by blocking memory accessibility while a row is being refreshed. Since ordinary periodic refreshing is the most common, it is used as a reference here for comparison with our algorithm.

We suggest a *hybrid* refreshing approach leveraging of the benefits of both; namely, high memory availability at a small hardware cost. The refreshing takes place *opportunistically*, row-by-row in a cyclic, but not necessarily contiguous manner. Unlike on-the-fly methods that unconditionally stop the memory access for row refreshing upon a data retention alert, the opportunistic method utilizes the clock cycles when an ERU is not accessed, which we dub *idle cycles*. Since this is an opportunistic refreshing policy that relies on random occurrences of memory idleness, adequate refresh must be guaranteed when idleness is insufficient. This requires a supplementary mechanism to ensure that no matter what idleness has occurred in an ERU, an N_{DRT} duration will never elapse between two successive refreshes of each ERU row.

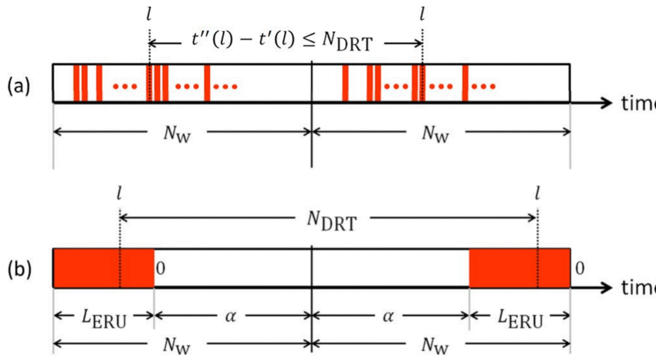


Fig. 3. Derivation of the largest refreshing window.

B. Derivation of the Maximal (Optimal) Refreshing Period

Let us denote by N_w the *time window* (in clock cycles) of an opportunistic refreshing round. Though refreshing utilizes cycles where there is no R/W access to the ERU, N_w must ensure that regardless of the amount of idleness, all the rows have been refreshed properly. Obviously, there must be $N_w > L_{\mathrm{ERU}}$, since all the L_{ERU} rows must be refreshed during N_w. The strict inequality follows since $N_w = L_{\mathrm{ERU}}$ would mean that the ERU is busy only by refreshing rows, thus it is always blocked for any system's R/W access. We therefore define a parameter $\alpha > 0$ such that

$$N_w = L_{\mathrm{ERU}} + \alpha \tag{2}$$

and inquire how large α should be without violating the requirement that at most N_{DRT} clock cycles must elapse between two consecutive refreshes of a row. By the definition of N_w, all the L_{ERU} ERU rows must be refreshed, either due to idleness, or initiatively by blocking the ERU for R/W access, thus degrading memory availability. Obviously the larger the size of α, the better the availability.

To find the maximal value of α, let us examine two successive refreshing rounds as depicted in Fig. 3(a). The time stamps at which the rows are refreshed, either opportunistically due to idleness or initiatively by ERU access stall (blockage), are scattered along N_w, and appear in red. We use the terms ERU stall and blockage interchangeably. Let $t'(l)$ be the time stamp when row $0 \le l \le L_{\mathrm{ERU}} - 1$ has been refreshed in the previous N_w window and let $t''(l)$ be the time stamp in the successive window. A proper refresh requires that $t''(l) - t'(l) \le N_{\mathrm{DRT}}$. The worst refreshing scenario of two successive windows is shown in Fig. 3(b). This occurs when there are L_{ERU} contiguous refreshing cycles at the beginning of the previous window, whereas in the subsequent window there are L_{ERU} contiguous refreshing cycles occurring at its end. Thus, the largest α should satisfy

$$N_{\mathrm{DRT}} = \underbrace{l + \alpha}_{(a)} + \underbrace{\alpha + (L_{\mathrm{ERU}} - l)}_{(b)} = L_{\mathrm{ERU}} + 2\alpha \tag{3}$$

where term a) stems from the previous window and b) stems from the subsequent one. Using (2) and (3), N_w is obtained by

$$N_w = \lfloor L_{\mathrm{ERU}} + \alpha \rfloor = \left\lfloor \frac{N_{\mathrm{DRT}} + L_{\mathrm{ERU}}}{2} \right\rfloor. \tag{4}$$

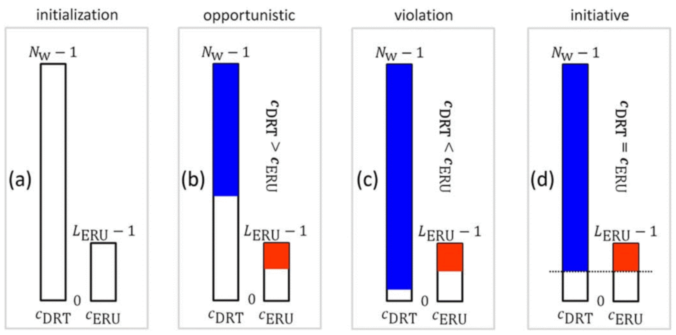


Fig. 4. Refreshing counters. (a) Initialization, (b) potential opportunistic refreshing completion, (c) refreshing violation, (d) initiative refreshing completion.

Note that an addition of just a single cycle to N_w in (4) will violate the proper refreshing if the worst case refreshing shown in Fig. 3(b) takes place. Hence the N_w defined in (4) is maximal indeed.

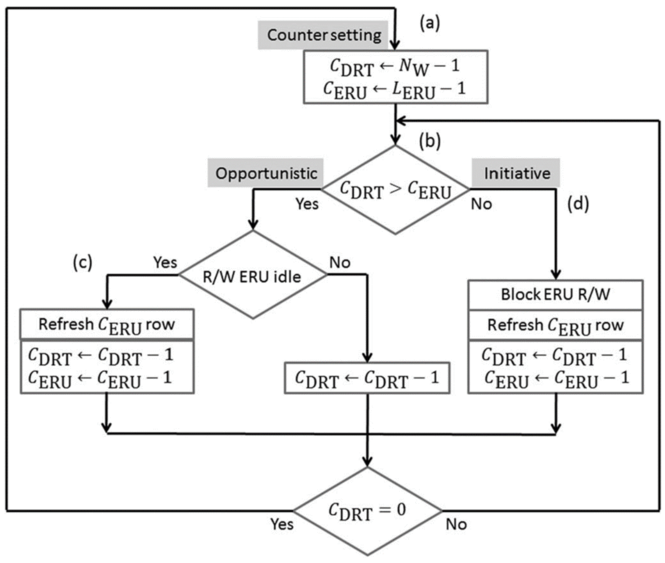

Counter setting
(a)
$c_{DRT} \leftarrow N_W-1$
$c_{ERU} \leftarrow L_{ERU}-1$
(b)
Opportunistic
Initiative
$c_{DRT} > c_{ERU}$
Yes
No
(d)
(c)
R/W ERU idle
Refresh c_{ERU} row
$c_{DRT} \leftarrow c_{DRT}-1$
$c_{ERU} \leftarrow c_{ERU}-1$
$c_{DRT} \leftarrow c_{DRT}-1$
Block ERU R/W
$c_{DRT} = 0$

Fig. 5. Opportunistic refreshing algorithm.

C. Hardware Implementation of Opportunistic Refreshing

The hardware implementation of the proposed refreshing algorithm requires two counters as shown in Fig. 4(a). A counter c_{DRT}, called *DRT counter*, down counts cycle-by-cycle in the range $N_w - 1 \geq c_{\text{DRT}} \geq 0$. Its role is to monitor the time stamp within N_w cycles. Another counter c_{ERU}, called *ERU row counter*, down counts cycle-by-cycle in the range $L_{\text{ERU}} - 1 \geq c_{\text{ERU}} \geq 0$. Its role is to determine which row should be refreshed next. The two counters are synchronized, and set to their initial values simultaneously.

A refreshing operation writes the content stored in the ERU refreshing register to the row at address c_{ERU}. c_{ERU} is then decreased by one, and the contents of the row in the new c_{ERU} address are written into the ERU refreshing register, awaiting the next refreshing to occur. An opportunistic row refreshing requires an idle ERU access state. ERU access by the system's R/W occurs occasionally, in which case refreshing halts. Once an idle state resumes, the refreshing continues at the next row, addressed by c_{ERU}. Thus a memory idleness signal conditions c_{ERU}. Once $c_{\text{ERU}} = 0$, the ERU refreshing round is completed, and both c_{DRT} and c_{ERU} are set to their initial values.

To guarantee proper refreshing, note that c_{DRT} and c_{ERU} compete. The "race" starts when they are initialized to $c_{\text{DRT}} = N_w - 1$ and $c_{\text{ERU}} = L_{\text{ERU}} - 1$, where their finishing line is zero. Initially there is $c_{\text{DRT}} \gg c_{\text{ERU}}$. At each clock cycle c_{DRT} decreases by one, whereas c_{ERU} decreases by one only if memory idleness occurs. As shown in Fig. 4(b), as long as $c_{\text{DRT}} > c_{\text{ERU}}$, $c_{\text{ERU}} = 0$ can take place earlier than $c_{\text{DRT}} = 0$, provided that enough idleness occurs before $c_{\text{DRT}} = 0$ is met.

The case of $c_{\text{ERU}} > c_{\text{DRT}}$ shown in Fig. 4(c) indicates that the retention condition $t''(l) - t'(l) \leq N_{\text{DRT}}$ in two successive N_w time windows will certainly be violated in some of the lower rows, and hence a retention violation will occur. To avoid DRT violations, the discriminating condition $c_{\text{ERU}} = c_{\text{DRT}}$ shown in Fig. 4(d) is used to switch from opportunistic to initiative refreshing, henceforth enforced at each clock cycle until $c_{\text{ERU}} = c_{\text{DRT}} = 0$ is met. The enforcement of initiative refreshing blocks the ERU from any R/W accesses until refreshing is completed, which may cause performance degradation if the system attempts to access it. This degradation is discussed in the next section.

Fig. 5 depicts the opportunistic refreshing algorithm. Every refreshing round takes N_w iterations. It starts in (a) by setting the values of the two counters c_{DRT} and c_{ERU}. The "yes" branch in (b) attempts to refresh opportunistically, provided that idle R/W cycle occurs in (c). In such a case both counters are decremented by one. Otherwise, only c_{DRT} is decremented. Opportunistic refreshing attempts take place as long as the feasibility condition $c_{\text{DRT}} > c_{\text{ERU}}$ holds. Once $c_{\text{DRT}} = c_{\text{ERU}}$, initiative refreshing takes over in (d) and it stays so until refreshing round completes. Notice that at completion after N_w iterations c_{ERU} can be either positive, a case where R/W idleness hid all the refreshing, or c_{ERU} can be zero if some initiative refreshing was enforced due to insufficient idleness.

There is a tricky point when a certain line is read into the refreshing register, awaiting to be re-written in a later cycle. This line may be addressed for ordinary data store between the refreshing read and write cycles, in which case the refreshing will override it with wrong data. Fortunately, writing a new data is nothing but a refresh, so instead of using the refreshing register, one should use the external data.

IV. Performance Analysis and System Implications

As discussed above, the ERU refreshing control may halt its access for proper refreshing, which affects memory availability. This is studied below and compared to the ERU availability of an ordinary periodic refresh. Let the *average ERU availability* η be defined as the fraction of N_w for which the ERU is accessible (available) for a system's R/W. By definition

$0 \le \eta \le 1$, where $\eta = 0$ indicates that it can never be accessed, and $\eta = 1$ indicates it is always available. Recalling that the ERU is blocked during the initiative refresh completion, η is given by the following expression:

$$\eta = 1 - \frac{\#\text{initiative refresh cycles}}{N_w}. \tag{5}$$

In ordinary periodic refreshing where refreshing takes place row-by-row, the ERU is blocked during a contiguous L_{ERU} time period. Let η_{ord} denote its ERU availability. The longest possible refreshing period is N_{DRT}, which from (4) is $2N_w - L_{\mathrm{ERU}}$. Hence

$$\eta_{\mathrm{ord}} = 1 - \frac{L_{\mathrm{ERU}}}{N_{\mathrm{DRT}}} = 1 - \frac{L_{\mathrm{ERU}}}{2N_w - L_{\mathrm{ERU}}}. \tag{6}$$

Considering the opportunistic refreshing algorithm, let $0 \le p_{\mathrm{idle}} \le 1$ be the idleness probability. Let β_{idle} be a random variable counting the number of ERU idle cycles within the N_w window, and let β_{init} be a random variable counting the number of cycles where the ERU is blocked due to initiative refreshing completion. Clearly, when $\beta_{\mathrm{idle}} \ge L_{\mathrm{ERU}}$, initiative refresh completion is not required, hence $\beta_{\mathrm{init}} = 0$. The random variable β_{init} is therefore defined by

$$\beta_{\mathrm{init}} = \max\{L_{\mathrm{ERU}} - \beta_{\mathrm{idle}}, 0\}. \tag{7}$$

It follows that β_{idle} is a binomial random variable, $\beta_{\mathrm{idle}} \sim \mathrm{Bin}(N_w, p_{\mathrm{idle}})$, which in our notation has the following probability density function:

$$\Pr[\beta_{\mathrm{idle}} = k] = \binom{N_w}{k} p_{\mathrm{idle}}^k (1 - p_{\mathrm{idle}})^{N_w - k}. \tag{8}$$

The probability density function of β_{init} is given as follows:

$$\Pr[\beta_{\mathrm{init}} = 0] = \sum_{k=L_{\mathrm{ERU}}}^{N_w} \binom{N_w}{k} p_{\mathrm{idle}}^k (1 - p_{\mathrm{idle}})^{N_w - k} \tag{9}$$

$$\Pr[\beta_{\mathrm{init}} = i] = \binom{N_w}{L_{\mathrm{ERU}} - i} p_{\mathrm{idle}}^{L_{\mathrm{ERU}} - i} \times (1 - p_{\mathrm{idle}})^{N_w - (L_{\mathrm{ERU}} - i)},\ 0 < i \le L_{\mathrm{ERU}} \tag{10}$$

$$\Pr[\beta_{\mathrm{init}} = i] = 0, \quad i > L_{\mathrm{ERU}}. \tag{11}$$

The situation in (11) is not feasible and its probability is therefore zero. Equation (10) addresses the situation where the number of idle cycles β_{idle} within N_w window is smaller than the ERU size L_{ERU}, which necessitates $L_{\mathrm{ERU}} - \beta_{\mathrm{idle}}$ initiative refresh cycles. The situation in (9) corresponds to the situation where there are sufficient idle cycles β_{idle} to accommodate L_{ERU} refreshing opportunistically.

Let η_{opt} denote the memory availability of the opportunistic refreshing algorithm. Knowing the probability of β_{init}, its expected value is $E[\beta_{\mathrm{init}}] \triangleq \sum_{i=0}^{L_{\mathrm{ERU}}} i \Pr[\beta_{\mathrm{init}} = i]$. Substitution of (11) and $j = L_{\mathrm{ERU}} - i$ yields

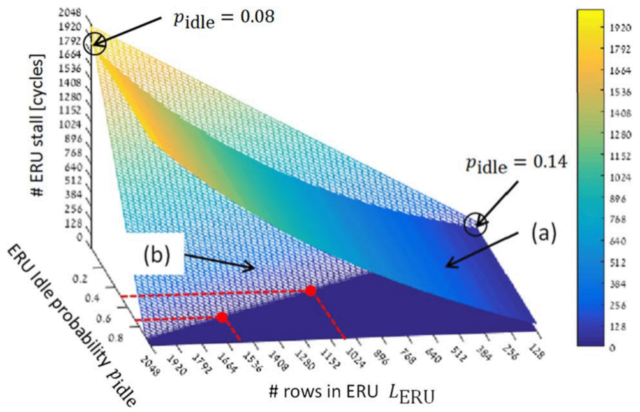


Fig. 6. The dependence of ERU access stalls on L_{ERU} and p_{idle}.

$$\begin{aligned} E[\beta_{\mathrm{init}}] &= \sum_{i=0}^{L_{\mathrm{ERU}}} i \binom{N_w}{L_{\mathrm{ERU}} - i} p_{\mathrm{idle}}^{L_{\mathrm{ERU}} - i} (1 - p_{\mathrm{idle}})^{N_w - (L_{\mathrm{ERU}} - i)} \\ &= \sum_{j=L_{\mathrm{ERU}}}^{0} (L_{\mathrm{ERU}} - j) \binom{N_w}{j} p_{\mathrm{idle}}^j (1 - p_{\mathrm{idle}})^{N_w - j} \\ &\triangleq \sum_{j=0}^{L_{\mathrm{ERU}}} (L_{\mathrm{ERU}} - j) \Pr(\beta_{\mathrm{idle}} = j). \end{aligned} \tag{12}$$

The expression in (12) can be computed numerically, and η_{opt} is obtained by

$$\eta_{\mathrm{opt}} = 1 - \frac{E[\beta_{\mathrm{init}}]}{N_w}. \tag{13}$$

Whereas η is the average ERU availability, $1 - \eta$ is its average blockage to the system's R/W accesses. The average blockage of opportunistic refreshing is compared to that of ordinary refreshing by considering the ratio $(1 - \eta_{\mathrm{ord}})/(1 - \eta_{\mathrm{opt}})$. Substitution of (13) and (6) yields

$$\frac{1 - \eta_{\mathrm{ord}}}{1 - \eta_{\mathrm{opt}}} = \frac{L_{\mathrm{ERU}}/(2N_w - L_{\mathrm{ERU}})}{E[\beta_{\mathrm{init}}]/N_w} = \frac{L_{\mathrm{ERU}}}{E[\beta_{\mathrm{init}}]} \times \frac{N_w}{2N_w - L_{\mathrm{ERU}}}. \tag{14}$$

By the availability definition, $\eta_{\mathrm{opt}} > \eta_{\mathrm{ord}}$ is the situation where opportunistic refreshing is better, whereas for $\eta_{\mathrm{opt}} < \eta_{\mathrm{ord}}$ ordinary refreshing is favored. It is thus worthwhile to determine how parameters L_{ERU} and p_{idle} affect the ERU blockage. To this end we used eDRAM memory implemented in 65 nm technology operated at 500 MHz ($T_{\mathrm{CLK}} = 2.0$ ns) with $T_{\mathrm{DRT}} = 10\ \mu$s, so $N_{\mathrm{DRT}} = T_{\mathrm{DRT}}/T_{\mathrm{CLK}} = 5000$. For each value of L_{ERU} we used the largest (and also the best) N_w, as derived in (4).

Note that for an ordinary periodic refresh the ERU blockage is $L_{\mathrm{ERU}}/N_{\mathrm{DRT}} = L_{\mathrm{ERU}}/(2N_w - L_{\mathrm{ERU}})$, which is independent of p_{idle}, as shown by surface (a) in Fig. 6. Surface (b) illustrates (12)'s dependence on L_{ERU} and p_{idle}. The opportunistic refresh is preferred everywhere surface (b) is below surface (a). Fig. 6 shows that for ERU size $L_{\mathrm{ERU}} = 128$, the opportunistic refresh is better for $p_{\mathrm{idle}} > 0.14$, whereas for $L_{\mathrm{ERU}} = 2048$

it is better for $p_{\text{idle}} > 0.08$. Recalling that L1 may comprise several ERUs, most programs have a far greater ERU idleness probability. Even for the case of zero L1 idleness, individual ERUs will have some positive idleness, since the system's R/W memory access implies access to only a single ERU at a time, and access addresses change over time from one ERU to others. This point is elaborated on in the next sections.

V. ERU Optimization

Section IV discussed the average number of ERU blockages per N_w time window. What counts from a system viewpoint is the availability of the entire L1, which should be maximized. Assuming that all L1 banks are accessed simultaneously, the average L1 availability is the outcome of the number of ERUs within a bank and their average availabilities. While L1 idleness is determined solely by the running program, the ERU availability is under the designer's control. The internal banks and ERU structure are up to the designer who usually aims at simplifying the refreshing control and minimizing its power consumption.

As shown in Fig. 2, an ERU is a self-contained refreshed entity, comprising a refreshing register, an ERU row counter c_{ERU}, and a DRT counter c_{DRT}. Since c_{DRT} counts a global absolute time, it is shared by all L1 units. The ERU net hardware overhead is therefore the refreshing register and the c_{ERU} counter. A natural way to reduce this overhead is to increase L_{ERU} by integrating more instances. This however raises a problem. The ERU enlargement increases its R/W access probability, hence reducing p_{idle}, which in turn reduces ERU availability η_{opt}. The problem is therefore how to determine the ERU size to obtain high availability at a reasonable hardware cost.

Figs. 4 and 5 illustrate that when N_w starts, opportunistic refreshing takes place. It may later turn into the initiative mode if the remaining time c_{DRT} reaches c_{ERU}, as shown in Fig. 4(d). Whereas opportunistic refreshing comes for free, initiative refreshing blocks the ERU for R/W access. The function $p_{\text{idle}}(L_{\text{ERU}})$ is therefore of interest. Though the ERU is practically quantized to instance [see (1) and Fig. 2], below we examine the continuous $p_{\text{idle}}(L_{\text{ERU}})$ function, which can be quantized once the memory is physically assembled.

Let L_{bank} denote the number of rows in a bank. Since the L1 architecture is such that all its banks are accessed simultaneously L_{bank} is also the number of rows in L1. Fig. 2 shows that an L1 bank comprises several ERUs. Let I_{ERU} be their number. Let q_{idle} be the L1 idleness probability (and hence of a bank as well), determined by the running program. An ERU can be refreshed either if L1 (and hence its banks) is not accessed by the program for R/W, or if L1 was accessed but the addressed row was not located at that ERU. Assuming all ERUs have same access probability, there is

$$p_{\text{idle}} = q_{\text{idle}} + (1 - q_{\text{idle}})\left(1 - \frac{1}{I_{\text{ERU}}}\right). \tag{15}$$

A bank has L_{bank} rows and it comprises I_{ERU} ERUs, hence

$$L_{\text{ERU}} = \frac{L_{\text{bank}}}{I_{\text{ERU}}}. \tag{16}$$

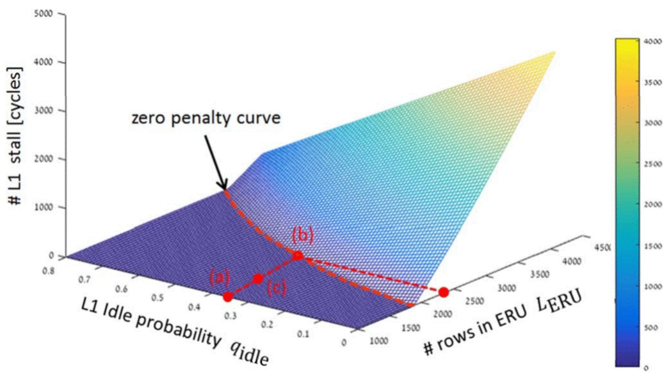


Fig. 7. The dependence of L1 access stalls (blockages) on L_{ERU} and q_{idle}.

Substitution of (16) in (15) yields

$$p_{\text{idle}} = q_{\text{idle}} + (1 - q_{\text{idle}})\frac{L_{\text{bank}} - L_{\text{ERU}}}{L_{\text{bank}}}. \tag{17}$$

Consider for example a 64 KB L1. Each row (cache block) contains four words (16 bytes) which yield 4 K rows. A logical column is folded physically into several banks to obtain a nearly square L1 footprint. Similar to Fig. 6, which illustrates the average number of ERU blockages per N_w time window, Fig. 7 illustrates the average blockages for the entire L1. This blockage penalty is derived by substituting the p_{idle} of (17) into (12), expressed in terms of the running program's q_{idle}. Note the optimal *zero penalty* curve in Fig. 7. It implies for instance that for a given $q_{\text{idle}} = 0.35$ shown in point (a), the maximal number of ERU rows such that no stalls (blockages) are required for valid refreshing is $L_{\text{ERU}} = 2300$, as shown in point (b). An attempt to decrease the ERU size; e.g., point (c), may increase the number I_{ERU} of ERUs, thus requiring more refreshing hardware overhead, which is useless since point (b) is optimal.

VI. Implementation Details and Tradeoffs

Let I_{inst} be the number of instances within an ERU (see Fig. 2), and L_{inst} be the number of rows of an instance. Fig. 8 illustrates a hardware implementation of the opportunistic refreshing algorithm for $I_{\text{inst}} = 2$, with some of the control signals and minor details ignored. The main hardware overheads are grayed.

The ERU's top MUX selects the data to be stored, either due to an ordinary write cycle where the data are external, or due to a refreshing cycle, where the data are obtained from the ERU refreshing register. The left MUX selects the row address to be either the ordinary R/W cycle, or the address of the presently refreshed row, generated by c_{ERU}. Recall that in a refreshing cycle two rows are addressed, one is written and the successive row is read into the refreshing register awaiting the next refreshing cycle. The two MUXs are controlled by the same select signal (not shown), generated by refreshing control logic. Note that the retention counter c_{DRT} is common to all L1's ERUs. Since an L1 access takes place for all the banks simultaneously, c_{ERU} and the refreshing control logic can be shared by all the banks.

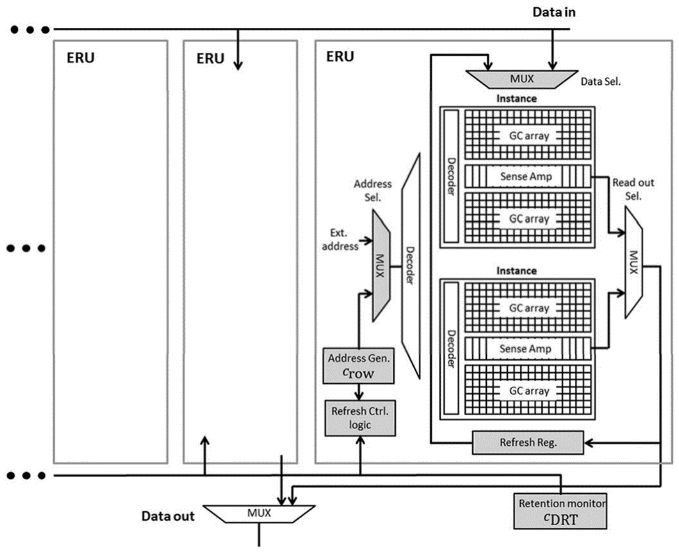


Fig. 8. Refreshing hardware implementation.

Section V assumed that L_{ERU} can vary continuously, but in reality L_{ERU} is quantized to a multiplication of instance size L_{inst} and (16) is quantized accordingly. Consider for example an L1 idleness of $q_{\text{idle}} = 0.35$, shown by point (a) in Fig. 7. The line extending from $q_{\text{idle}} = 0.35$ intersects the zero penalty curve at point (b) where $L_{\text{ERU}} = 2300$. For $L_{\text{inst}} = 256$ this implies $I_{\text{inst}} = \lfloor L_{\text{ERU}}/L_{\text{inst}} \rfloor = 8$. The nearest power of two from below is 8, yielding $L_{\text{ERU}} = I_{\text{inst}} \times L_{\text{inst}} = 2048$. For $q_{\text{idle}} = 0.35$ this is the optimal design since it yields the largest possible ERU for zero L1 R/W access blockages. An attempt to decrease I_{inst}, e.g. to 4, cannot improve the number of stalls which anyway is zero, but rather will double the number I_{ERU} of ERUs within a bank and the refreshing hardware overhead accordingly. On the other hand, increasing I_{inst} to 16 will cause nonzero blockage cycles.

The above analysis found the largest ERU for which opportunistic refreshing does not invoke initiative refreshing, and hence not pay L1 blockages. It is important to note that this result holds under the assumption that the accesses to L1 result in uniformly distributed random addressing of ERUs. Even if L1 is continuously accessed for a long period, such as when $q_{\text{idle}} = 0$, equation (17) tells us that under random uniform addressing, the idleness of an ERU is positive. Substitution of $q_{\text{idle}} = 0$ in (17) yields $p_{\text{idle}} = (L_{\text{bank}} - L_{\text{ERU}})/L_{\text{bank}}$. For $q_{\text{idle}} = 0$ an L1 comprising two ERUs there is $p_{\text{idle}} = 0.5$, and for four ERUs we get $p_{\text{idle}} = 0.75$. The two red spots positioned on surface (b) in Fig. 6 show that for such cases, any $L_{\text{ERU}} \leq 1088$ for $p_{\text{idle}} = 0.5$ and any $L_{\text{ERU}} \leq 1600$ for $p_{\text{idle}} = 0.75$ ensures that the opportunistic refreshing does not invoke any initiative refreshing cycles, and thus zero blockages are guaranteed.

Zero blockages were obtained under the assumption of access to random ERUs. Real applications may encounter worst-scenario cases. A possible worst case may occur in the L1 *Instruction-cache* (I-cache) where an instruction is fetched at each clock cycle. In case of a program loop, the I-cache reads may be stuck at a certain ERU for a long period. A loop consuming N_w instructions or more will avoid any opportunistic ERU refreshing cycles. Such situations are less frequent in the L1 *Data-cache* (D-cache). These extreme cases prohibit opportunistic refreshing and only initiative ones will take place, thus causing some degradation compared to ordinary periodic refreshing.

To assess this degradation, recall that ordinary refreshing always blocks L1 access for contiguous L_{ERU} cycles during the N_{DRT} period, whereas opportunistic refreshing blocks L1 access in the worst case for L_{ERU} cycles during the $N_w = (1/2)(N_{\text{DRT}} + L_{\text{ERU}})$ period. The worst case access degradation is the ratio between the corresponding refreshing periods $N_{\text{DRT}}/N_w = 2N_{\text{DRT}}/(N_{\text{DRT}} + L_{\text{ERU}})$. For the example in Fig. 7 where $N_{\text{DRT}} = 5000$ and $L_{\text{ERU}} = 2048$, it follows that the worst case access degradation is 1.42.

VII. Power Analysis

This section analyzes and compares the power efficiency of eDRAM employing the ordinary refreshing algorithm and the eDRAM employing the opportunistic refreshing algorithm for 2 T and 3 T GCs. Let us first consider the average power consumed by a single bit. There is the read, write and leakage power, and also a refresh component P_{refresh}. Since ordinary periodic refreshing has a period of T_{DRT}, the per-bit average refreshing power is

$$P_{\text{refresh}}^{\text{ord}} = \frac{E_{\text{read}} + E_{\text{write}}}{T_{\text{DRT}}} \tag{18}$$

where E_{read} and E_{write} are the read and write energies, respectively, measurable by SPECTRE (SPICE) simulations. The period of the opportunistic refreshing is $N_w \times T_{\text{CLK}}$, yielding a per-bit average refreshing power

$$P_{\text{refresh}}^{\text{opt}} = \frac{E_{\text{read}} + E_{\text{write}}}{N_w \times T_{\text{CLK}}} \tag{19}$$

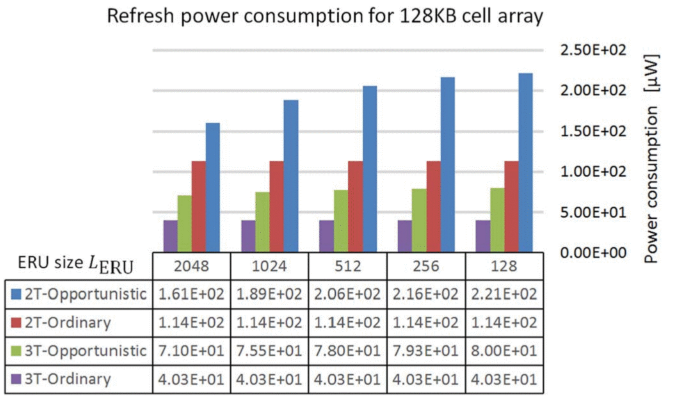
It is important to note that by definition, when the opportunistic algorithm and the ordinary algorithm are employed on the same memory, with the same internal division into ERUs, opportunistic refreshing is less power efficient than ordinary refreshing. This follows from the fact that both must refresh all the rows, so the same refreshing energy is consumed, but the former amortizes it over N_w period, which is shorter than the period N_{DRT} used by the latter [see (3) and (4)]. On the other hand, the system availability of the opportunistic refreshing is larger. A fair measure is therefore the power consumption per availability unit, which is shown in the next section, where real application memory traces are simulated.

Table I shows the respective parameters of 2 T and 3 T memory bit-cells obtained by SPICE post-layout simulations in 65 nm technology. 6 T SRAM cell is presented for reference, showing that eDRAM read, write and leakage power is one to two order of magnitude smaller than SRAM. The results are similar to those described in [2], [5], [24].

The following comparison accounts for the bit-cell array alone, and excludes the peripheral circuits such as decoders and sense-amplifiers. This is justified by the fact that SRAM's sense-amplifiers involve a differential amplifier [25], whereas

TABLE I
POWER AND ENERGY CONSUMPTION OF VARIOUS MEMORY BIT-CELLS

Cell [measured per-bit]	6T SRAM	eDRAM Gain Cell	
		2T	3T
Area / 6T SRAM	1	0.4	0.5
Supply Voltage [Volt]	0.9	0.9	0.9
Retention Time [10^{-6} Sec]	∞	10	30
Write Energy [10^{-15} Joule]	1.04	0.103	0.124
Read Energy [10^{-15} Joule]	5.86	0.98	1.03
Leakage Power [10^{-9} Watt]	3.38	0.0113	0.0126
Refresh Power (periodic) [10^{-9} Watt]	0	0.108	0.0385
Refresh Power (opportunistic) [10^{-9} Watt]	0	0.189	0.0755



ERU size L_{ERU}	2048	1024	512	256	128
🟦 2T-Opportunistic	1.61E+02	1.89E+02	2.06E+02	2.16E+02	2.21E+02
🟥 2T-Ordinary	1.14E+02	1.14E+02	1.14E+02	1.14E+02	1.14E+02
🟩 3T-Opportunistic	7.10E+01	7.55E+01	7.80E+01	7.93E+01	8.00E+01
🟪 3T-Ordinary	4.03E+01	4.03E+01	4.03E+01	4.03E+01	4.03E+01

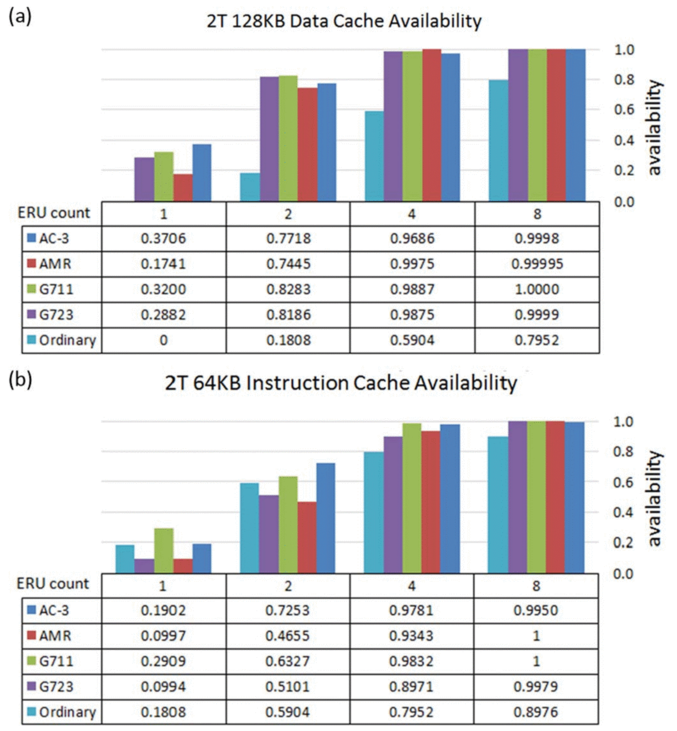
Fig. 9. Refreshing power of 128 KB eDRAM arrays of 2 T and 3 T bit-cells, employing opportunistic and ordinary periodic algorithms.

those in the eDRAM are single-ended [21]. The latter are far more power efficient. Hence accounting the peripheral will always improve the relative power consumption in favor of eDRAM. In any case as regards the comparison of the power consumed by the eDRAM refreshing algorithms, they use the same peripheral circuits.

Fig. 9 shows the refreshing power consumption of the ordinary and opportunistic algorithms employed for 2 T and 3 T eDRAM 128 KB arrays as function of the ERU size. It was explained earlier that the opportunistic algorithm consumes more power than ordinary refreshing. As shown in Table I, this extra power is still negligible compared to the 6 T-SRAM.

VIII. EXPERIMENTAL RESULTS

The performance of the opportunistic refreshing algorithm was tested with real applications obtained from the Ceva Corporation, run on its TeakLite DSP and XM4 vision processor. Ceva's simulator generates memory access traces for the instruction memory and the data memory. An entry of a trace includes the time stamp (clock cycle), a memory address and whether a read or write operation (for instruction- only reads) is required. The traces were fed into a 64 KB instruction eDRAM and 128 KB data eDRAM models, with 16 Bytes per row (line), thus yielding $L_{\text{bank}} = 4$ K and $L_{\text{bank}} = 8$ K memory rows, respectively. A memory access addressed the entire line (all the banks). Two GC topologies of 2 T and 3 T, with respective $T_{\text{DRT}} = 10\ \mu$s and $T_{\text{DRT}} = 30\ \mu$s retention



ERU count	1	2	4	8
🟦 AC-3	0.3706	0.7718	0.9686	0.9998
🟥 AMR	0.1741	0.7445	0.9975	0.99995
🟩 G711	0.3200	0.8283	0.9887	1.0000
🟪 G723	0.2882	0.8186	0.9875	0.9999
■ Ordinary	0	0.1808	0.5904	0.7952

ERU count	1	2	4	8
🟦 AC-3	0.1902	0.7253	0.9781	0.9950
🟥 AMR	0.0997	0.4655	0.9343	1
🟩 G711	0.2909	0.6327	0.9832	1
🟪 G723	0.0994	0.5101	0.8971	0.9979
■ Ordinary	0.1808	0.5904	0.7952	0.8976

Fig. 10. 2 T GC memory access availability of opportunistic refreshing compared to ordinary periodic refreshing, (a) 128 KB data cache and (b) 64 KB instruction cache.

time were tested. The clock speed in all the experiments was 500 MHz, namely $T_{\text{CLK}} = 2.0$ n s. The impact of the memory division into $I_{\text{ERU}} = 1, 2, 4, 8$ ERUs (see Fig. 2) on memory availability was studied by simulations. N_{DRT} was obtained from $N_{\text{DRT}} = T_{\text{DRT}}/T_{\text{CLK}}$, whereas L_{ERU} was determined by $L_{\text{ERU}} = L_{\text{bank}}/I_{\text{ERU}}$.

To simulate the cache performance, the traces were divided into time windows of N_w clock cycles, as defined in (4). After the mapping of a memory address into an ERU index, the access to every ERU was recorded during N_w. Given 2^k ERUs, let N_{i_access} count the number of accesses to ERU i, $0 \le i \le 2^k$, occurring during N_w cycles. Recall the role of the two counters c_{DRT} and c_{ERU} shown in Fig. 4. If $c_{\text{DRT}} = c_{\text{ERU}}$, an initiative refreshing completion will take place before N_w ends [see Fig. 4(d)]. This condition is maintained independently and in parallel for all ERUs, and accounts for the memory blockage.

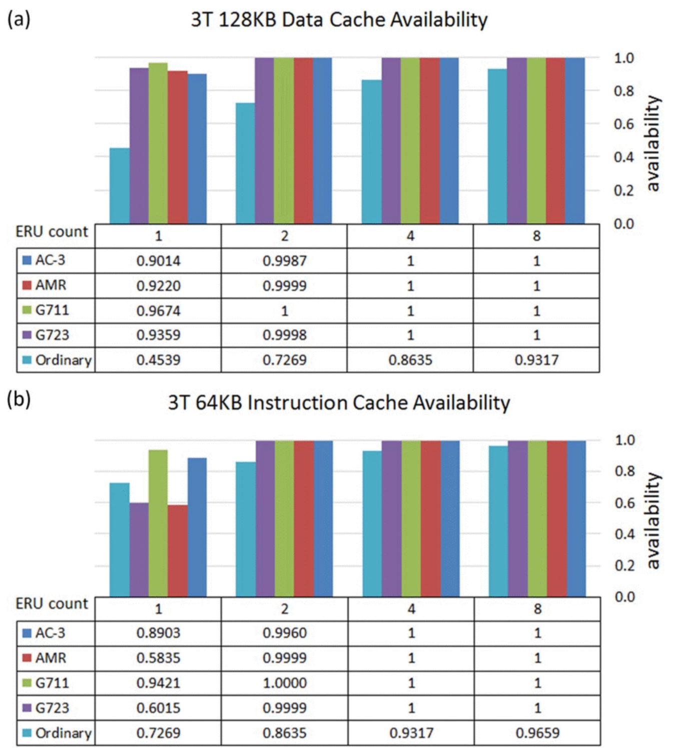
The expression

$$N_{\text{access}} = \max_{0 \le i \le 2^k} N_{i\_\text{access}} \tag{20}$$

dictates the worst case memory access occurring during N_w cycles. If $L_{\text{ERU}} - (N_w - N_{\text{access}}) \le 0$, no initiative refreshing completion is required. Therefore the amount of blockage per N_w is

$$\max\{L_{\text{ERU}} - (N_w - N_{\text{access}}), 0\}. \tag{21}$$

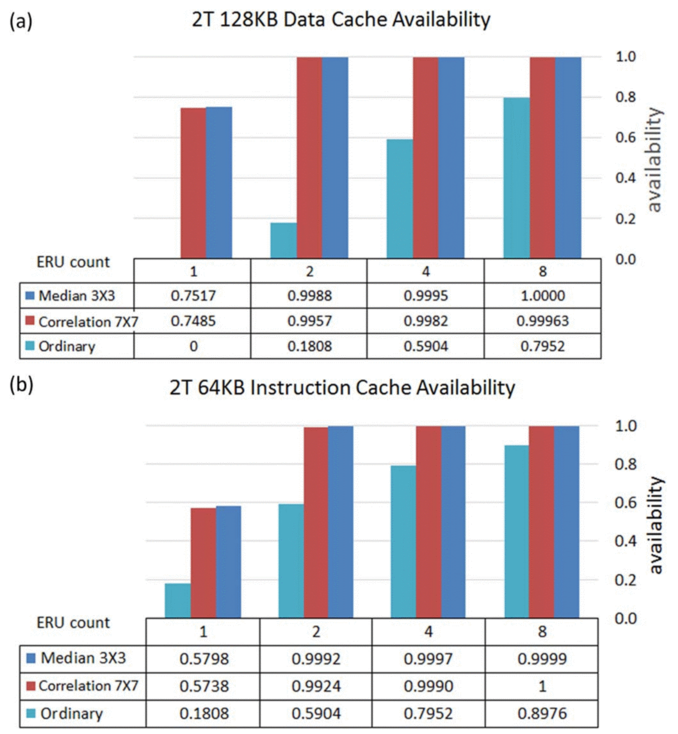
For each setting of the parameters (2 T and 3 T GC, 64 KB and 128 KB memory) and each memory access trace, (21) was averaged over all N_w to obtain the average availability. The results for 2 T GC are shown in Fig. 10 and for 3 T GC in Fig. 11, where (a) is for the data cache and (b) is for the



ERU count	1	2	4	8
🟦 AC-3	0.9014	0.9987	1	1
🟥 AMR	0.9220	0.9999	1	1
🟩 G711	0.9674	1	1	1
🟪 G723	0.9359	0.9998	1	1
🟦 Ordinary	0.4539	0.7269	0.8635	0.9317

ERU count	1	2	4	8
🟦 AC-3	0.8903	0.9960	1	1
🟥 AMR	0.5835	0.9999	1	1
🟩 G711	0.9421	1.0000	1	1
🟪 G723	0.6015	0.9999	1	1
🟦 Ordinary	0.7269	0.8635	0.9317	0.9659

Fig. 11. 3 T GC memory access availability of opportunistic refreshing compared to ordinary periodic refreshing, (a) 128 KB data cache and (b) 64 KB instruction cache.



ERU count	1	2	4	8
🟦 Median 3X3	0.7517	0.9988	0.9995	1.0000
🟥 Correlation 7X7	0.7485	0.9957	0.9982	0.99963
🟦 Ordinary	0	0.1808	0.5904	0.7952

ERU count	1	2	4	8
🟦 Median 3X3	0.5798	0.9992	0.9997	0.9999
🟥 Correlation 7X7	0.5738	0.9924	0.9990	1
🟦 Ordinary	0.1808	0.5904	0.7952	0.8976

Fig. 12. 2 T GC memory access availability comparison of video application, (a) 128 KB data cache and (b) 64 KB instruction cache.

instruction cache. Each experiment contained four memory access simulations of the following TeakLite DSP algorithms:

1. Acoustic Coder 3 (AC3), which is a high quality audio coding format developed by Dolby Labs.
2. Adaptive Multi-Rate (AMR) voice coding, used for mobile communication.
3. G.711, which is a standard for audio companding.
4. G.723.1, which is an audio codec for voice that compresses voice audio in 30 ms frames.

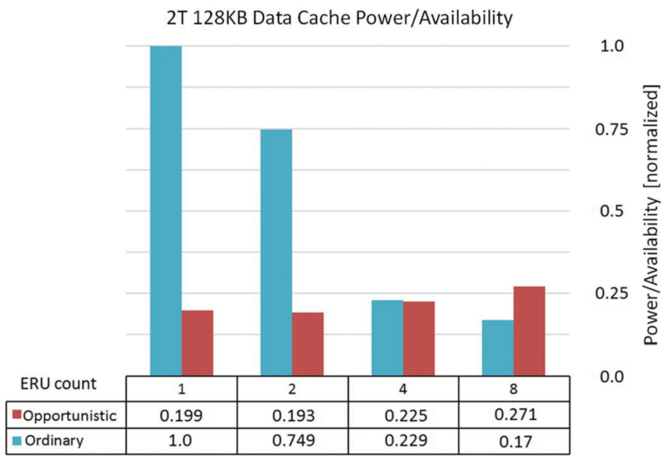
For each of the cache divisions into ERUs and all the above traces, the opportunistic refreshing algorithm was employed. The fifth entry of the tables was used as a reference, and dubbed "ordinary," to represent the ordinary periodic refreshing employed for each ERU in parallel. It is important to note that ordinary refreshing is independent of the memory access patterns, and takes place deterministically, by refreshing the entire rows of an ERU within every N_{DRT} period.

Figs. 10(a) shows that for a 128 KB data cache comprising a single ERU, ordinary periodic refreshing is not feasible. This follows from the fact that for 2 T GC there is $N_{\mathrm{DRT}} = 10\ \mu\mathrm{s}/500\ \mathrm{MHz} = 5000$, whereas the cache comprises 8 K lines; hence ordinary refreshing cannot be employed. Though the opportunistic refreshing presents low availability for a single ERU, it is still feasible for the given benchmark. As expected, as the number of ERUs increased, so did the memory availability. For opportunistic refreshing it approached 100% at four ERUs, whereas the ordinary refreshing lagged behind.

Fig. 10(b) presents the findings for the 64 KB instruction cache. It shows a similar trend as in data memory. It is important to note that the memory availability of ordinary refreshing depends solely on the ERU size (and not on the traces). Since the size of the instruction cache ERU is half the size of the data cache, the fifth row of the table in Fig. 10(b) was shifted one position left with respect to Fig. 10(a). By contrast, the memory availability of opportunistic refreshing depends on memory access traces. Instruction traces are by nature more intense (occurring almost every cycle) than data cache traces. Therefore, while for data memory traces the opportunistic memory availability is superior to the ordinary algorithm for all ERU divisions, for instruction memory traces opportunistic refreshing is by far superior when it occurs for divisions into four and eight ERUs. It is clear from Fig. 10 that for the same memory availability, ordinary refreshing requires more ERUs than opportunistic refreshing, implying higher hardware overhead, since each ERU maintains its own refreshing register and control logic.

Similar behavior for the 3 T GC is shown in Fig. 11. The memory availability approaches 100% with fewer ERUs compared to 2 T. This stems from the much longer DRT of 3 T GC (30 μs) compared to 2 T (10 μs). Obviously, the superiority of opportunistic refreshing over ordinary refreshing is more apparent in shorter DRT, which is the trend reflected in modern technologies.

The superiority of the opportunistic refreshing over ordinary refreshing is further demonstrated for extensive memory access. Fig. 12 show the memory availability obtained by simulations of two XM4 vision processor algorithms, named Median 3×3 and Correlation 7×7, which are image convolution filters. Ordinary refreshing availability is lagging far behind.



ERU count	1	2	4	8
🟥Opportunistic	0.199	0.193	0.225	0.271
🟦Ordinary	1.0	0.749	0.229	0.17

Fig. 13. 2 T GC memory refreshing power/availability efficiency for 128 KB data cache.

Though (18) and (19) show that the refreshing power of the opportunistic is higher than the ordinary, as shown in Fig. 9, the picture changes when availability is also accounted. A fair efficiency measure is the refreshing power consumption per availability unit. To this we divided the refreshing power shown in Fig. 9 by the average availability of various video applications. Fig. 13 shows the results. For small number of ERUs in a bank (large size of ERUs) the efficiency of the opportunistic refreshing wins by far. As ERU count increases, hence implying more refreshing hardware overhead, the efficiency of the ordinary refreshing is getting better. This trend does not change for I-cache and 3 T GCs.

IX. Conclusion

To overcome the memory access blockage incurred by the need for eDRAM periodic refreshing, this paper proposed an optimal opportunistic refreshing algorithm. The algorithm takes advantage of memory access idleness that may occur during the run-time of programs. The algorithm achieves availability approaching SRAM, with considerably less energy consumption. Opportunistic refreshing was shown to be by far superior to ordinary periodic refreshing commonly used for eDRAMs. The above advantages were demonstrated for various eDRAM cells, various memory sizes, and both instruction and data caches across a variety of real applications from industrial DSPs.

Acknowledgment

The authors are thankful for the useful and helpful comments made by the anonymous reviewers. They are also thankful to Ceva Corporation for providing memory traces simulation data.

References

[1] B. John, W. R. Reohr, P. Parries, G. Fredeman, J. Golz, S. E. Schuster, R. E. Matick, H. Hunter, C. C. Tanner, J. Harig, H. Kim, and S. S. Iyer, "A 500 MHz random cycle, 1.5 ns latency, SOI embedded DRAM macro featuring a three-transistor micro sense amplifier," *IEEE J. Solid-State Circuits*, vol. 43, no. 1, pp. 86–95, 2008.

[2] K. C. Chun, P. Jain, T.-H. Kim, and C. H. Kim, "A 667 MHz logic-compatible embedded DRAM featuring an asymmetric 2 T gain cell for high speed on-die caches," *IEEE J. Solid-State Circuits*, vol. 47, no. 2, pp. 547–559, 2012.

[3] L. Xiaoyao, R. Canal, G.-Y. Wei, and D. Brooks, "Process variation tolerant 3T1D-based cache architectures," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2007, pp. 15–26.

[4] S. Dinesh, Y. Ye, P. Aseron, S.-L. Lu, M. M. Khellah, J. Howard, G. Ruhl, K. T, B. S, V. De, and A. Keshavarzi, "2 GHz 2 Mb 2 T gain cell memory macro with 128 GBytes/sec bandwidth in a 65 nm logic process technology," *IEEE J. Solid-State Circuits*, vol. 44, no. 1, pp. 174–185, 2009.

[5] K. C. Chun, P. Jain, J. H. Lee, and C. H. Kim, "A 3 T gain cell embedded DRAM utilizing preferential boosting for high density and low power on-die caches," *IEEE J. Solid-State Circuits*, vol. 46, no. 6, pp. 1495–1505, 2011.

[6] R. Giterman, A. Teman, P. Meinerzhagen, A. Burg, and A. Fish, "4 T gain-cell with internal-feedback for ultra-low retention power at scaled CMOS nodes," in *Proc. IEEE ISCAS*, Melbourne, Australia, 2014.

[7] A. Teman, P. Meinerzhagen, A. Burg, and A. Fish, "Review and classification of gain cell eDRAM implementations," in *2012 IEEE 27th Conv. IEEEI*.

[8] P. Meinerzhagen, A. O. Andiç, J. Treichler, and A. P. Burg, "Design and failure analysis of logic-compatible multilevel gain-cell-based DRAM for fault-tolerant VLSI systems," in *Proc. 21st Ed. Great Lakes Symp. VLSI*, 2011.

[9] J. M. Rabaey, A. P. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits*, Ch. 12, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, 2002.

[10] N. Jing, L. Jiang, T. Zhang, C. Li, F. Fan, and X. Liang, "Energy-efficient eDRAM-based on-chip storage architecture for GPGPUs," *IEEE Trans. Comput.*, vol. 65, no. 1, pp. 122–135, 2016.

[11] R. W. Reohr, "Memories: Exploiting them and developing them," in *Proc. IEEE Int. SOC Conf.*, 2006.

[12] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-aware intelligent DRAM refresh," *ACM SIGARCH Comput. Archit. News*, vol. 40, no. 3, pp. 1–12, 2012.

[13] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM refresh-power through critical data partitioning," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 213–224, 2012.

[14] A. Teman, P. Meinerzhagen, R. Giterman, A. Fish, and A. Burg, "Replica technique for adaptive refresh timing of gain-cell-embedded DRAM," *IEEE Trans. Circuits Syst. II, Express Briefs*, vol. 61, no. 4, pp. 259–263, 2014.

[15] M.-T. Chang, P. Rosenfeld, S.-L. Lu, and J. Biji, "Technology comparison for large last-level caches (L 3 Cs): Low-leakage SRAM, low write-energy STT-RAM, and refresh-optimized eDRAM," in *Proc. IEEE 19th Int. Symp. HPCA2013*.

[16] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," *ACM SIGARCH Comput. Archit. News*, vol. 29, no. 2, pp. 240–251, 2001.

[17] K. Toshiaki, P. Parries, D. R. Hanson, H. Kim, J. Golz, G. Fredeman, R. Rajeevakumar, J. Griesemer, N. Robson, A. Cestero, B. A. Khan, G. Wang, M. Wordeman, and S. S. Iyer, "An 800-MHz embedded DRAM with a concurrent refresh mode," *IEEE J. Solid-State Circuits*, vol. 40, no. 6, pp. 1377–1387, 2005.

[18] M. Alizadeh, A. Javanmard, S.-T. Chuang, S. Iyer, and Y. Lu, "Versatile refresh: Low complexity refresh scheduling for high-throughput multi-banked eDRAM," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 1, pp. 247–258, 2012.

[19] N. Jing, Y. Shen, Y. Lu, S. Ganapathy, Z. Mao, M. Guo, R. Canal, and X. Liang, "An energy-efficient and scalable eDRAM-based register file architecture for GPGPU," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, 2013.

[20] K.-W. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving DRAM performance by parallelizing refreshes with accesses," in *Proc. IEEE 20th Int. Symp. HPCA*, 2014.

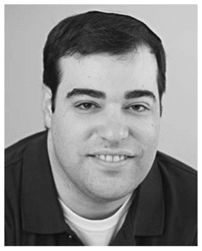
[21] R. Giterman, A. Teman, P. Meinerzhagen, L. Atias, A. Burg, and A. Fish, "Single-supply 3 T gain-cell for low-voltage low-power applications," *IEEE Trans. VLSI Syst.*, vol. 24, no. 1, pp. 358–362, 2015.

[22] P. Meinerzhagen, A. Teman, R. Giterman, A. Burg, and A. Fish, "Exploration of sub-VT and near-VT 2 T gain-cell memories for ultra-low power applications under technology scaling," *J. Low Power Electron. Appl.*, vol. 3, no. 2, pp. 54–72, 2013.

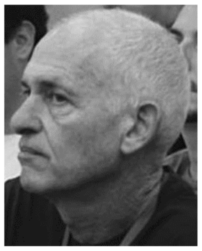
[23] J. T. Butera, "*OpenRAM: An open-source memory compiler*," 2013.

[24] M. Q. Do, M. Drazdziulis, P. Larsson-Edefors, and L. Bengtsson, "Parameterizable architecture-level SRAM power model using circuit-simulation backend for leakage calibration," in *Proc. 7th ISQED'06*.

[25] J. R. Baker, *CMOS: Circuit Design, Layout, Simul.*, vol. 1. Hoboken, NJ, USA: Wiley, 2008.



Amit Kazimirsky received his B.Sc. degree in electrical engineering from Bar-Ilan University, Israel, in 2014, where he is currently working toward his M.Sc. degree. He was a Backend Engineer with Intel from 2013 to 2014, working on power optimization. His research interests include embedded DRAM design for low power and high performance, integrated architecture of static and dynamic RAM optimization for demand applications, and refresh optimization for eDRAM cache memories. In 2014 he was honored with the Bar-Ilan University's outstanding B.Sc. graduate engineering project.



Shmuel Wimer (M'99) received his B.Sc. and M.Sc. degrees in mathematics from Tel-Aviv University, Israel, in 1978 and 1981, respectively, and the D.Sc. degree in Electrical Engineering from the Technion-Israel Institute of Technology in 1988.

From 1978 to 2009, he worked in industry in R&D, engineering, and managerial positions. From 1999 to 2009 he was with Intel, and prior to that with IBM, National Semiconductor, and IAI-Israeli Aerospace Industry. He is an Associate Professor with the Engineering Faculty of Bar-Ilan University, Israel. From 2011 to 2015, he was an Associate Visiting Professor with the Electrical Engineering Faculty of the Technion. He is interested in VLSI circuits and systems design optimization and combinatorial optimization.