Computers and Electrical Engineering 71 (2018) 505–514

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Optimal queuing-based memory refreshing algorithm for energy efficient processors☆

Roi Herman, Binyamin Frankel, Shmuel Wimer*

Engineering Faculty Bar-Ilan University, Ramat-Gan 52900, Israel

ARTICLE INFO

Keywords:
Refreshing
Embedded memories
Queuing
Finite capacity queue

ABSTRACT

The embedded memories of ultra-low power processors require periodic refreshing, which blocks the CPU-memory access and degrades performance. In addition, refreshing queues cause a drop in system performance not only when they are saturated but also when they are empty. We present an optimal queuing-based opportunistic refreshing algorithm that eliminates performance loss. We analyze system performance dependence on queue capacity and memory size to derive a closed-form performance expression that provides clear guidelines for memory design implementation. Comparison of a hardware implementation in a RISC-V ultra-low power processor to ordinary periodic refreshing demonstrates the algorithm can provide a considerable performance speedup in a wide variety of real applications.

1. Rationale

Ultra-low power processors are the enablers of the internet of things (IoT) era, since they operate at very low power supply voltages. However, cache memories, an essential component of all processors, are usually implemented by a *static random access memory* (SRAM) technology that cannot function properly or reliably at low power supply voltages. SRAM replacements by a new type of *embedded dynamic random access memory* (eDRAM) based on *gain-cell* low-cost technology [1] (GCeDRAM) can already be found in certain products [2]. One of the key advantages of GCeDRAM compared to other dynamic memories lies in its two separate read and write ports, which makes it possible to design memories supporting simultaneous read and write operations.

GCeDRAM requires periodic refreshing caused by charge leakage over time. The GCeDRAM *data retention time* (DRT) dictates the refreshing period, which can vary from a few to hundreds of microseconds depending on the technology used. The main drawback to refreshing is that it blocks access to the central processing unit (CPU) read/write (R/W) during the refreshing period. We present an *opportunistic, queuing-based* refreshing algorithm which runs concurrently with CPU R/W operations to overcome this performance degradation, which to the best of our knowledge, has not been suggested elsewhere. Because finite-capacity queues are often filled, thus causing a range of system performance losses, this work focuses on the stochastic characterization of queuing-based refreshing. A detailed model and an analysis of system performance are described and confirm the success of the algorithm in curbing performance losses, exhibiting a perfect match to a real hardware implementation.

The remainder of this paper is organized as follows. Section 2 overviews related work. Section 3 formulates the specific refreshing problem, followed by the derivation of the longest (and hence optimal) refreshing period. Section 4 elaborates on the refreshing algorithm. System performance dependence on memory size, refreshing queue capacity and CPU R/W probabilities are analyzed in

☆ Reviews processed and approved for publication by the Editor-in-Chief.

* Corresponding author.
E-mail address: wimers@biu.ac.il (S. Wimer).

https://doi.org/10.1016/j.compeleceng.2018.07.057
Received 20 July 2018; Accepted 25 July 2018
0045-7906/ © 2018 Elsevier Ltd. All rights reserved.

Section 5 with their implications for memory design implementation. Section 6 presents experimental results obtained from a real hardware implementation. Section 7 draws conclusions.

2. Related work

There are two types of refreshing algorithms. The most common straightforward variety is a periodic algorithm [3,4] termed *administrative* refreshing in [5]. It is also known as *global refreshing* because it refreshes the entire memory sequentially *row-by-row* (we use row and line interchangeably.) Its main drawback is the blockage to system R/W access during the refreshing period. Ways to reduce power consumption and system access blockage are discussed in [6,7].

Global refreshing uses the same ports as ordinary memory access. These are blocked for normal access by the CPU when used for refreshing. Because the GCeDRAM has separate read and write ports, the read and write of individual rows can take place simultaneously. Since refreshing requires a read and write operation, a sequence of contiguous refreshing can be overlapped to deliver an effective refreshing rate of one line per clock cycle. Global refreshing degrades performance because of memory blockage. The cache implementation in [8] reported about 8% performance loss. The authors suggested but did not elaborate on the idea that the under-utilization of L1 from R/W access idleness can be leveraged to reduce performance loss and conceal refresh operations. Another solution to avoid performance degradation is to add extra dedicated R/W refreshing ports, making it independent of the ordinary CPU R/W accesses. This, however, comes at the cost of considerable L1 area and power growth.

The other 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 [9]. 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 [8]. 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 the type of refreshing scheme. To this end [8] 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 expired (are 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. However, although 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 by using the memory blockage periods for other purposes.

The viability of eDRAM was examined in-depth for *Large Last-Level caches* (L^3Cs) in [10]. These 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 32MB L^3C this time can be upwards of 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 [11]. In the eDRAM cache, line refreshing is skipped if it is predicted to be dead, thereby saving energy.

In [12] a concurrent eDRAM refreshing regime was reported that leads to low degradation of memory availability. This was achieved by dividing the cache into 16 banks. For random access, the authors showed 96–99% availability in memories of banks comprising 512–128 lines, respectively. In this method each bank is supplemented by an independent line counter that generates the address of the currently refreshed line. While a certain bank is being addressed for ordinary system access, one or two other banks, defined in a circular manner, are refreshed and increment their respective counter. The authors did not describe what would happen when a certain bank is addressed repeatedly for an arbitrarily long period thus prohibiting refresh for that period, and it must be assumed that some other control enforces 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 3.

A more recent refreshing algorithm, called *versatile refresh*, was described in [13], and was claimed to yield near-optimal throughput. In this case, the algorithm traverses the memory banks in 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 in the form of a time-sliding window, which uses a history shift register. This versatile algorithm nevertheless 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 algorithm avoids this refreshing history bitmap.

Using eDRAM in a GPGPU across its entire memory hierarchy was recently proposed in [4,14] by using the bubble (idle) memory cycles for refreshing. A 4-bit counter for each cache entry was suggested, implying considerable over-refreshing and power overhead. For greater efficiency the authors mentioned replacing them by just two counters for an entire bank. This replacement would nevertheless make the determination of the optimal refreshing period very tricky, an issue that was unfortunately not addressed in either publication. All forms of opportunistic refreshing must also guarantee that if bubbles do not occur for a long period of time, the

data will remain valid. This has a crucial impact on the overall memory availability for CPU R/W but has been ignored in the literature. A different type of concealment of the refreshing penalty for DRAM was proposed in [15] that utilized refreshing in parallel to the write operation which took place at bank granularity, which is too coarse for L1 purposes.

We suggest overcoming this performance degradation with an *opportunistic, queuing-based* refreshing algorithm that runs concurrently with CPU R/W operations. Because finite-capacity queues are often filled, thus causing system performance degradation, research has focused on the stochastic characterization of their filled state [16,17]. By contrast, in our algorithm, system performance degrades not only upon saturation of the queues but also when they are empty. Below we examine the ways in which system performance depends on the queue capacity and its empty and saturated state probabilities, as well as memory size. A closed-form performance expression is derived that provides clear guidelines for memory design implementation.

3. Memory refreshing

Though GCeDRAM authorizes simultaneous read and write by the CPU, it is assumed that the memory is accessed either for read or for write, denoted by the R-cycle and the W-cycle, respectively, which is the situation in ultra-low power processor architectures [18]. We consider the worst case where there are no idle cycles in which the memory (henceforth denoted by M) is not accessed by the CPU. The simpler and easier case where M is idle was analyzed in [19]. There, a refreshing queue was not necessary and refreshing only took place during CPU idle cycles.

Proper refreshing must guarantee that the duration between two successive refreshes of any line does not exceed the *data retention time* (DRT), denoted by N_{DRT}, and measured in clock cycles. Opportunistic refreshing takes place sequentially row-by-row, but not necessarily contiguously in time, because refreshing stalls can occur in R-cycles if Q is empty, or in W-cycles if Q is full. Since CPU R/W access sequences can be arbitrary, there may be insufficient simultaneous refreshing opportunities to comply with the N_{DRT} data retention time period constraint. This requires a supplementary mechanism to ensure that no matter which R/W patterns occur, no N_{DRT} cycles will ever elapse between two successive refreshes of any row of M. If there are insufficient simultaneous refreshing opportunities, the system must initiatively block the CPU access to M and enforce refreshing completion before the N_{DRT} cycles elapse.

Below, we derive the longest, and hence optimal *refreshing round* period, denotedN_{RR} (measured in clock cycles). This constitutes the time window within which all M rows must be refreshed. N_{RR} must ensure that for any CPU R/W access pattern, all the M L_{M} rows are refreshed properly. Since the refreshing of a row first requires reading it into Q and then writing it back into M, a total of $2 \times L_{\mathrm{M}}$clock cycles are required to accomplish refreshing simultaneously with CPU R/W access. Fig. 1(a) illustrates two successive refreshing rounds of N_{RR} cycles. The time stamps at which rows are written back from Q into M are distributed along N_{RR}, and are depicted as gray marks. Let $t'(l)$ be the time stamp when row l, $0 \le l \le L_{\mathrm{M}} - 1$ was refreshed in the first refreshing round, and let $t''(l)$ be the time stamp it was refreshed in the successive one. Proper refreshing must satisfy the data retention time requirement $t''(l) - t'(l) \le N_{\mathrm{DRT}}$ as illustrated in Fig. 1(a).

The worst refreshing case is shown in the second N_{RR} refreshing round on the right side of Fig. 1(b). This occurs when M is accessed by the CPU only for reads, so no row can be read into the refreshing queue Q and $L_{\mathrm{M}} + 1$ cycles are required to enforce refreshing completion. During these $L_{\mathrm{M}} + 1$ cycles the CPU access to M is blocked. On each cycle except the first one when the first M row is read into the empty Q, Q writes back a line into the M row and the next row is read from M into Q.

The larger the N_{RR}, the greater likelihood that a higher portion of L_{M} refreshing will be concealed by simultaneous opportunistic refreshing, and hence a smaller portion (if any at all) will be required to enforce refreshing completion. Thus, the goal is to maximize

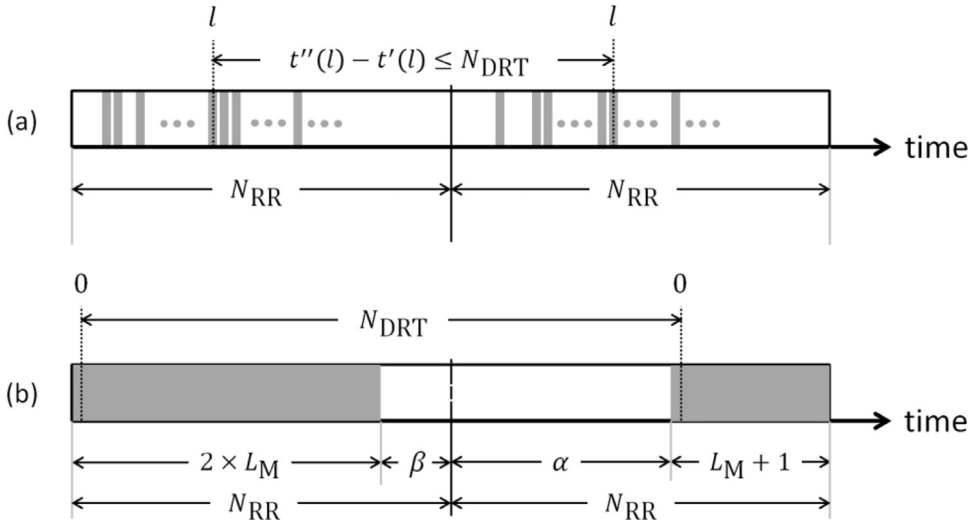

Fig. 1. Derivation of the maximal refreshing round period.

N_{RR}. The worst scenario consisting of two successive N_{RR} periods is shown in Fig. 1(b). This occurs for M's first row when there are $2 \times L_M$ contiguous opportunistic refreshing cycles at the beginning of the previous N_{RR} period, whereas in the subsequent N_{RR} period there are $L_M + 1$ enforced refreshing cycles occurring contiguously at its end. Note the one cycle delay for writing back the first row since it must first be read into Q. Obviously, N_{RR} will be maximized if α and β in Fig. 1(b) are maximized. To ensure proper refreshing where $t''(l) - t'(l) \le N_{DRT}$, $0 \le l \le L_M - 1$, as illustrated in Fig. 1(a), the largest α and β must satisfy

$$(2 \times L_M - 1) + \beta + \alpha + 1 = N_{DRT}. \tag{1}$$

By definition of the refreshing round N_{RR} there is

$$N_{RR} = 2 \times L_M + \beta = \alpha + L_M + 1. \tag{2}$$

It follows from (2) that $\alpha + \beta + 1 = 2N_{RR} - 3L_M$. Substituting it into (1) yields$2N_{RR} = N_{DRT} + L_M + 1$. Division by two and taking the floor since N_{RR} must be an integer yields

$$N_{RR} = \lfloor (N_{DRT} + L_M + 1)/2 \rfloor. \tag{3}$$

Note that the addition of just a single cycle to N_{RR} in (3) will violate proper refreshing if the worst case scenario shown in Fig. 1(b) occurs. Hence (3) is indeed optimal, maximizingN_{RR}.

4. Simultaneous opportunistic refreshing algorithm

M refreshing takes place sequentially, line-by-line, *simultaneously* and in coordination with the CPU access to M. Simultaneously means that while the CPU is reading from or writing into M, refreshing is engaged in a counter operation; namely, writing into or reading from a refreshed line of M within the same clock cycle. Fig. 2 depicts the hardware implementation of simultaneous refreshing. The right side illustrates how refreshed data are transferred to and from M. In an M W-cycle the CPU read is disabled and the read port is used to read a line into the refreshing register. In an M R-cycle the CPU write is disabled and the write port is used to write the contents of the refreshing register back into M. Since there may be write sequences where no read intervenes and vice versa, it is clear that a single register may lead to a loss of refreshing opportunities, so a register queue (FIFO) Q is needed.

The memory data bus connected to the CPU is bidirectional and is used for both read and write. A *refreshing round* takes place *simultaneously* with CPU access. In a W-cycle a line is read into Q, whereas in an R-cycle a Q line is written back into M. This simultaneous access and refreshing during the same cycle is a significant improvement over existing methods where refreshing only takes place during the idle cycle. Refreshing during the idle cycle writes the head line of Q back into M and reads the next line of M into the tail of Q.

The simultaneous opportunistic refreshing algorithm requires the three down-counters shown on the left side of Fig. 2. The first counter is c_{RR}, $N_{RR} - 1 \ge c_{RR} \ge 0$, which counts the refreshing round cycle-by-cycle. The second counter is c_{MR}, $L_M - 1 \ge c_{MR} \ge 0$, which determines the M row to be read next into Q. The third counter is c_{MW}, $L_M - 1 \ge c_{MW} \ge 0$, which determines the next M row to be written back from Q. The three counters are synchronized by the system clock and set to their initial values simultaneously.

Fig. 3 shows the relationship between c_{RR} and c_{MW}. For any time $N_{RR} - 1 \ge t \ge 0$ of a refreshing round (measured in clock cycles) there is

$$L_Q \ge c_{MW}(t) - c_{MR}(t) = Q(t) \ge 0 \tag{4}$$

where $Q(t)$ is the size of Q at time t. The counter c_{RR} is unconditionally decremented at every clock cycle$N_{RR} - 1 \ge t \ge 0$. Upon a CPU W-cycle, an M row is read into the Q tail and c_{MR} is decremented, whereas c_{MW} is unchanged. Upon a CPU R-cycle Q writes its head back into line c_{MW} of M and c_{MW} is decremented, whereas c_{MR} is unchanged. Initially $c_{RR} > c_{MW} + 1$, and as long as this inequality holds, as shown in Fig. 3(a), simultaneous opportunistic refreshing proceeds safely. Once $c_{RR} = 0$, the M refreshing round is completed and the counters are initialized to $c_{RR} = N_{RR} - 1$ and $c_{MR} = c_{MW} = L_M - 1$.

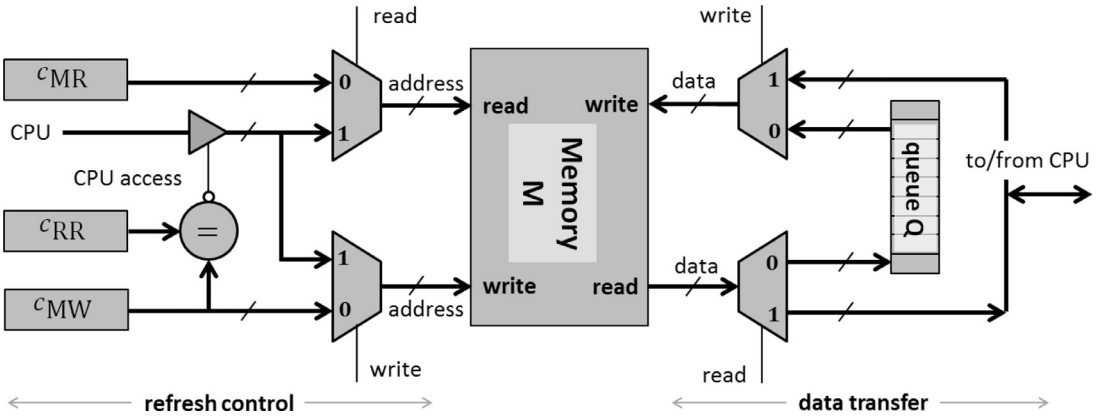

Fig. 2. Refreshing queue working simultaneously with the CPU access to the memory.

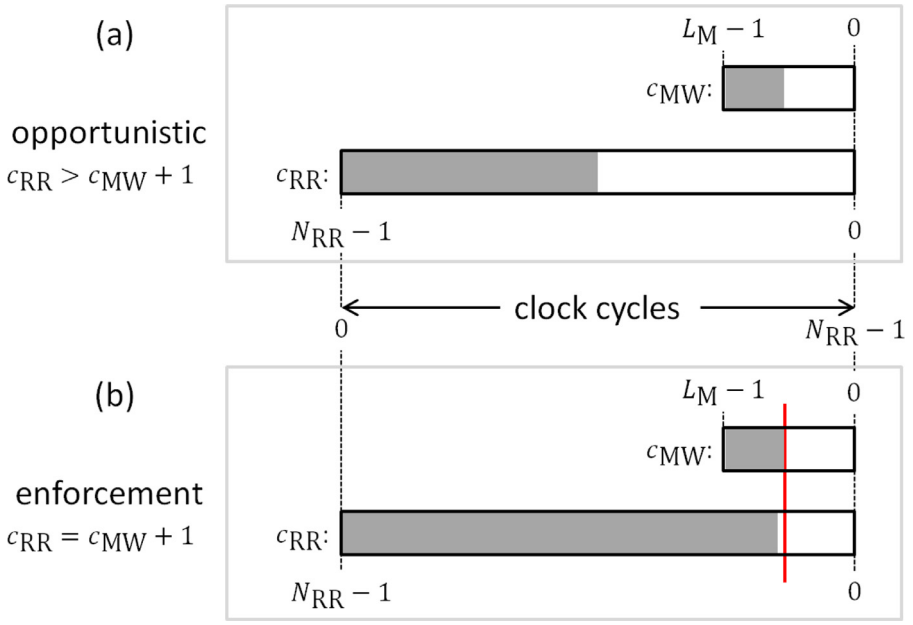

Fig. 3. Refreshing modes: (a) opportunistic, (b) enforcement of refreshing completion.

If it occurs at some cycle τ, $N_{RR} - 1 > \tau > 0$, that $c_{RR}(\tau) = c_{MW}(\tau) + 1$, as shown in Fig. 3(b), refresh completion must be enforced. Otherwise there will not be enough cycles to write back and complete the refreshing within the N_{RR} period. Enforced refreshing stalls the CPU access to M for $c_{MW}(\tau) + 1$ cycles by disabling the CPU memory bus in Fig. 2. Since the read and write ports of M are not being activated by the CPU, at each cycle a line is read from M into the tail of Q and a line is written from the head of Q back to M, and all the three counters are decremented. Once $c_{RR} = 0$, the refreshing round is completed and the counters are set to their initial values as before. Note from (4) that c_{MR} never lags behind c_{MW}, but rather reaches zero earlier and waits $Q(\tau)$ cycles until c_{RR} and c_{MW} finish counting. The performance degradation incurred by refreshing enforcement is discussed in the next section. Note that $c_{MW}(\tau) + 1$ cycles are required for refreshing completion if $Q(\tau) = 0$, whereas $c_{MW}(\tau)$ suffice if $Q(\tau) > 0$.

5. Memory size optimization

The cache memories of microprocessors are too large to be monolithic, and hence are commonly divided into smaller units, each of which is a self-contained refreshable unit (Fig. 2 in [19]). In our terminology, a unit is the memory M of Fig. 2. Although the total cache capacity L_{cache} is defined by the system architecture, its division into refreshable units of size L_M each is a matter of design implementation choice. A cache thus consists of L_{cache}/L_M units, each of which involves a non-negligible refreshing hardware overhead. Fig. 2 shows that M involves Q and counters. Whereas c_{RR} is an absolute time counter and thus can be shared by all Ms, each M has its own down-counters c_{MR} and c_{MW} with its associated logic.

Maximization of L_M will minimize the total hardware overhead. On the other hand, the larger the L_M, the greater the likelihood that simultaneous opportunistic refreshing will not suffice (see (3) and Fig. 1), and refreshing completion which stalls the system will be enforced. This causes performance degradation, which should be minimized; hence, L_M should be small enough. Another interesting issue is how the capacity L_Q affects system performance. A larger Q will apparently yield more opportunistic refreshing, and hence higher performance. Large L_Q, however, constitutes hardware overhead.

To calculate the probability that refreshing completion enforcement will occur, which is the cause of performance loss, below we trade off the above-mentioned conflicting requisites of maximizing and minimizing L_M by capturing L_Q into the tradeoff. It is assumed that the CPU R-cycle and W-cycle occur with probabilities of $1 \geq \mu \geq 0$ and $\lambda = 1 - \mu$, respectively. The performance loss of N_{stall} cycles, $L_M + 1 \geq N_{stall} \geq 0$, within a N_{RR} refreshing round period depends on the ratio L_M/N_{RR}. The higher the ratio, the more probable the refreshing completion enforcement is, and hence of a largerN_{stall}.

Performance loss does not occur if refreshing is able to be purely opportunistic. This implies that a line was read from M into Q and written back from Q into M L_M times, utilizing a total of $2 \times L_M$ clock cycles. Reading a line from M into Q upon W-cycle is impossible if Q is saturated, so a refreshing opportunity is lost. Similarly and symmetrically, writing back a line from Q into M upon an R-cycle is impossible if Q is empty.

Fig. 4 illustrates the possible states of Q filling $\{E, 0, 1, \ldots, L_Q - 1, L_Q, S\}$ and their transition probabilities μ and λ, upon R and W cycles, respectively, denoted in the transition diagram by R/μ and W/λ. This is a special case of a finite capacity queue. In an ordinary queue, a state never transitions to itself. In the queue in Fig. 4 each of the emptied and filled states surrounded by the dashes, are further split into two sub-states. For the grayed sub-states, performance is lost since a self-transition means that a refreshing operation will not take place due to respective emptiness or saturation. Each state $1 \leq i \leq L_Q - 1$ passes into state $i + 1$ with probability λ and

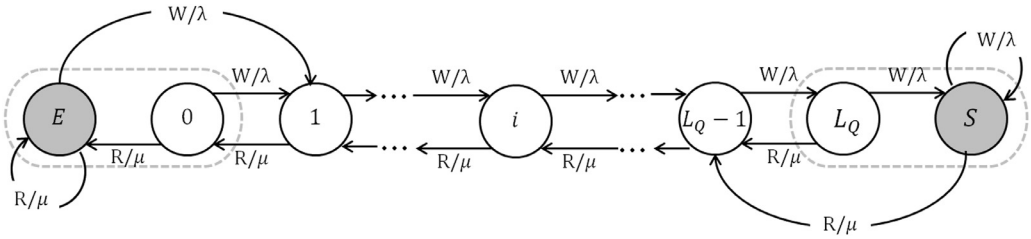

Fig. 4. State transitions of refreshing queue filling.

to state $i - 1$ with probability μ.

An R-cycle in state 0 causes a queue emptying cycle loss, designated by state *E*. Subsequent R-cycles leave *Q* in state *E*. A W-cycle occurring in state *E* passes *Q* directly to state 1. A symmetric situation occurs in state L_Q, where a W-cycle causes a queue filling cycle loss, designated by state *S*. Subsequent W-cycles will result in a queue filling cycle loss, leaving *Q* in state *S*. An R-cycle occurring in state *S* passes *Q* directly to state $L_Q - 1$. The state transitions imply a stationary Markov chain with a corresponding $(L_Q + 3) \times (L_Q + 3)$ state transition probability matrix $\mathbf{P} = (P_{ij})$, $i, j \in \{E, 0, \ldots, L_Q, S\}$ as illustrated in Fig. 5, where the corresponding states are shown on the left and top sides.

Cycles of opportunistic reads into and writing back from *Q* occur when *Q* is in any of the states $0 \le i \le L_Q$, but never in states *E* and *S*. Therefore, the probabilities p_E and p_S answer the question of whether there are sufficient opportunistic refreshing cycles to refresh the L_M memory lines. The probabilities $p_E, p_0, \ldots, p_{L_Q}, p_S$ are obtained from $\lim_{n\to\infty} \mathbf{P}^n$ by solving the linear system [20]

$$p_j = \sum_{i\in\{E,0,\cdots,L_Q,S\}} p_i P_{ij}, j \in \{E, 0, \cdots, L_Q, S\} \tag{5}$$

together with

$$\sum_{i\in\{E,0,\cdots,L_Q,S\}} p_i = 1 \tag{6}$$

Defining $\rho \triangleq \lambda/\mu$, by some algebraic manipulations the solution of (5) and (6) yields the probabilities of emptied and saturated *Q* as follows

$$p_E = \frac{1-\rho}{(1+\rho)(1-\rho^{L_Q+1})} \tag{7}$$

and

$$p_S = \frac{(1-\rho)\rho^{L_Q+1}}{(1+\rho)(1-\rho^{L_Q+1})} \tag{8}$$

The expressions of P_E in (7) and p_S in (8) are not defined for $\rho \triangleq \lambda/\mu = 1$, but by applying l' Hôpital's rule, we get $\lim_{\rho\to 1} p_E = \lim_{\rho\to 1} p_S = 1/(2L_Q)$. The probability of a successful opportunistic refreshing cycle is

$$1 - (p_E + p_S) = \frac{2\rho(1-\rho^{L_Q})}{(1+\rho)(1-\rho^{L_Q+1})} \tag{9}$$

$$\mathbf{P} = \begin{array}{c} \\ E \\ 0 \\ 1 \\ \vdots \\ i \\ \vdots \\ L_Q \\ S \end{array} \begin{array}{c} \begin{array}{cccccccc} E & 0 & 1 & \cdots & i & \cdots & L_Q & S \end{array} \\ \begin{bmatrix} \mu & 0 & \lambda & & & & & \\ \mu & 0 & \lambda & & & & 0 & \\ & \mu & 0 & \lambda & & & & \\ & & \ddots & \ddots & & & & \\ & & & \mu & 0 & \lambda & & \\ & & & & \ddots & \ddots & & \\ & 0 & & & & \mu & 0 & \lambda \\ & & & & & \mu & 0 & \lambda \end{bmatrix} \end{array}$$

Fig. 5. Refreshing queue filling state transition probability matrix.

This is the probability that Q is neither emptied nor saturated, enabling writing back from Q to M upon a CPU R-cycle, and reading from M into Q upon CPU W-cycle.

Since there may be at most L_Q reads from M into Q before a line is written back from Q into M, there is some $0 \le \varepsilon \le L_Q$ such that if

$$(1 - p_E - p_S)N_{\mathrm{RR}} \ge 2L_{\mathrm{M}} + \varepsilon \tag{10}$$

there will surely be enough simultaneous opportunistic refreshing cycles, so refreshing completion enforcement will not be required, and no performance loss occurs. Note that since $L_Q \ll L_{\mathrm{M}}$, ε can be neglected for all practical considerations. If

$$(1 - p_E - p_S)N_{\mathrm{RR}} < 2L_{\mathrm{M}} + \varepsilon \tag{11}$$

the refreshing deficit $N_{\mathrm{comp}} = 2L_{\mathrm{M}} + \varepsilon - (1 - p_E - p_S)N_{\mathrm{RR}}$ needs to be recovered by enforcing refreshing completion along N_{comp} clock cycles. Recalling that at each of these N_{comp} cycles both line insertion into and deletion from Q is performed (the CPU access in Fig. 2 is disabled), we get

$$N_{\mathrm{comp}} = L_{\mathrm{M}} + \tfrac{1}{2}\varepsilon - \tfrac{1}{2}(1 - p_{\mathrm{E}} - p_{\mathrm{S}})(N_{\mathrm{RR}} - N_{\mathrm{comp}}) \tag{12}$$

Substitution of (9) in (12), N_{comp} is solved to

$$N_{\mathrm{comp}} = \frac{\left(L_{\mathrm{M}} + \frac{1}{2}\varepsilon\right)(1+\rho)(1-\rho^{L_Q+1}) - N_{\mathrm{RR}}\rho(1-\rho^{L_Q})}{1-\rho^{L_Q+2}} \tag{13}$$

Thus, from (9), (10) and (13) it can be concluded that the performance loss $0 \le \gamma = N_{\mathrm{comp}}/N_{\mathrm{RR}} \le 1$ incurred by refreshing completion enforcement is

$$\gamma = \begin{cases} 0 & \text{if} \frac{2\rho(1-\rho^{L_Q})}{(1+\rho)(1-\rho^{L_Q+1})} N_{\mathrm{RR}} \ge 2L_{\mathrm{M}} + \varepsilon \\ \frac{\left(L_{\mathrm{M}} + \frac{1}{2}\varepsilon\right)(1+\rho)(1-\rho^{L_Q+1}) - N_{\mathrm{RR}}\rho(1-\rho^{L_Q})}{N_{\mathrm{RR}}(1-\rho^{L_Q+2})}, & \text{otherwise} \end{cases} \tag{14}$$

Eq. (14) provides a closed-form expression that captures the memory size L_{M}, the memory write to read probability ratio ρ and the refreshing queue capacity L_Q.

Fig. 6 illustrates the CPU access performance $1 - \gamma$ obtained from (14) for $L_Q = 1$, $L_Q = 2$ and $L_Q = 8$. The symmetry of the surface around $\lambda = \mu = 0.5$ follows from $\gamma(\rho) = \gamma(1/\rho)$. As noted above, the smaller the L_{M}, the larger the $1 - \gamma$. When $N_{\mathrm{RR}} = L_{\mathrm{M}}$ refreshing must take place in each cycle; hence, the CPU access to the memory is always blocked and $1 - \gamma = 0$. As L_{M} gets smaller, more cycles are left for useful CPU access and performance increases accordingly. Performance also depends on the R/W probabilities, $1 \ge \lambda \ge 0$ and $\mu = 1 - \lambda$, respectively. For a very small λ Q will often be empty, because it cannot supply rows to write back into M upon R-cycles. Oppositely and symmetrically, a very small μ will often saturate Q, avoiding reading the M lines into Q upon W-cycles. These phenomena are illustrated by the decline of the performance surface towards the margins at $\mu = 1$ and $\mu = 0$.

Note that for a sufficiently large $N_{\mathrm{RR}}/L_{\mathrm{M}}$ and for some symmetric interval around $\mu = \lambda = 1/2$ there is no performance loss, as shown by the flat surface enclosed by the red curve in Fig. 6. Recalling that we are interested in minimizing the refreshing hardware overhead by keeping L_{M} as large as possible while minimizing the performance loss, this curve defines the optimal design points in the following sense. Given N_{RR}, L_Q and μ, if zero performance loss is achievable for some L_{M}, the largest, and hence optimal L_{M} should be selected on the curve. There is no point in choosing an internal design point of the flat surface, which only decreases L_{M} without increasing performance. Other than the flat surface, any combination of L_{M} and μ uniquely defines the memory access performance loss.

To validate the analytical solution in (14), the design in Fig. 2 was implemented in hardware and integrated into a fully featured RISC-V based ultra-low power processor called PULPino [18] to replace its SRAM data memory by a GCeDRAM refreshable one. The HDL Veilog code was synthesized to gate-level and then simulated with a Mentor Graphics ModelSim simulator. We used random R/

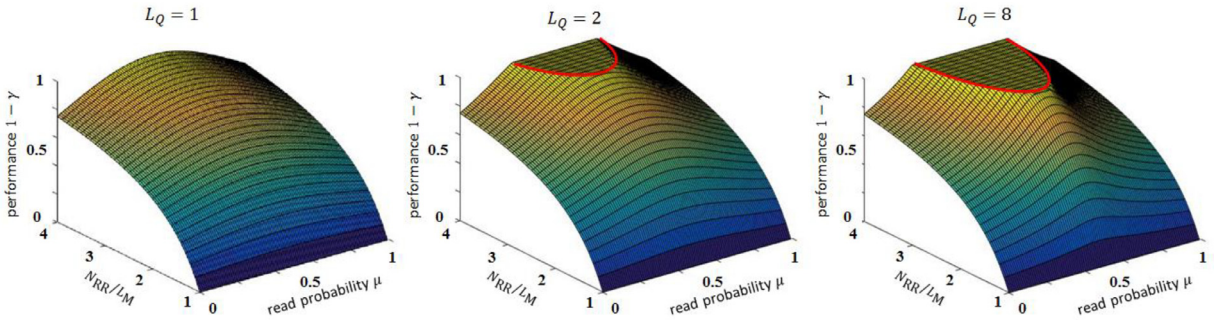

Fig. 6. Memory access performance in an analytical model.

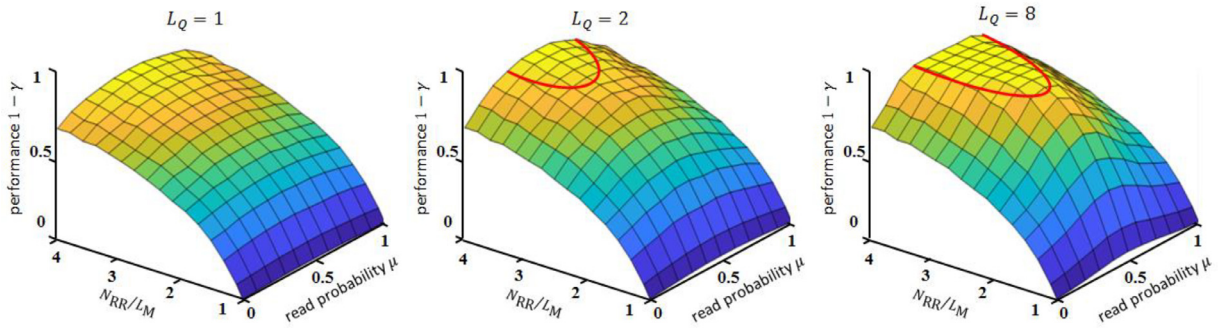

Fig. 7. Memory access performance in a hardware implementation.

W traces for$L_Q = 1$, $L_Q = 2$ and $L_Q = 8$, and various $0 \le \mu \le 1$ and N_{RR}/L_M ratios. The simulated performance surfaces illustrated in Fig. 7 are a perfect match to the analytical ones in Fig. 6.

As expected, Fig. 6 shows that larger L_Q yield higher performance. To grasp the impact of L_Q, the boundary curve of the zero performance loss in Fig. 6 can be derived from (14) for any choice of L_Q, yielding the parametric equation depicted in Fig. 8 for $1 \le L_Q \le 10$. It is clear that the increase of L_Q beyond a certain point hardly increases the zero performance loss area. It can be deduced easily from (14) that $\lim_{L_Q \to \infty} \partial\gamma/\partial L_Q = 0$.

6. Experimental results

As depicted in Figs. 6 and 7, the hardware simulations matched the analytic model under random R/W traces. To demonstrate the advantages of queuing-based GCeDRAM opportunistic refreshing, we designed two memory refreshing controllers. We first implemented the ordinary periodic controller discussed in Section 2, which is the one most commonly used by DRAM. This was compared to the proposed queue-based opportunistic controller. We employed a test bench comprised of a range of eight applications. Their corresponding RISC-V machine instruction codes were simulated with the fully featured PULPino processor [18]. The speedup results are shown in Table 1. The upper part indicates the percentage of the CPU memory reads and writes. The lower part shows the speedup of the underlying applications in run-time obtained by opportunistic refreshing compared to ordinary periodic refreshing. The opportunistic refreshing also used idle cycles, where refreshing could perform both read and write on the same cycle. Run-time speedups from 1.25 to 11.30 were achieved. For each test and each entry $(N_{RR}/L_M, L_Q) \in \{1.5, 1.3, 1.1\} \times \{1, 8\}$ in the table, the speedup was calculated by dividing the total number of clock cycles required to complete the underlying test when using periodic refreshing by that required by queuing-based refreshing.

Section 1 showed that the advantage of the full GCeDRAM stems from its very low operation voltage and very small area. This dictates a short DRT [21], which from (3) results in a short refreshing round periodN_{RR} and the small size of the refreshable unit L_M. The PULPino architecture features 32 KBytes of L1 data memory, of which we used refreshable units (banks) of 2 KBytes size each, similar to the one recently designed and manufactured in [22]. We considered N_{RR}/L_M of 1.5, 1.3 and 1.1. Note that performance improvement considerably increased with the decrease in N_{RR}/L_M. This may be a crucial factor in ultra-low power designs using very

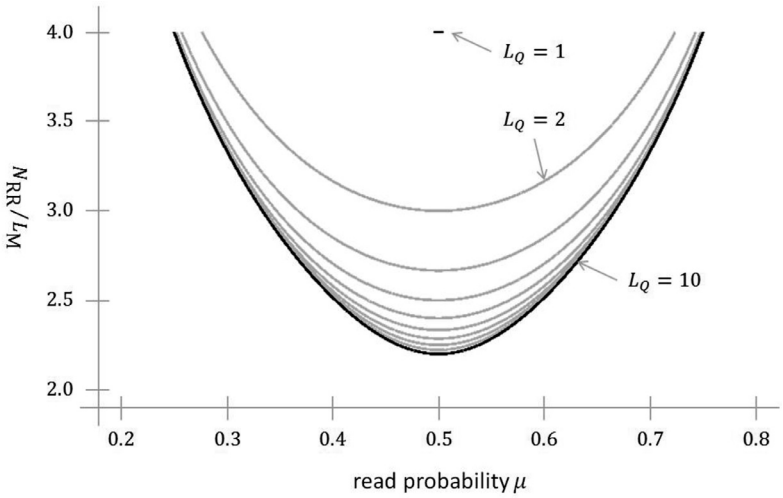

Fig. 8. Zero performance loss dependence on queue capacity.

Table 1
Speedup obtained by queuing-based refreshing compared to periodic refreshing.

Test		2D convolution		FFT		SHA (crypto hash)		Bubble sort		Matrix multiplication		CRC32		Dot-product		Fibonacci	
CPU read [%]		32.52		19.61		20.00		37.39		34.11		9.38		29.72		3.96	
CPU write [%]		1.82		9.67		13.02		16.77		2.87		0.04		2.41		4.91	
Queue size		1	8	1	8	1	8	1	8	1	8	1	8	1	8	1	8
N_{RR}/L_M	1.5	1.25	1.25	1.64	2.09	1.82	2.32	1.42	1.93	1.80	1.81	2.91	2.91	1.65	1.67	1.43	1.43
	1.3	1.84	1.88	2.57	4.06	2.92	4.38	2.22	3.11	2.60	2.60	7.36	7.36	2.08	2.13	1.97	2.03
	1.1	2.66	2.79	2.60	4.40	2.45	3.62	2.03	2.98	3.06	3.11	11.3	11.3	2.74	3.06	2.13	2.19

low operation voltage. There, the DRT may be subject to wide variability, yielding worst-case bit cells DRT of very few microseconds [22]. To avoid refreshing failures, the counter c_{RR} shown in Fig. 2 can be configured at silicon testing so that N_{RR} is set appropriately.

The impact of the refreshing queue capacity can clearly be seen in the performance speedup achieved by $L_Q = 8$ compared to $L_Q = 1$. Although for some of the tests the improvement was just a few percent, in some tests it reached 70%, where the impact of L_Q was greater when $N_{RR}/L_M$was small. This can be accounted for by the relative numbers of reads and writes. A large queue is mostly effective when there is a high percentage of reads and writes. Otherwise, most of the refreshing occurs in M idle cycles. There, a read into Q and write back into M (of different lines) take place in one cycle, without changing the Q filling, so that L_Q hardly matters.

7. Conclusion

A queuing-based opportunistic refreshing algorithm for gain-cell embedded dynamic L1 cache memories enabling the design of low-voltage ultra-low power processors was proposed. We showed how this refreshing avoids a great deal of the blockages to the memory access by the CPU incurred by refreshing. An essential component of this algorithm is its queue, whose impact on system performance was examined by using a stochastic model. The interrelation between memory size, the CPU read/write probabilities and the queue capacity was derived as a closed-form expression which matched the simulations of hardware implementation perfectly. This expression is conclusive and can be implemented to optimally architect memories for ultra-low power processors to maximize their performance. The setting of the refreshing period and the refreshable unit size underscores the optimality of the refreshing. A hardware implementation of the opportunistic refreshing controller as a part of a RISC-V ultra-low power processor was compared to ordinary periodic refreshing, and exhibited a performance speedup factor of 1.25–11.30 for a wide range of real applications.

Acknowledgments

This work was supported by the Israel Innovation Authority (MAGNET program) under the HiPer consortium. The authors are grateful to Prof. L. Benini of ETHZ and Prof. D. Rossi of the University of Bologna and their team for making available and supporting the PULPino design. We acknowledge the useful comments by the anonymous reviewers who enabled us to improve the manuscript.

References

[1] Teman A, Meinerzhagen P, Burg A, Fish A. Review and classification of gain cell eDRAM implementations. Electrical & electronics engineers in Israel (IEEEI), 2012 IEEE 27th convention of. 2012.
[2] Chun KC, Jain P, Kim T-H, Kim CH. A 667 MHz logic-compatible embedded DRAM featuring an asymmetric 2T gain cell for high speed on-die caches. IEEE J Solid-State Circ 2012;47(2):547–59.
[3] Rabaey JM, Chandrakasan AP, Nikolic B. Digital integrated circuits Ch. 12. 2nd ed. Prentice hall; 2002.
[4] Jing N, Jiang L, Zhang T, Li C, Fan F, Liang X. Energy-Efficient eDRAM-based on-chip storage architecture for GPGPUs. IEEE Trans Comput 2016;65(1):122–35.
[5] Reohr RW. Memories: exploiting them and developing them. IEEE international SOC conference, 2006. 2006.
[6] Liu J, Jaiyen B, Veras R, Mutlu O. RAIDR: Retention-aware intelligent DRAM refresh. ACM SIGARCH Comput Arch News 2012;40(3):1–12.
[7] Liu S, Pattabiraman K, Moscibroda T, Zorn BG. Flikker: saving DRAM refresh-power through critical data partitioning. ACM SIGPLAN Not 2012;47(4):213–24.
[8] Xiaoyao L, Canal R, Wei G-Y, Brooks D. Process variation tolerant 3T1D-based cache architectures. Proceedings of the 40th annual IEEE/ACM international symposium on microarchitecture. 2007. p. 15–26.
[9] Teman A, Meinerzhagen P, Giterman R, Fish A, Burg A. Replica technique for adaptive refresh timing of gain-cell-embedded DRAM. IEEE Trans. Circ Syst II 2014;61(4):259–63.
[10] Chang M-T, Rosenfeld P, Lu S-L, Biji J. Technology comparison for large last-level caches (L^3Cs): low-leakage SRAM, low write-energy STT-RAM, and refresh-optimized eDRAM. High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th international symposium on. 2013.
[11] Kaxiras S, Hu Z, Martonosi M. Cache decay: exploiting generational behavior to reduce cache leakage power. ACM SIGARCH Comput Arch News 2001;29(2):240–51.
[12] Toshiaki K, Parries P, Hanson DR, Kim H, Golz J, Fredeman G, Rajeevakumar R, Griesemer J, Robson N, Cestero A, Khan BA, Wang G, Wordeman M, Iyer SS. An 800-MHz embedded DRAM with a concurrent refresh mode. IEEE J Solid-State Circ 2005;40(6):1377–87.
[13] Alizadeh M, Javanmard A, Chuang S-T, Iyer S, Lu Y. Versatile refresh: low complexity refresh scheduling for high-throughput multi-banked eDRAM. ACM SIGMETRICS Perform Eval Rev 2012;40(1):247–58.
[14] Jing N, Shen Y, Lu Y, Ganapathy S, Mao Z, Guo M, Canal R, Liang X. An energy-efficient and scalable eDRAM-based register file architecture for GPGPU. ACM SIGARCH Comput Arch News 2013;41(3).
[15] Chang K-WK, Lee D, Chishti Z, Alameldeen AR, Wilkerson C, Kim Y, Mutlu O. Improving DRAM performance by parallelizing refreshes with accesses. IEEE 20th international symposium on High Performance Computer Architecture (HPCA), 2014. 2014.
[16] Samouylov K, Sopin E, Gudkova I. Sojourn time analysis for processor sharing loss queuing system with service interruptions and MAP arrivals. International conference on distributed computer and communication networks. 2016.

[17] Zhen Q, Knessl C. On sojourn times in the finite capacity M/M/1 queue with processor sharing. Oper Res Lett 2009;37(6):447–50.
[18] Traber A, Zaruba F, Stucki S, Pullini A, Haugou G, Flamand E, Gürkaynak FK, Benini L. https://riscv.org/wp-content/uploads/2016/01/Wed1315-PULP-riscv3_noanim.pdf; 2016. Online.
[19] Kazimirsky A, Wimer S. Opportunistic refreshing algorithm for eDRAM Memories. IEEE Trans Circ Syst I 2016;63(11):1921–32.
[20] Ross SM. Introduction to probability models. Academic press; 2014.
[21] Somasekhar D, Ye Y, Aseron P, Lu S-L, Khellah MM, Howard J, Ruhl G. 2 GHz 2 Mb 2T gain cell memory macro with 128 GBytes/sec bandwidth in a 65 nm logic process technology. IEEE J Solid-State Circ 2009;44(1):174–85.
[22] Giterman R, Fish A, Burg A, Teman A. A 4-transistor nMOS-only logic-compatible gain-cell embedded DRAM with over 1.6-ms retention time at 700 mV in 28-nm FD-SOI. IEEE Trans Circt Syst I 2017.

Roi Herman received his B.*Sc*. degree (cum laude) in Computer Engineering from Bar-Ilan University, Israel, in 2015, where he is currently working toward his M.*Sc*. degree. He was with Intel from 2013 to 2017. He is now with Samsung, working as a logic design engineer. He is interested in refreshing architecture optimization for eDRAM memories, and approximate computing methods.

Binyamin Frankel received his B.*Sc*. and M.*Sc*. degrees in Electrical Engineering from Bar-Ilan University in 2014 and 2016, where he is currently pursuing his Ph.D. degree in Computer Engineering. He is interested in VLSI circuits and systems design optimization.

Shmuel Wimer received his B.*Sc*. and M.*Sc*. degrees in Mathematics from Tel-Aviv University, and his D.*Sc*. degree in Electrical Engineering from the Technion-Israel Institute of Technology. From 1978 to 2009 he worked in industry. He is an Associate Professor at the Engineering Faculty of Bar-Ilan University, Israel. He is interested in VLSI circuits and systems design optimization and combinatorial optimization.