Queuing-Based eDRAM Refreshing for Ultra-Low Power Processors

Binyamin Frankel, Roi Herman, and Shmuel Wimer, *Member, IEEE*

Abstract—Ultra-low power processors designed to work at very low voltage are the enablers of the internet of things (IoT) era. Their internal memories, which are usually implemented by a static random access memory (SRAM) technology, stop functioning properly at low voltage. Some recent commercial products have replaced SRAM with embedded memory (eDRAM), in which stored data are destroyed over time, thus requiring periodic refreshing that causes performance loss. This article presents a queuing-based opportunistic refreshing algorithm that eliminates most if not all of the performance loss and is shown to be optimal. The queues used for refreshing miss refreshing opportunities not only when they are saturated but also when they are empty, hence increasing the probability of performance loss. We examine the optimal policy for handling a saturated and empty queue, and the ways in which system performance depends on queue capacity and memory size. This analysis results in a closed-form performance expression capturing read/write probabilities, memory size and queue capacity leading to CPU-internal memory architecture optimization.

Index Terms—Embedded cache memories, finite capacity queue, queuing, refreshing

1 Introduction

Ultra-low power processors are a crucial feature in the internet of things (IoT) era, since they operate at a very low power-supply voltage. Cache memories, an essential component of any processor, are implemented by a static random access memory (SRAM) technology that cannot function properly or reliably in ultra-low power processors. Recent calls to replace them with embedded dynamic random access memory (eDRAM) have been heeded and incorporated in certain commercial products [1], [2]. However, bit cells of ordinary eDRAM store their data in special capacitors, which significantly increases the manufacturing cost of processors. They also suffer from charge destruction during the read operation that requires immediate, power expensive refreshing for data restoration. This read-refresh is needed in addition to the mandatory periodic refreshing caused by charge leakage over time.

A new type of bit cell known as the *gain-cell* (GC) has recently been devised [3]. The GC eDRAM (GCeDRAM) has a low manufacturing cost, and avoids the read-refresh operation, but still requires periodic refreshing due to charge leakage. A key advantage of GC is its two separate read and write ports, which makes it possible to architect memories supporting simultaneous read and write. The GCeDRAM *data retention time* (DRT) dictates the refreshing period, and can vary from a few to hundreds of microseconds [4] depending on the bit cell structure and the technology used. As in the case of the refreshing operation of any eDRAM, GCeDRAM is blocked for system access, thus causing performance loss.

The most commonly used and straightforward refreshing algorithms are *periodic* [5], [6] that refresh the entire memory sequentially *row-by-row*. Here, we use the term row and line interchangeably. Their main drawback is the blockage of the central processing unit (CPU) read/write (R/W) access during the refreshing period. Methods to reduce this blockage have been discussed in [7], [8]. An overview of refreshing algorithms for DRAM/eDRAM can be found in [9]. Refreshing uses the same ports as for CPU access. Due to GCeDRAM's 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 pipelined to deliver an effective refreshing rate of one cycle per line.

We suggest overcoming the performance degradation problem with an *opportunistic*, *queuing-based* refreshing algorithm that has the advantage of not intervening in the normal memory access. Rather, refreshing takes place concurrently with CPU R/W operations. To overcome the problem of probable insufficient opportunistic refreshing, initiative refreshing completion is enforced.

We examine three key issues. The first addresses the maximization of the refreshing period. The second deals with the optimal policy of handling a saturated queue, and the third explores the ways in which system performance depends on queue capacity and memory size. The analysis yields a closed-form performance expression, providing clear guidelines for the optimization of GCeDRAM memories in ultra-low power processor designs.

The remainder of this paper is organized as follows. Section 2 describes the memory-CPU interface and the

• *The authors are with the Engineering Faculty, Bar-Ilan University, Ramat-Gan 52900, Israel.*
E-mail: {binyamin.frankel, roiher}@gmail.com, wimers@biu.ac.il.

Manuscript received 30 Aug. 2017; revised 24 Dec. 2017; accepted 27 Feb. 2018. Date of publication 7 Mar. 2018; date of current version 15 Aug. 2018.
(Corresponding author: Shmuel Wimer.)
Recommended for acceptance by Z. Shao.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2018.2811470

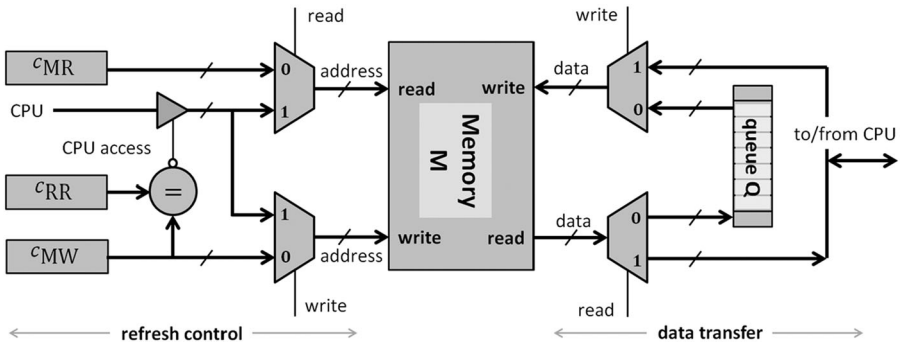

Fig. 1. Refreshing queue working simultaneously with the CPU access to the memory.

refreshing process, followed by the derivation of the longest (and hence optimal) refreshing period. Section 3 elaborates on the refreshing algorithm. Section 4 discusses the implications of saturated and empty refreshing queues. System performance dependence on memory size, refreshing queue capacity and CPU $\mathrm{R/W}$ probabilities are examined in Section 5 via a stochastic model. The generalization of the CPU-memory access model is presented in Section 6. Section 7 presents the experimental results and Section 8 draws the conclusions.

2 Dynamic Memory Refreshing

Though GCeDRAM allows simultaneous read and write by the CPU (denoted as the $\mathrm{R+W}$ cycle), we first assume that the memory is accessed either for read or for write, (denoted as the $\mathrm{R/W}$ cycle), but not both. This is in any case the situation in ultra-low power processor architectures [10]. We use M to denote the memory and consider the worst case where there are no idle cycles in which M is not accessed by the CPU. The simpler case where there is either a $\mathrm{R+W}$ cycle or M is idle was studied in [9]. There, refreshing took place only at idle cycles. We show below that accounting for idle and $\mathrm{R+W}$ cycles do not change the conclusions when only assuming $\mathrm{R/W}$ access; hence there is no loss of generality in terms of the analysis.

M refreshing takes place sequentially, line-by-line, *simultaneously* and in coordination and with the CPU access to M. Simultaneously means that while the CPU is reading from or writing into M, the refreshing is performing a counter operation; namely, writing into or reading from a refreshed line of M within the same clock cycle. Fig. 1 shows the hardware implementation of the simultaneous refreshing. The right side illustrates how refreshed data is 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 a 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 cause a loss of refreshing opportunities, so a register queue (FIFO) Q is needed. We subsequently use the terms queue and buffer interchangeably.

The usage of buffers (queues) in computing hardware is very common. The numerous other applications include the cache memories using write-through buffers [11], and the reorder buffer (ROB) used in high-performance processors [12]. Due to their finite size, such buffers are often filled, which usually degrades system performance. Research on finite-capacity queues therefore has tended to focus on the stochastic characterization of their filled state [13], [14]. The queue of finite size in this paper has the unique property that system performance is lost not only upon saturation but also when the queue is empty.

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 refreshing methods where M is either accessed for $\mathrm{R+W}$ which makes refreshing impossible, or M has an idle cycle. Refreshing idle cycle can take place by writing the head line of Q back into M, and reading the next line of M into the tail of Q on the same clock cycle.

Proper refreshing must guarantee that the duration between two successive refreshes of any line will not exceed the *data retention time*, denoted by N_{DRT}, and measured in clock cycles. 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 $\mathrm{R/W}$ access sequences can be arbitrary, there may be insufficient simultaneous refreshing opportunities to fulfil the N_{DRT} data retention time period constraint. This requires a supplementary mechanism to ensure that no matter which $\mathrm{R/W}$ 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.

Let N_{RR} denote the *refreshing round* period (in clock cycles). It must ensure that for any $\mathrm{R/W}$ access pattern all the M L_{M} rows are refreshed properly. Since the refreshing of a row requires first 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 the refreshing simultaneously with $\mathrm{R/W}$ access. Fig. 2a illustrates two successive refreshing rounds. 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, $0 \leq l \leq L_{\mathrm{M}} - 1$, has been refreshed in the first refreshing round, and let $t''(l)$ be the time stamp in the successive one. Proper refreshing must satisfy $t''(l) - t'(l) \leq N_{\mathrm{DRT}}$.

The worst refreshing case occurs when M is accessed by the CPU only for reads. Since no row can be read into Q,

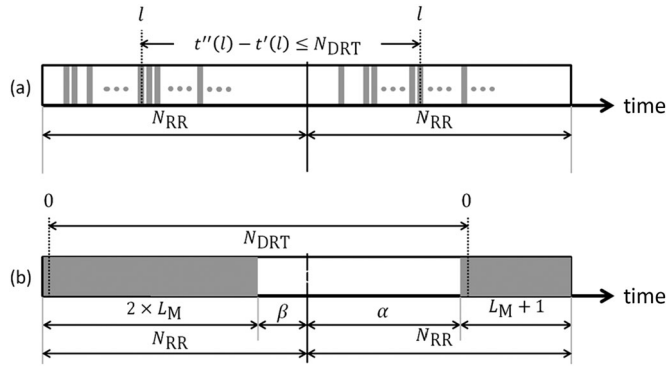

Fig. 2. Derivation of maximal refreshing round period.

$L_M + 1$ cycles will be required to enforce refreshing. In this case, the program is stalled and each cycle writes back a line into the M row and the next row is read into Q, except the first cycle when only a line is read into Q.

The larger the N_{RR}, the greater the likelihood that a higher portion of the L_M refreshing will be concealed by simultaneous opportunistic refreshing, and hence a smaller portion (if at all) will be required to enforce completion. Thus, the goal is to maximize N_{RR}. The worst scenario consisting of two successive N_{RR} periods is shown in Fig. 2b. This occurs for the first M's row when there are $2 \times L_M$ contiguous opportunistic refreshing cycles at the beginning of the previous round, whereas in the subsequent round 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. The worst refreshing scenario in Fig. 2b comprises two unknowns α and β, which need to be maximized to obtain the longest refreshing round period N_{RR}. The largest α and β must satisfy two constraints. The first relates to the longest duration between two successive refreshes of line zero, which for the maximization of N_{RR} should be N_{DRT}

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

The second constraint ensures that the two successive refreshing rounds have the same length N_{RR}

$$2 \times L_M + \beta = \alpha + L_M + 1 = N_{RR}. \quad (2)$$

Solving (1) and (2) for N_{RR} yields

$$N_{RR} = \left\lfloor \frac{N_{DRT} + L_M + 1}{2} \right\rfloor. \quad (3)$$

Note that an addition of just a single cycle to N_{RR} in (3) will violate proper refreshing if the worst-case scenario shown in Fig. 2b occurs. Hence (3) is indeed the maximal N_{RR}.

3 Implementation of Simultaneous Opportunistic Refreshing

The simultaneous opportunistic refreshing algorithm requires three down-counters. The first counter is c_{RR}, $N_{RR} - 1 \ge c_{RR} \ge 0$ that counts the refreshing round cycle-by-cycle. The second counter is c_{MR}, $L_M - 1 \ge c_{MR} \ge 0$ that determines which M row should be read next into Q. The third counter is c_{MW}, $L_M - 1 \ge c_{MW} \ge 0$ that determines which M row should next be written back from Q. The three

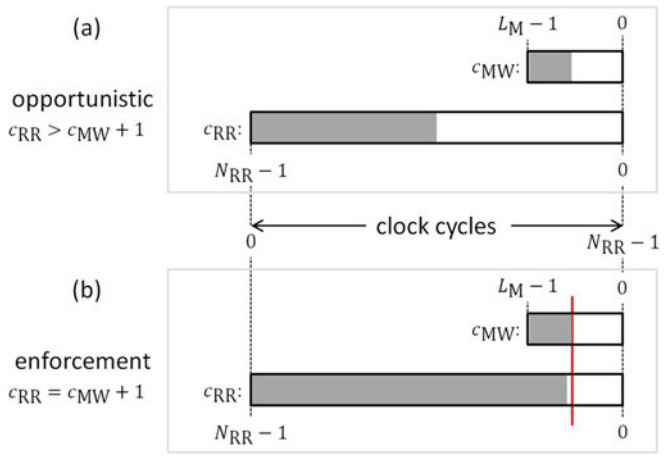

Fig. 3. Refreshing modes: (a) Opportunistic, (b) enforcement of refreshing completion.

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}. Let L_Q be the capacity of Q. For any time $N_{RR} - 1 \ge t \ge 0$ of a refreshing round (measured in clock cycles) there is

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

where $Q(t)$ is the size of Q at time t.

Refreshing takes place simultaneously with CPU accesses to M. The counter c_{RR} is unconditionally decremented at every clock cycle $N_{RR} - 1 \ge t \ge 0$. Upon a W-cycle, an M row is read into the Q tail and c_{MR} is decremented, whereas c_{MW} is unchanged. Upon an 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, simultaneous opportunistic refreshing proceeds as shown in Fig. 3a. Once $c_{RR} = 0$, the M refreshing round is completed and the counters are set to their initial values $c_{RR} = N_{RR} - 1$ and $c_{MR} = c_{MW} = L_M - 1$.

If it occurs at some cycle τ, $N_{RR} - 1 > \tau > 0$, that $c_{RR}(\tau) = c_{MW}(\tau) + 1$, as shown in Fig. 3b, refresh completion must be enforced since otherwise there will not be enough cycles to write back and complete the refreshing within N_{RR} cycles. Enforced refreshing stalls the CPU access for $c_{MW}(\tau) + 1$ cycles by disabling the memory bus in Fig. 1. 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 Q and a line is written back from Q 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 $c_{RR} = N_{RR} - 1$ and $c_{MR} = c_{MW} = L_M - 1$. Note that c_{MR} is never 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 $Q(\tau) > 0$.

The left side of Fig. 1 depicts the hardware implementation of the refreshing algorithm. It comprises the M's refreshing counters c_{MR} and c_{MW}, whose role is to respectively generate the read and write addresses of the current lines to be refreshed. There is also the refreshing round counter c_{RR},

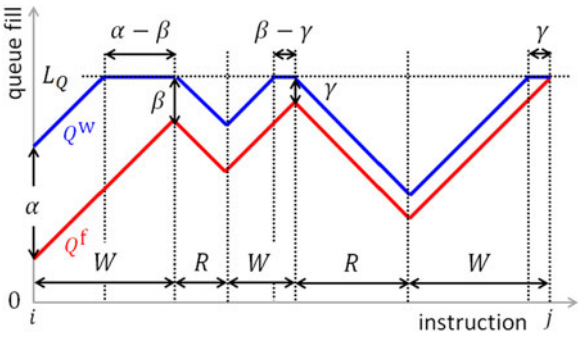

Fig. 4. Filling state of queues for R/W sequence starting with W-cycles.

with control logic to enforce refreshing completion if the condition depicted in Fig. 3b is met. In an M R-cycle, data is read into the CPU through the lower-right de-MUX, whereas data from the refreshing buffer is written back into M via the upper-right MUX. The CPU read and write addresses are obtained by the upper-left and lower-left MUXes, respectively. An M W-cycle works symmetrically. Compliance with the refreshing completion condition disables the CPU access (some details are ignored) so at every cycle of the rest of the refreshing round an M line is read into the tail of Q and its head line is written back into M.

Crucially, the replacement of the SRAM memory by GCeDRAM by opportunistic refreshing is completely transparent to the underlying running code and no special machine instructions are required. Rather, existing memory request/grant signals are used for the handshake of the CPU, GCeDRAM and the refreshing controller. The various read/write control signals on Fig. 1 were derived and determined by the memory request/grant signals.

4 Saturated and Empty Refreshing Queue

Below we use CPU and the term 'program' interchangeably. For long sequences comprising extensive W-cycles Q can become saturated. This occurs when $Q(t) = L_Q$ and a W-cycle occurs at $t+1$, causing the refreshing attempts to read into Q. In this case it is possible to stall program execution and enforce Q flushing and writing its L_Q lines back into M. We denote the flushing queue by Q^{f}. Once Q^{f} is empty, the pending program resumes its execution. Let PC^{f} and $c^{\mathrm{f}}_{\mathrm{MW}}$ denote the program counter and the refreshing write back counter, and let Q^{f} be saturated at clock cycle t, then

$$PC^{\mathrm{f}}(t+L_Q) = PC^{\mathrm{f}}(t), \tag{5}$$

$$c^{\mathrm{f}}_{\mathrm{MW}}(t+L_Q) = c^{\mathrm{f}}_{\mathrm{MW}}(t) + L_Q. \tag{6}$$

An alternative saturation treatment policy is not to do anything and simply let the program continue and the refreshing wait until the next CPU R-cycle occurs, enabling Q to write back into M. We denote the waiting queue by Q^{w}. Let PC^{w} denote the program counter and $c^{\mathrm{w}}_{\mathrm{MW}}$ the refreshing write back counter. Let Q^{w} be saturated at clock cycle t and $n_{\mathrm{W}} \geq 0$ be the number of contiguous W-cycles occurring immediately after Q^{w} is saturated, then

$$PC^{\mathrm{w}}(t+n_{\mathrm{W}}) = PC^{\mathrm{w}}(t) + n_{\mathrm{W}}, \tag{7}$$

$$c^{\mathrm{w}}_{\mathrm{MW}}(t+n_{\mathrm{W}}) = c^{\mathrm{w}}_{\mathrm{MW}}(t). \tag{8}$$

Whereas Q^{f} in (5) and (6) stalls the program execution but progresses the refreshing, Q^{w} in (7) and (8) progresses the program execution but stalls the refreshing.

Let a program be comprised of a sequence of N memory R/W access instructions, and let $Q^{\mathrm{f}}(i)$ and $Q^{\mathrm{w}}(i)$, $1 \leq i \leq N$ be the respective queue states after instruction i is executed; namely, $PC^{\mathrm{f}} = PC^{\mathrm{w}} = i$. We define $Q^{\mathrm{f}}(0) = Q^{\mathrm{w}}(0) = 0$. When the program starts, Q^{f} and Q^{w} behave similarly until at some instruction $j < N$ there is $Q^{\mathrm{f}}(j) = Q^{\mathrm{w}}(j) = L_Q$. If such j does not exist, Q^{f} and Q^{w} behave similarly during the entire program, and hence assume that such a j does exist. It is important to note that $Q^{\mathrm{w}}(i) \geq Q^{\mathrm{f}}(i)$, $1 \leq i \leq N$.

Fig. 4 illustrates Q^{f} and Q^{w} for a possible R/W sequence. It begins with $Q^{\mathrm{w}}(i) - Q^{\mathrm{f}}(i) = \alpha > 0$, and ends when Q^{f} saturates at instruction j. Note that the excessive filling α of $Q^{\mathrm{w}}(i)$ is exhausted by the cycles when Q^{w} is saturated, since it is unable to read from M, and hence the refreshing stalls. The exhaustion of the excessive filling α follows from the telescopic sum of Q^{w} saturation periods. Once Q^{f} saturates at $PC^{\mathrm{f}} = j$, PC^{f} stalls for L_Q cycles as expressed in (5) while Q^{f} proceeds with refreshing by flushing, as expressed in (6). For the example in Fig. 4 there is $(\alpha - \beta) + (\beta - \gamma) + \gamma = \alpha$.

An opposite, somewhat symmetric situation occurs in Fig. 5 when Q^{w} is emptied. The excessive filling $Q^{\mathrm{w}}(i) - Q^{\mathrm{f}}(i) = \alpha$ is exhausted by the cycles when Q^{f} is empty and Q^{w} is not. The exhaustion of the excessive filling α follows from the telescopic sum of Q^{f} emptiness periods. While Q^{f} stops writing back into M and refreshing stalls, Q^{w} continues with refreshing. Once Q^{w} is emptied at $PC^{\mathrm{w}} = j$, it also stalls refreshing. As long as the program is reading from M, both queues stay empty.

As noted above, the program and refreshing progression may come on the expense of each other. Since at saturation and emptiness Q^{f} and Q^{w} behave in opposite ways with respect to the refreshing and program execution progression, this becomes a question of which one is preferable. The following theorem proves that Q^{w} is favored over Q^{w}. The proof is found in appendix A.

Theorem: *Let a program perform either a memory read or a memory write at each clock cycle. Let Q be a refreshing queue of finite capacity as shown in Fig. 1, reading in a line at a memory write and writing back a line at a memory read. Let Q do nothing when it saturates, but rather wait for the next memory read to get out of saturation. This queuing policy minimizes the performance loss caused by stalling program execution to enforce refreshing completion.*

5 Optimal Memory Sizing

The cache memories of microprocessors are too large to be monolithic, and hence are commonly divided into banks, each of which is a self-contained refreshable unit (Fig. 2 in [9]). Although the total cache capacity L_{cache} is defined by the system architecture, its division into refreshable units of L_{M} size each is a matter of design optimization. A cache thus consists of $L_{\mathrm{cache}}/L_{\mathrm{M}}$ units, each of which involves a non-negligible refreshing hardware overhead. Fig. 1 shows that M involves Q and counters. While 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.

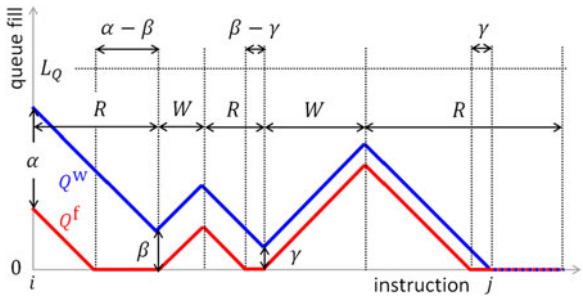

Fig. 5. Filling state of queues for R/W sequence starting with R-cycles.

Maximization of L_{M} will minimize $L_{\mathrm{cache}}/L_{\mathrm{M}}$ and hence the total refreshing 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. 2), and refreshing completion which stalls the system will be enforced. This causes performance degradation that needs to be minimized; hence, L_{M} should be small enough. Another interesting issue is how Q capacity L_Q affects the performance loss due to insufficient opportunistic refreshing. Large L_Q will apparently yield more opportunistic refreshing, and hence higher system performance. Large L_Q, however, represents a hardware overhead.

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

Performance loss does not occur if the 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 $2 \times L_{\mathrm{M}}$ clock cycles. Reading the M line 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. 6 illustrates the possible states of Q filling $\{E, 0, 1, \ldots, L_Q - 1, L_Q, S\}$ and their transition probabilities upon R and W cycles. This is a special case of a finite capacity queue; each of the emptied and filled states encircled in Fig. 6 are further split into two sub-states, where for the grayed sub-state performance is lost and for the other it is not. Every state $1 \leq i \leq L_Q - 1$ passes into state $i + 1$ with probability λ and to state $i - 1$ with probability μ.

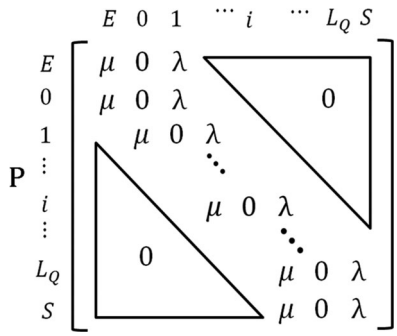

Fig. 7. Refreshing queue filling state transition probability matrix.

An R-cycle at state 0 causes queue emptying cycle loss, designated by state E. Subsequent R-cycles leave Q in state E. A W-cycle occurring at state E passes Q directly to state 1. A symmetric situation occurs at state L_Q, where a W-cycle causes queue filling cycle loss, designated by state S. Subsequent W-cycles will result in queue filling cycle loss, leaving Q in state S. Memory R-cycle occurring at 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 $\mathbf{P} = (P_{ij})$, $i, j \in \{E, 0, \ldots, L_Q, S\}$ illustrated in Fig. 7, 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 $0 \leq i \leq L_Q$, 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 limiting 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 [15]

$$p_j = \sum_{i \in \{E,0,\cdots,L_Q,S\}} p_i P_{ij}, j \in \{E, 0, \ldots, L_Q, S\}, \tag{9}$$

together with

$$\sum_{i \in \{E,0,\cdots,L_Q,S\}} p_i = 1. \tag{10}$$

Defining $\rho \triangleq \lambda/\mu$, the solution of (9) and (10) yields the probabilities of emptied and saturated Q by the algebraic manipulations elaborated in Appendix B as follows

$$p_E = \frac{1-\rho}{(1+\rho)\left(1-\rho^{L_Q+1}\right)}, \tag{11}$$

and

$$p_S = \frac{(1-\rho)\rho^{L_Q+1}}{(1+\rho)\left(1-\rho^{L_Q+1}\right)}. \tag{12}$$

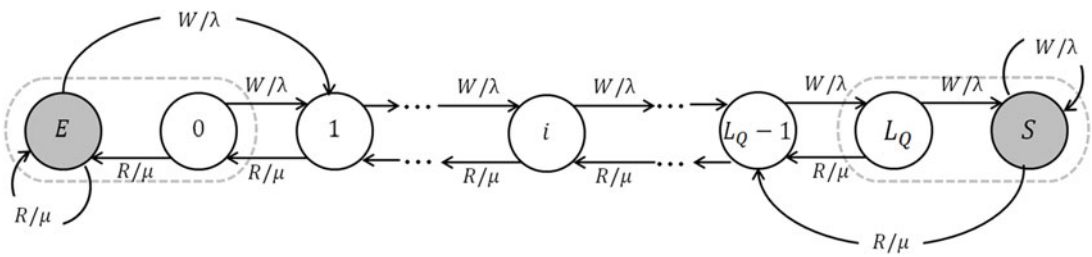

Fig. 6. State transitions of refreshing queue filling.

The expressions of p_E in (11) and p_S in (12) are not defined for $\rho \triangleq \lambda/\mu = 1$. However, taking their limits for $\rho \to 1$ and applying L'Hopital'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 therefore

$$1 - (p_E + p_S) = \frac{2\rho\left(1-\rho^{L_Q}\right)}{(1+\rho)\left(1-\rho^{L_Q+1}\right)}. \tag{13}$$

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

$$(1 - p_E - p_S)N_{\mathrm{RR}} \ge 2L_{\mathrm{M}} + \varepsilon, \tag{14}$$

there will surely be enough simultaneous opportunistic refreshing cycles, so refreshing completion enforcement will not be required, and no performance loss occurs. If

$$(1 - p_E - p_S)N_{\mathrm{RR}} < 2L_{\mathrm{M}} + \varepsilon, \tag{15}$$

the refreshing deficit $2L_{\mathrm{M}} + \varepsilon - (1 - p_E - p_S)N_{\mathrm{RR}}$ 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 are performed (CPU access in Fig. 1 is disabled), we get

$$N_{\mathrm{comp}} = L_{\mathrm{M}} + \frac{1}{2}\varepsilon - \frac{1}{2}(1 - p_{\mathrm{E}} - p_{\mathrm{S}})\left(N_{\mathrm{RR}} - N_{\mathrm{comp}}\right). \tag{16}$$

Substitution of (13) in (16), N_{comp} is solved to

$$N_{\mathrm{comp}} = \frac{\left(L_{\mathrm{M}} + \frac{1}{2}\varepsilon\right)(1+\rho)\left(1-\rho^{L_Q+1}\right) - N_{\mathrm{RR}}\rho\left(1-\rho^{L_Q}\right)}{1-\rho^{L_Q+2}}. \tag{17}$$

Thus, from (13), (14) and (17) it can be concluded that the performance loss $0 \le \gamma = N_{\mathrm{comp}}/N_{\mathrm{RR}} \le 1$ occurred by refreshing completion enforcement is

$$\gamma = \begin{cases} 0 & \text{if } \frac{2\rho\left(1-\rho^{L_Q}\right)}{(1+\rho)\left(1-\rho^{L_Q+1}\right)} N_{\mathrm{RR}} \ge 2L_{\mathrm{M}} + \varepsilon \\ \frac{\left(L_{\mathrm{M}}+\frac{1}{2}\varepsilon\right)(1+\rho)\left(1-\rho^{L_Q+1}\right) - N_{\mathrm{RR}}\rho\left(1-\rho^{L_Q}\right)}{N_{\mathrm{RR}}\left(1-\rho^{L_Q+2}\right)}, & \text{otherwise.} \end{cases} \tag{18}$$

Equation (18) 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. Note that since $L_Q \ll L_{\mathrm{M}}$, ε can in practical terms be neglected.

Fig. 8a plots the CPU access performance $1 - \gamma$ as obtained from (18) for $L_Q = 2$ 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 L_{M}, the larger $1 - \gamma$. This trend is shown clearly. When $N_{\mathrm{RR}} = L_{\mathrm{M}}$ refreshing must take place at 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.

The amount of opportunistic refreshing also depends on the memory write and read probabilities, $1 \ge \lambda \ge 0$ and $\mu = 1 - \lambda$, respectively. For a very small λ Q will often be empty, so 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 M lines into Q upon W-cycles. These phenomena are illustrated by the decline in performance of $1 - \gamma$ towards the surface margins at $\mu = 1$ and $\mu = 0$.

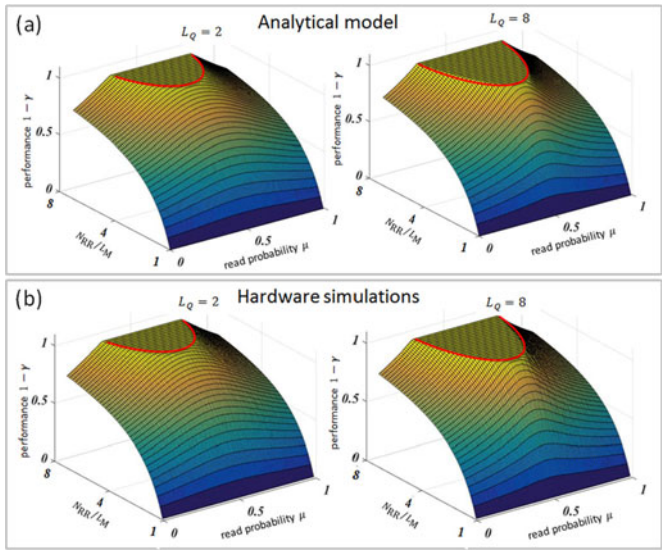

Fig. 8. Memory access performance by analysis in (a) and simulation of hardware (b).

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 curves in Fig. 8a. 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 (18), the design in Fig. 1 was implemented in hardware and integrated into a fully featured RISC-V based ultra-low power processor called PULPino [10] to replace its SRAM data memory. The HDL Veilog code was synthesized to gate-level and then simulated with a Mentor Graphics ModelSim simulator. We used random R/W traces for $L_Q = 2$ and $L_Q = 8$, and various $0 \le \mu \le 1$ and $N_{\mathrm{RR}}/L_{\mathrm{M}}$ ratios. The simulated performance surfaces illustrated in Fig. 8b are perfectly match with the analytical ones in Fig. 8a.

As expected, Fig. 8 shows that larger L_Q yields higher performance. To grasp the impact of L_Q, the boundary curve of zero performance loss in Fig. 8 can be derived from (18) for any choice of L_Q, yielding the parametric equation depicted in Fig. 9 for $1 \le L_Q \le 10$. It is clearly shown that the increase of L_Q beyond a certain point hardly increases the zero performance loss area. It can be deduced easily from (18) that $\lim_{L_Q\to\infty} \partial\gamma/\partial L_Q = 0$.

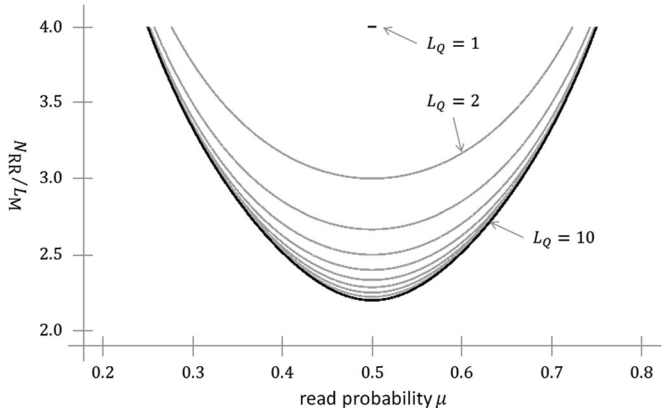

Fig. 9. Zero performance loss dependence on queue capacity.

6 Generalization of Memory Access

So far it has been assumed that the CPU either reads from or writes back into the memory at each clock cycle. Recall that the GCeDRAM bit cell has separate read and write ports, enabling the CPU to simultaneously read from and writes into the memory (different lines). Moreover, programs may have cycles where the CPU does not access the memory. Processors can therefore have all four memory access cycle types: R, W, R + W and idle. The respective opportunistic refreshing operations thus are writing back from Q into M, reading from M into Q, do not refresh, and finally, both reading M into Q and writing back from Q into M (different lines). The new memory access modes do not affect the states of Q, which does not change its filling size in the case of no refreshing or when reading into and writing back from Q take place on same cycle. Therefore, the conclusion regarding the optimality of Q^w holds for general memory access.

The new memory access modes affect the amount of opportunistic refreshing and the progression of c_{MR} and c_{MW} counters, and thus impact the performance loss due to the program stalls required to complete refreshing. If memory access idleness is more frequent than R + W cycles, c_{MR} and c_{MW} progress faster, thus increasing performance. When memory access idleness is less frequent than R + W, c_{MR} and c_{MW} progression is slower, thus decreasing performance. The state transition diagram in Fig. 8 and the corresponding transition probabilities matrix in Fig. 7 can be extended to describe the general memory access model.

7 Experimental Results

As depicted in Fig. 8 the hardware simulations matched the analytic model perfectly under random R/W traces. In order to show the advantage of queuing-based GCeDRAM opportunistic refreshing, we employed a test bench comprising ten real applications. Their corresponding RISC-V machine instruction code was simulated with the fully featured PULPino processor [10] and the results are shown in Table 1. Its left side includes the lengths of the code and the percentage of the CPU memory reads and writes. The right side shows the speedup in run-time compared to using ordinary periodic refreshing achievable by using our queuing-based opportunistic refreshing algorithm. As mentioned in Section 6, the opportunistic refreshing also used idle cycles.

TABLE 1
Speedup by Queuing-Based Opportunistic Refreshing

Test				N_{RR}/L_M					
Name	Total Instructions	CPU Read [%]	CPU Write [%]	Queue size	1.5	1.4	1.3	1.2	1.1
2D convolution	85978	32.52	1.82	1	1.25	1.74	1.84	2.05	2.66
				8	1.25	1.77	1.88	2.11	2.79
FastDCT	68009	28.54	2.71	1	1.27	1.38	1.34	1.51	1.93
				8	1.28	1.41	1.40	1.61	2.11
FIR filter	162102	24.06	1.21	1	1.86	2.78	3.20	3.38	3.38
				8	1.86	2.87	3.33	3.53	3.61
FFT	274574	19.61	9.67	1	1.64	2.49	2.57	2.63	2.60
				8	2.09	3.55	4.06	4.46	4.40
SHA-3 (crypto hash)	797858	20.00	13.02	1	1.82	2.96	2.92	2.84	2.45
				8	2.32	4.12	4.38	4.40	3.62
Fibonacci	24136	30.96	4.91	1	1.43	1.84	1.97	1.99	2.13
				8	1.43	1.87	2.03	2.17	2.19
Bubble sort	121315	37.39	16.77	1	1.42	2.56	2.22	2.18	2.03
				8	1.93	3.17	3.11	3.14	2.98
Dot-product	24482	29.72	2.41	1	1.65	2.05	2.08	2.24	2.60
				8	1.67	2.08	2.13	2.31	2.74
Matrix multiplication	38165	34.11	2.87	1	1.80	2.60	2.60	2.65	3.06
				8	1.81	2.61	2.60	2.68	3.11
CRC32	1461929	9.38	0.04	1	2.91	5.77	7.36	10.52	11.29
				8	2.91	5.77	7.36	10.52	11.30

The full GCeDRAM advantage stems from its very low operation voltage (below 700 mV) and very small area (2T gain cell). This dictates a short DRT [16], which from (3) results in a short refreshing round period N_{RR} and a small size of the refreshable unit L_M. The PULPino architecture features 32 KBytes of L1 data memory, for which we used 16 refreshable units (banks) of 2 KBytes each, similar to the one recently designed and manufactured in [17]. We considered N_{RR}/L_M in the range of 1.5 to 1.1 as dictated by the above parameters. Run-time speedup from 1.25 to 11.30 was achieved.

It is important to note that the performance improvement compared to ordinary periodic refreshing considerably increases with the decrease in N_{RR}/L_M. This may be a crucial factor in ultra-low power designs using very low operation voltage (500 mV). There, the DRT may be subject to wide variability, yielding worst-case bit cells DRT of very few microseconds [17]. To avoid refreshing failures, it is possible to configure the counter c_{RR} at silicon testing, so N_{RR} is set appropriately.

To highlight the impact of the refreshing queue capacity, Table 2 summarizes the performance speedup obtained by $L_Q = 8$ compared to $L_Q = 1$. While for some of the tests the improvement was just a few percent, in some tests it reached 70 precent. This is explained by the relative amounts of reads and writes. Large queue is mostly effective in 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, unchanging Q filling, hence L_Q does not matter.

The impact L_Q is greater when N_{RR}/L_M is small, a case that stresses refreshing constraints. The table shows that for tests comprising higher CPU write rates such as FFT, SHA-3 and bubble sort, a larger queue is more effective than a smaller one. This can be explained by the fact that extensive

TABLE 2
Impact of Refreshing Queue Capacity

N_{RR}/L_M	2D convolution	Fast DCT	FIR filter	FFT	SHA-3 (crypto hash)	Fibonacci	Bubble sort	Dot-product	Matrix multiplication	CRC32
1.1	1.05	1.09	1.07	1.69	1.48	1.13	1.46	1.05	1.02	1.00
1.2	1.03	1.06	1.05	1.70	1.54	1.09	1.44	1.03	1.01	1.00
1.3	1.02	1.04	1.04	1.58	1.50	1.07	1.40	1.02	1.00	1.00
1.4	1.01	1.02	1.04	1.42	1.39	1.02	1.39	1.02	1.00	1.00
1.5	1.00	1.01	1.00	1.12	1.29	1.0	1.36	1.01	1.01	1.00

CPU writes enable sufficient queue filling which is then emptied at CPU reads.

8 Conclusion

This article showed how a great deal of the blockages to memory access by the CPU incurred from memory refreshing can be avoided by applying an opportunistic refreshing algorithm. An essential component of this algorithm is a queue, whose impact on system performance was examined by using a stochastic model. The interrelation between memory size, the probability of the CPU read/write memory access and the queue capacity was derived in a closed-form expression. This expression is conclusive and can be implemented to optimally architect memories for ultra-low power processors such that their performance loss due to refreshing is minimized.

Appendix A: Proof of Q^w Favor Over Q^f

Proof. We first show the superiority of Q^{w} over Q^{f}. Consider a program executing N_{RR} instructions, each of which is either a memory read or a memory write. Let us divide the program progression by the instructions j_k, $0 < j_k < N_{\mathrm{RR}} - 1$, $1 \le k \le m$, for which $Q^{\mathrm{f}}(j_k) = L_Q$. It was mentioned above that if this never occurs, then Q^{f} and Q^{w} behave similarly for the entire program, so one is not favored over the other. □

The instructions j_k divide PC into $m+1$ intervals $i_k \le i \le j_k$, where $i_1 = 0$, $i_{k+1} = j_k + 1$ and $j_{m+1} = N_{\mathrm{RR}} - 1$. At the beginning of interval k, $2 \le k \le m+1$, Q^{f} is empty after it became saturated at instruction j_{k-1}. The behavior of Q^{f} and Q^{w} at the beginning of interval k, where the interval starts with R-cycles in (a) and W-cycles in (b), is illustrated in Fig. 10. During $i_k \le i \le j_k$ Q^{f} and Q^{w} may comprise segments types as in Figs. 4 and 5. To decide whether Q^{f} or Q^{w} is better, let us consider the progression of the program counters $PC^{\mathrm{f}}(t)$ and $PC^{\mathrm{w}}(t)$ with respect to the time, and the amount of reads into and writes from Q^{f} and Q^{w} in the instruction intervals $1 \le k \le m+1$.

To this end we first modify Q^{w} into Q^{w^*}. We will later modify Q^{w^*} back into Q^{w}, and prove that they perform identically regarding the progression in program execution and the refreshing completion requirement with respect to the complete refreshing round period N_{RR}. At instruction j_k

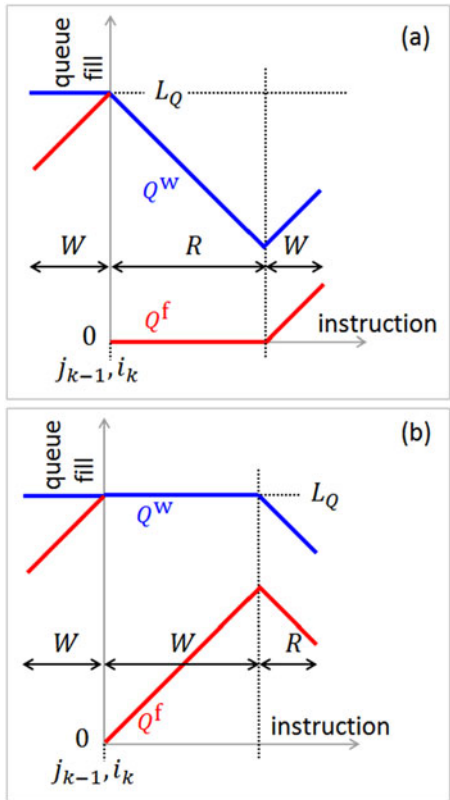

Fig. 10. Behavior of the "flush" and "wait" queues.

Q^{w^*} will stall the program for L_Q cycles to enforce refreshing M by writing back the line at the head of Q^{w^*} into M, and reading a line from M into the tail of Q^{w^*}. In this manner Q^{w^*} will stay saturated but the refreshing of M will progress by L_Q lines. If $t(i)$ denotes the time elapsed when instruction i has been executed, then

$$PC^{\mathrm{w}^*}\big(t(j_k) + L_Q\big) = PC^{\mathrm{w}^*}(t(j_k)), \tag{19}$$

$$c^{\mathrm{w}^*}_{\mathrm{MW}}\big(t(j_k) + L_Q\big) = c^{\mathrm{w}^*}_{\mathrm{MW}}(t(j_k)) + L_Q. \tag{20}$$

Fig. 11 shows the possible behaviors of Q^{w^*} during the flush of Q^{f}. The instructions executed by PC^{f} and PC^{w^*} progress identically, and hence are synchronized with respect to time. Recalling that $i_k = j_{k-1} + 1$ (see Fig. 10), in (a) M is accessed for writes after Q^{f} is flushed, whereas in (b) it is accessed for reads.

Let Λ^{f}_k and $\mathrm{M}^{\mathrm{f}}_k$, $1 \le k \le m$, be the amount of reads from M into Q^{f}(queue filling) and the amount of writes back from Q^{f} into M (queue emptying), respectively. $\Lambda^{\mathrm{w}^*}_k$ and $\mathrm{M}^{\mathrm{w}^*}_k$ are defined similarly for Q^{w^*}. For the reads from M into Q^{f} and Q^{w^*} there is

$$\begin{aligned} &\Lambda^{\mathrm{f}}_1 = \Lambda^{\mathrm{w}^*}_1 \\ &\Lambda^{\mathrm{f}}_k = \Lambda^{\mathrm{w}^*}_k, 2 \le k \le m. \\ &\Lambda^{\mathrm{f}}_{m+1} \le \Lambda^{\mathrm{w}^*}_{m+1} \end{aligned} \tag{21}$$

During the period when both PC^{f} and PC^{w^*} stall in the intervals $2 \le k \le m$, Q^{w^*} keeps filling whereas Q^{f} simply drains to empty. The reads from M into Q^{w^*} thus lead over Q^{f} by L_Q. This lead however nullifies gradually during the

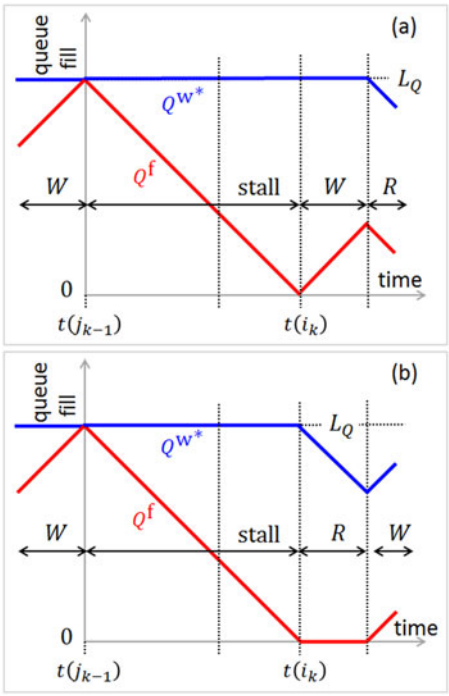

Fig. 11. Behavior of "flush" and "modified wait" queues: writes after flush (a) and reads after flush (b).

rest of the interval, starting when both PC^{f} and PC^{w^*} resume and ending when Q^{f} is saturated again, a time where Q^{w^*} is saturated as well. The lead nullification is shown in Fig. 4 by the telescopic sum of the time intervals where Q^{f} keeps filling and Q^{w^*} is saturated. The last interval $m+1$ may end before Q^{f} becomes saturated; hence some of the reads lead into Q^{w^*} over Q^{f} remains. For the writes from Q^{f} and Q^{w^*} back into M there is

$$\begin{aligned} &\mathrm{M}_1^{\mathrm{f}} = \mathrm{M}_1^{\mathrm{w}^*} \\ &\mathrm{M}_k^{\mathrm{f}} \le \mathrm{M}_k^{\mathrm{w}^*}, 2 \le k \le m+1. \end{aligned} \tag{22}$$

The inequality in the intervals $2 \le k \le m+1$ follows from the fact Q^{w^*} never gets empty before Q^{f} does, whereas they always leave emptiness together. Hence, the periods of Q^{f} emptiness contain the periods of Q^{w^*} emptiness. Since during an emptiness period the queue cannot write back into M, $\mathrm{M}_k^{\mathrm{f}} \le \mathrm{M}_k^{\mathrm{w}^*}$ follows. Summation of (21) and (22) yields respectively

$$\sum_{k=1}^{m+1} \Lambda_k^{\mathrm{f}} \le \sum_{k=1}^{m+1} \Lambda_k^{\mathrm{w}^*}, \tag{23}$$

and

$$\sum_{k=1}^{m+1} \mathrm{M}_k^{\mathrm{f}} \le \sum_{k=1}^{m+1} \mathrm{M}_k^{\mathrm{w}^*}. \tag{24}$$

While PC^{f} and PC^{w^*} progress identically, namely $PC^{\mathrm{f}}(t) = PC^{\mathrm{w}^*}(t)$, $0 \le t \le N_{\mathrm{RR}} - 1$ (see Fig. 11), it follows from (23) and (24). that the refreshing with Q^{w^*} is never behind the refreshing with Q^{f}, but rather may be ahead.

Recall from Fig. 3 that the refreshing algorithm maintains a down-counter c_{MW} initialized to $c_{\mathrm{MW}}(0) = L_{\mathrm{M}} - 1$, which counts the M lines that still need refreshing before N_{RR} cycles elapse. Another down-counter c_{RR} initialized to $c_{\mathrm{RR}}(0) = N_{\mathrm{RR}} - 1$, counts the remaining clock cycles until N_{RR} cycles elapse. If at some $\tau \le N_{\mathrm{RR}} - 1$ $c_{\mathrm{RR}}(\tau) = c_{\mathrm{MW}}(\tau) + 1$, the program stalls for $c_{\mathrm{MW}}(\tau) + 1$ cycles to enforce refreshing completion of the remaining $c_{\mathrm{MW}}(\tau)$ M lines. At each clock cycle an M line is read into the Q tail and the Q head is written back into M.

It follows from (23) and (24) that at any t before refreshing completion is enforced there is $c_{\mathrm{MW}}^{\mathrm{w}^*}(t) \le c_{\mathrm{MW}}^{\mathrm{f}}(t)$. If for both counters refreshing completion is enforced, there are two times τ^{f} and τ^{w^*} such that $c_{\mathrm{RR}}(\tau^{\mathrm{f}}) = c_{\mathrm{MW}}^{\mathrm{f}}(\tau^{\mathrm{f}}) + 1$, and $c_{\mathrm{RR}}(\tau^{\mathrm{w}^*}) = c_{\mathrm{MW}}^{\mathrm{w}^*}(\tau^{\mathrm{w}^*}) + 1$. Since c_{RR} is an absolute time counter, independent of the executed program, there is $\tau^{\mathrm{w}^*} \ge \tau^{\mathrm{f}}$. Consequently, within N_{RR} period PC^{w^*} will perform $\tau^{\mathrm{w}^*} - \tau^{\mathrm{f}} \ge 0$ more instructions than PC^{f}. Returning to Fig. 11, the refreshing maintained by Q^{w^*} during L_Q cycles at instruction j_{k-1}, $2 \le k \le m+1$ does not change its state (stays saturated), so it can be postponed to j_{m+1} without affecting proper refreshing completion. These postponements make Q^{w^*} behave exactly like Q^{w}, thus proving that Q^{w} performs better than Q^{f}.

Any partial flushing which writes back only l entries, $0 < l < L_Q$ to the M, is also inferior to Q^{w}. This is shown by stalling PC^{w^*} for l cycles during which Q^{w^*} reads from and writes into M at every cycle, thus making the progression of PC^{f} and PC^{w^*} synchronized again. Using the same arguments of postponing the l refreshes of Q^{w^*} at each instruction j_{k-1}, $2 \le k \le m+1$, to j_{m+1}, modifies Q^{w^*} to Q^{w}, while proper refreshing completion is preserved.

APPENDIX B: DERIVATION OF p_E AND p_S

The transition matrix in Fig. 7 yields $L_Q + 3$ equations

$$\begin{aligned} p_E &= \mu p_E + \mu p_0, \\ p_0 &= \mu p_1, \\ p_1 &= \lambda p_E + \lambda p_0 + \mu p_2, \\ p_i &= \lambda p_{i-1} + \mu p_{i+1}, && 2 \le i \le L_Q - 2, \\ p_{L_Q-1} &= \lambda p_{L_Q-2} + \mu p_{L_Q} + \mu p_S, \\ p_{L_Q} &= \lambda p_{L_Q-1}, \\ p_S &= \lambda p_{L_Q} + \lambda p_S. \end{aligned} \tag{25}$$

Defining $x_0 \triangleq p_E + p_0$, $x_i \triangleq p_i, 1 \le i \le L_Q - 1$ and $x_{L_Q} \triangleq p_{L_Q} + p_S$, substitution in (25) turns it into the following $L_Q + 1$ equations

$$\begin{aligned} x_0 &= \mu x_0 + \mu x_1 \\ x_i &= \lambda x_{i-1} + \mu x_{i+1}, && 1 \le i \le L_Q - 1, \\ x_{L_Q} &= \lambda x_{L_Q-1} + \lambda x_{L_Q} \end{aligned} \tag{26}$$

where

$$\sum_{i=0}^{L_Q} x_i = 1. \tag{27}$$

Equations (26) and (27) are solved to

$$x_i = x_0 \left(\frac{\lambda}{\mu}\right)^i, \quad 1 \le i \le L_Q. \tag{28}$$

Defining $\rho \triangleq \lambda/\mu$, and substitution of (28) in (27), there is $x_0 \sum_{i=0}^{L_Q} \rho^i = 1$, yielding

$$x_0 = \frac{1-\rho}{1-\rho^{L_Q+1}}. \tag{29}$$

Recalling from (25) that $p_E = \mu p_E + \mu p_0$ and $\mu = 1-\lambda$, there is $p_0 = \rho p_E$. By definition there is $x_0 \triangleq p_E + p_0 = (1+\rho)p_E$, which by substitution of (29) yields

$$p_E = \frac{1-\rho}{(1+\rho)\left(1-\rho^{L_Q+1}\right)}.$$

Using similar substitutions, there is

$$p_S = \frac{(1-\rho)\rho^{L_Q+1}}{(1+\rho)\left(1-\rho^{L_Q+1}\right)}.$$

Acknowledgments

This work was supported by the Israel Ministry of Industry (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 also acknowledge the useful comments by the anonymous reviewers who enabled us to improve the manuscript.

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, Jan. 2008.

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

[3] A. Teman, P. Meinerzhagen, A. Burg, and A. Fish, "Review and classification of gain cell eDRAM implementations," in *Proc. IEEE 27th Convention Elect. Electron. Engineers Israel*, Nov. 2012, pp. 1–5.

[4] 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 Edition Great Lakes Symp. Great Lakes Symp. VLSI*, 2011, pp. 343–346.

[5] J. M. Rabaey, A. P. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits* Ch. 12, 2 ed., Englewood Cliffs, NJ, USA: Prentice hall, 2002.

[6] 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, Jan. 2016.

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

[8] 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.

[9] A. Kazimirsky and S. Wimer, "Opportunistic refreshing algorithm for eDRAM memories," *IEEE Trans. Circuits Syst. I: Regular Papers*, vol. 63, no. 11, pp. 1921–1932, Nov. 2016.

[10] A. Traber, F. Zaruba, S. Stucki, A. Pullini, G. Haugou, E. Flamand, F. K. Gürkaynak, and L. Benini, 2016. [Online]. Available: https://riscv.org/wp-content/uploads/2016/01/Wed1315-PULP-riscv3_noanim.pdf

[11] K. Skadron and D. W. Clark, "Design issues and tradeoffs for write buffers," in *Proc. Third Int. Symp. High-Perform. Comput. Archit.*, 1997, pp. 144–155.

[12] D. Ponomarev, G. Kucuk, and K. Ghose, "Energy-efficient design of the reorder buffer," in *Proc. 12th Int. Workshop Integr. Circuit Design. Power Timing Modeling Optimization Simul.*, 2002, pp. 289–299.

[13] K. Samouylov, E. Sopin, and I. Gudkova, "Sojourn time analysis for processor sharing loss queuing system with service interruptions and MAP arrivals," in *Proc. Int. Conf. Distrib. Comput. Commun. Netw.*, 2016, pp. 406–417.

[14] Q. Zhen and C. Knessl, "On sojourn times in the finite capacity M/M/1 queue with processor sharing," *Operations Res. Lett.*, vol. 37, no. 6, pp. 447–450, 2009.

[15] S. M. Ross, *Introduction to Probability Models*. Oxford, U.K.: Academic Press, 2014.

[16] D. Somasekhar, Y. Ye, P. Aseron, S.-L. Lu, M. M. Khellah, J. Howard, G. Ruhl, T. Karnik, S. Borkar, V. K. De, and K. Ali, "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 Circuits*, vol. 44, no. 1, pp. 174–185, Jan. 2009.

[17] R. Giterman, A. Fish, A. Burg, and A. Teman, "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. Circuits Syst. I: Regular Papers*, vol. pp, no. 99, Sep. 2017.

Binyamin Frankel received the BSc and MSc degrees in electrical engineering from Bar-Ilan University, in 2014 and 2016, respectively. He is currently working toward the PhD degree in computer engineering at the Bar-Ilan University. He is interested in VLSI circuits and systems design optimization.

Roi Herman received the BSc degree (cum laude) in computer engineering from Bar-Ilan University, Israel, in 2015, where he is currently working toward the MSc degree. He was a backend engineer with Intel from 2013 to 2017. He is now with Samsung, where he is currently working as a Logic Design Engineer. His research interests include refresh architecture optimization for eDRAM cache memories, and approximation computing methods for logic design enhancements.

Shmuel Wimer received the BSc and MSc degrees in mathematics from Tel-Aviv University, Israel, in 1978 and 1981, respectively, and the DSc degree in electrical engineering from the Technion-Israel Institute of Technology, Israel, in 1988. From 1978 to 2009 he worked in industry and held R&D, engineering and managerial positions. From 1999 to 2009 he was with Intel, and prior to that with IBM, National Semiconductor, and the IAI-Israeli Aerospace Industry. He is an associate professor in the Engineering Faculty of Bar-Ilan University, Israel and associate visiting professor in the Electrical Engineering Faculty of the Technion. He is interested in VLSI circuits and systems design optimization and combinatorial optimization. He is a member of the IEEE.