Versatile Refresh: Low Complexity Refresh Scheduling for High-throughput Multi-banked eDRAM

Mohammad Alizadeh, Adel Javanmard, Shang-Tse Chuang[†], Sundar Iyer[†], and Yi Lu[‡]
Stanford University [†]Memoir Systems [‡]University of Illinois at Urbana-Champaign
{alizade, adelj}@stanford.edu, {dachuang, sundaes}@memoir-systems.com, yilu4@illinois.edu

ABSTRACT

Multi-banked embedded DRAM (eDRAM) has become increasingly popular in high-performance systems. However, the data retention problem of eDRAM is exacerbated by the larger number of banks and the high-performance environment in which it is deployed: The data retention time of each memory cell decreases while the number of cells to be refreshed increases. For this, multi-bank designs offer a concurrent refresh mode, where idle banks can be refreshed concurrently during read and write operations. However, conventional techniques such as periodically scheduling refreshes—with priority given to refreshes in case of conflicts with reads or writes—have variable performance, increase read latency, and can perform poorly in worst case memory access patterns.

We propose a novel refresh scheduling algorithm that is low-complexity, produces near-optimal throughput with universal guarantees, and is tolerant to bursty memory access patterns. The central idea is to decouple the scheduler into two simple-to-implement modules: one determines which cell to refresh next and the other determines when to force an idle cycle in all banks. We derive necessary and sufficient conditions to guarantee data integrity for all access patterns, with any given number of banks, rows per bank, read/write ports and data retention time. Our analysis shows that there is a tradeoff between refresh overhead and burst tolerance and characterizes this tradeoff precisely. The algorithm is shown to be near-optimal and achieves, for instance, 76.6% reduction in worst-case refresh overhead from the periodic refresh algorithm for a 250MHz eDRAM with $10\mu s$ retention time and 16 banks each with 128 rows. Simulations with Apex-Map synthetic benchmarks and switch lookup table traffic show that VR can almost completely hide the refresh overhead for memory accesses with moderate-to-high multiplexing across memory banks.

Categories and Subject Descriptors: B.3.1 [**Memory Structures**]: Semiconductor Memories—*Dynamic memory (DRAM)*; C.4 [**Performance of Systems**]: Design studies

General Terms: Algorithms, Design, Performance

Keywords: Embedded DRAM, Multi-banked, Memory Refresh Scheduling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

1. INTRODUCTION

Multi-banked embedded DRAM (eDRAM) is widely used in high-performance systems; for example, in packet buffers and control path memories in networking [24], L3 caches in multi-processor cores [8, 1], and internal memories for select communication and consumer applications [19, 23]. Multi-banked eDRAM offers higher performance, lower power and smaller board footprint in comparison to embedded SRAM and discrete memory solutions [21, 9, 10]. For instance, a typical 1Mb SRAM running up to 1GHz occupies 0.5mm^2-1mm^2 in area, and consumes 70-200mW in leakage power on a 45nm process node. Both the low density and high leakage power limit the total amount of SRAM that can be embedded on-chip. In contrast, eDRAMs run at about $1/2$-$1/3^{rd}$ the speed of SRAMs, but are two to three times as dense, and have an order of magnitude lower leakage per bit as compared to SRAMs. For instance, a typical 1Mb eDRAM running up to 500 MHz occupies 0.15mm^2-0.25mm^2 in area, and consumes 6mW-10mW in leakage power on a 45nm process node. Currently, eDRAMs are offered by a number of leading ASIC and foundry vendors, such as IBM, TSMC, ST and NEC.

However, eDRAMs have a memory retention problem: The data in each memory cell wears out over time and needs to be *refreshed* periodically to avoid loss. This is not specific to eDRAMs; the retention issues with discrete DRAMs are also well-known. Traditionally, memory refresh is handled by using an internal counter to periodically refresh all the cells within a particular eDRAM instance. This is done by forcing a refresh operation and giving it highest priority causing conflicting reads or writes to the same bank to be queued. The policy is widespread due to its simplicity. Historically, the penalties due to refresh were low enough to not warrant additional complexity.

However, the memory retention problem is getting worse over time and is exacerbated by the multi-banked design of eDRAMs and the high-performance environments in which they are deployed:

1. Shorter data retention time. The typical data retention time of eDRAM cells at $75°$C-$85°$C is 50μs-200μs, during which every memory cell needs to be refreshed. With high performance applications such as networking, the operating temperature increases to about $125°$C and consequently the data retention time decreases super-linearly to 15μs-100μs. Also, shrinking geometry means less capacitance on an eDRAM bit cell which further lowers the data retention time. As refreshing a memory cell prevents concurrent read or write accesses, shorter data retention time results in more throughput loss.

2. Larger number of memory cells. Multi-banked eDRAMs achieve high density by assembling multiple banks of memory with a common set of I/O and refresh ports. This drastically increases the potential refresh overhead as an order of magnitude more memory cells need to be refreshed sequentially within the same data retention time. In addition, the manufacturing of memory banks with a larger number of rows (for higher density) increases refresh overhead even further.

In this paper we propose a novel refresh scheduling algorithm, Versatile Refresh (VR), which is simple to implement, provides universal guarantees of data integrity and produces near-optimal throughput. VR exploits *concurrent refresh* [11], a capability in multi-banked eDRAM macros that allows an idle bank to be refreshed during a read/write operation in other banks. The central idea in VR is to decouple the scheduler into two separate modules, each of which is simple to implement and complements each other to achieve high-throughput scheduling. The "Tracking" module consists of a small number of pointers and addresses the question *which* cells to refresh next. The "Back-pressure" module consists of a small bitmap and addresses the question *when* to force an idle cycle in all banks by back-pressuring memory accesses. In order to provide worst-case guarantees with a low-complexity algorithm, we ask the question:

Given a fixed number of slots (Y), how many (X) idle cycles in every consecutive Y slots are necessary and sufficient to guarantee data integrity, that is, all memory cells are refreshed in time?

The X idle cycles include those resulting from the workload itself when there are no read/write operations, as well as the idle cycles forced by the scheduler. A larger Y increases the flexibility of placement of the X idle cycles and helps improve the goodput of bursty workloads by deferring refresh operations longer.

Our main contributions are as follows:

(i) We give an explicit expression of the relationship of X and Y to the question above for the VR algorithm in a general setting, that is, for any given number of banks, rows per bank, read/write ports and data retention time. This allows an efficient scheduling algorithm with a worst-case guarantee on the refresh overhead.

(ii) We lower-bound the worst-case refresh overhead of any refresh scheduling algorithm that guarantees data integrity and show that the proposed algorithm is very close to optimal. It achieves, for instance, 76.6% reduction in worst-case refresh overhead compared to the conventional periodic refresh algorithm for a 250MHz eDRAM with $10\mu s$ retention time and 16 banks each with 128 rows.

(iii) The VR algorithm displays a tradeoff between worst-case refresh overhead and burst tolerance: To tolerate larger bursts (allow refreshes to be deferred longer), a higher worst-case refresh overhead is necessary. We analytically characterize this tradeoff and provide guidelines for optimizing the algorithm for different workloads.

(iv) We simulate the algorithm with Apex-Map [17] synthetic benchmarks and switch lookup table traffic. The simulations show that VR has almost no refresh overhead for memory access patterns with moderate-to-high multiplexing across memory banks.

Organization of the paper. We describe the refresh scheduling problem in §2, and present the VR algorithm for a single read/write port in §3 and its analysis in §4. The algorithm and analysis are generalized for multiple read/write ports in §5. We independently corroborate our analytical results using a formal verification property checking tool in §6. Our performance evaluation using simulations is presented in §7. We discuss the related work in §8 and conclude in §9.

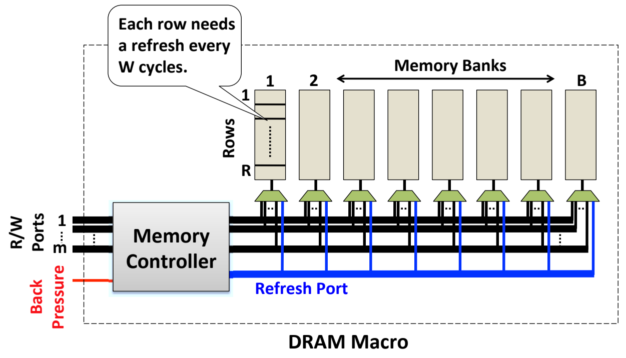


Figure 1: Schematic diagram of a generic eDRAM macro with concurrent refresh capability. Note that in every time slot, each bank can be accessed by only one read/write or refresh port.

2. REFRESH SCHEDULING PROBLEM

In this section we describe the refresh scheduling problem and the main requirements for a refresh scheduling algorithm.

2.1 Setup

We consider a generic eDRAM macro composed of $B \geq 2$ banks, each with R rows,[1] as shown Figure 1. The macro has $1 \leq m < B$ read/write ports and one internal refresh port (typically $m = 1, \ldots, 4$ in practice). A row loses its data if it is not refreshed at least once in every W time slots (W is equal to the retention time, `tREF`, times the clock frequency). In each time slot, the user may read or write data from up to m rows in different banks. All read/write and refresh operations take one time slot.

A bank being accessed by a read/write port is *blocked*, meaning it cannot be refreshed. In this case, the memory controller can refresh a row from a non-blocked bank using the internal refresh port. This is called a *concurrent refresh* operation. Alternatively, the memory controller may choose to *back-pressure* user I/O and give priority to refresh. This forces pending memory accesses to be suspended/queued until the next time slot. Back-pressure effectively results in some loss of memory bandwidth and additional latency when a read operation gets queued. Write latency can sometimes be hidden since writes can typically be cached or posted later, but we do not consider such optimizations in this paper since we are designing for the worst case (all accesses may be reads).

Remark 1. It is important to note that our model is for a typical eDRAM with a SRAM-like interface [20, 2]. Conventional discrete DRAM chips are more complex and have a large number of timing constraints that specify when a command can be legally issued [6]. For example, a refresh operation usually takes more than one clock cycle in DRAMs [18].

2.2 Requirements

There are three main requirements for a refresh scheduling algorithm:

- **Data Integrity:** This is the most important requirement. Refreshes must always be in time so that no data is lost.
- **Efficiency:** The algorithm must make judicious use of back-pressure so that the throughput (and latency) penalty associated with refresh is low.

[1]Memory vendors sometimes bunch multiple physical rows into one 'logical' row which are all refreshed in one operation [14]. In this paper, R refers to the number of 'logical' rows.

- **Low Complexity:** It is not uncommon for large chips to have hundreds of embedded memory instances, all of which will require their own refresh memory controller. Therefore, it is crucial that the algorithm be lightweight in terms of implementation complexity.

Periodic Refresh. In anticipation of our proposed solution, we briefly consider a conventional refresh scheduling algorithm, henceforth referred to as *Periodic Refresh*. The algorithm schedules refreshes to all the rows according to a fixed periodic schedule. Refreshes are done, in round-robin order across banks, every W/RB time slots to ensure that all RB rows are refreshed in time. Since refresh is a blocking operation, memory accesses to that particular bank are back-pressured. Note that concurrent accesses to other idle banks are allowed.

This is the most straight-forward refresh scheduling algorithm and is commonplace due to its very low implementation complexity. However, since all refresh operations can result in back-pressure in case of conflict with memory accesses, the throughput loss with Periodic Refresh is as high as RB/W in the worst case. Historically, this hasn't been much of a concern since the memory bandwidth lost has typically been $\sim$1-3%. However, as previously described (§1), because of decreasing retention time (smaller W) and increasing density (larger B and R), the memory bandwidth loss can be unacceptably large in high-performance eDRAM macros.

3. VERSATILE REFRESH

In this section we describe the Versatile Refresh (VR) algorithm. The VR algorithm provides very high throughput (in fact, it is provably close to optimal) and it has a low implementation complexity.

3.1 Basic Design

The VR algorithm simplifies managing refreshes by decoupling the refresh controller into two separate components: The *Tracking* and the *Back-pressure* modules. The Tracking module keeps track of which row needs to be be refreshed in each time slot. It operates under the assumption that the memory access pattern contains (at least) X idle slots where no bank is being accessed in *any* Y consecutive time slots (X and Y are parameters of the algorithm). This condition is enforced by the Back-pressure module which back-pressures memory accesses whenever necessary.

In the following, we describe the two modules in detail.

3.1.1 Back-pressure module

The Back-pressure module simply consists of a bitmap of length Y and a single counter that sums the 1's in the bitmap. At any time, the bits that are 1 correspond to the idle slots and the counter provides the total number of idle slots in the last Y slots. The Back-pressure module guarantees that there are at least X idle slots in any consecutive Y slots by back-pressuring pending memory accesses if the value of the counter is about to drop below X.

Flexibility. It is important to note that the Back-pressure module imposes no restrictions on the exact placement of the idle slots; so long as there are X idle slots *anywhere* in every sliding window of Y slots, the memory access pattern is valid. In particular, if the user memory access pattern itself satisfies this constraint, back-pressure is not needed. This affords the user of the memory the flexibility to supply idle slots when a memory access is not needed, and, consequently, access the memory in bursts without any stalls for refresh. In fact, larger X and Y imply more freedom in distributing the idle slots. For example, the constraint $X = 10$, $Y = 100$ has the same fraction of idle slots as $X = 1$, $Y = 10$. However, the former constraint allows for much longer bursts of consecutive memory accesses (up to 90 slots) than the latter (up to 9 slots).

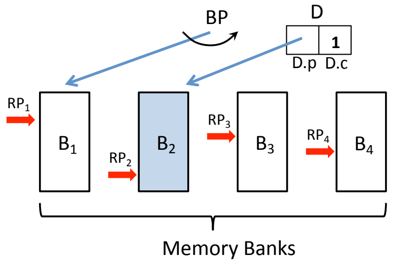


Figure 2: **Block diagram of the VR Tracking module. In this example, bank B_2 has a deficit of one refresh.**

3.1.2 Tracking module

The Tracking module determines the schedule at which rows are refreshed. In normal operations, it attempts to refresh the rows in a round-robin order. If a row is blocked and cannot be refreshed on its turn, a *deficit counter* is incremented for that row's bank. The algorithm gives priority to refreshes for the bank with deficit, until its deficit is cleared.

We now formally describe the algorithm. For simplicity, we assume the case of a single read/write port ($m = 1$; see Figure 1). The general version of the algorithm for any m is given in §5.1. The Tracking module maintains the following state:

- A row pointer per bank, $1 \le RP_i \le R$ (for $i = 1, \ldots, B$).
- A single bank pointer, $1 \le BP \le B$.
- A deficit register, D, which consists of a counter, $0 \le D_c \le D_{MAX}$, and a bank pointer $1 \le D_p \le B$.

The row pointers move in round-robin fashion across the rows of each bank and the bank pointer moves round-robin across the banks. These pointers keep track of which row is to be refreshed next according to round-robin order, skipping the refreshes that are blocked due to a conflicting memory access. The deficit register, D, records which bank, if any, has a deficit of refreshes and its deficit count (see Figure 2 for an illustration). The value of the deficit counter is capped at:

$$D_{MAX} = X + 1, \tag{1}$$

where X is the parameter from the constraint imposed on memory accesses by the Back-pressure module (at least X idle slots in every Y slots). This choice for the maximum deficit is based on the analysis of the algorithm that is presented in §4.

In each time slot, the bank that should be refreshed, B^*, is chosen according to Algorithm 1. The algorithm first checks whether the bank with a deficit (if one exists) is idle, and if so, it chooses this bank to refresh and decrements the deficit counter (Lines 1-3). Next, the algorithm tries to refresh the bank BP which is next in round-robin order (Lines 4-6). Finally, if the bank BP is blocked, it is skipped over, the deficit register is associated with it, and the deficit counter is incremented (Lines 7-11). Note that the deficit register may already be associated with bank BP in which case Line 9 is superfluous.

Once a bank B^* is chosen for refresh, its row pointer RP_{B^*} determines which of its rows to refresh. After the refresh, RP_{B^*} is incremented. All increments to row and bank pointers are modulo R and B respectively.

Algorithm 1 Versatile Refresh Scheduling Algorithm ($m = 1$)

```
Input: $BP$, $D$, currently blocked bank $\hat{B}$.
Output: $B^*$, the bank from which a row should be refreshed in
    this time slot.
 1: if $D_c > 0$ and $D_p \neq \hat{B}$ then
 2:     $B^* \leftarrow D_p$           {Refresh the bank with deficit.}
 3:     $D_c \leftarrow D_c - 1$
 4: else if $BP \neq \hat{B}$ then
 5:     $B^* \leftarrow BP$      {Refresh the bank in round robin order.}
 6:     $BP \leftarrow BP + 1$
 7: else
 8:     $B^* \leftarrow BP + 1$ {Skip blocked bank and increment deficit.}
 9:     $D_p \leftarrow BP$
10:     $D_c \leftarrow \min(D_c + 1, D_{MAX})$
11:     $BP \leftarrow BP + 2$
12: end if
```

A simple but important property is that at most one bank can have a deficit of refreshes at any time. Therefore, only a single deficit register is required. This is because with one read/write port ($m = 1$), at most one bank can be blocked in each time slot. In general, m deficit registers are required (§5.1).

Example. The timing diagram in Figure 3 shows an example of the operation of the refresh scheduling algorithm with $B = 4$, $X = 1$, and $Y = 7$. During the first 6 clock cycles, bank B_2 is continuously accessed and the algorithm chooses B^* in round-robin order, skipping over B_2 at time slots $t = 2$ and $t = 5$. The deficit counter, D_c, is also incremented at these time slots and D_p points to B_2. Time slot $t = 7$ is an idle slot,[2] and subsequently, bank B_3 is accessed. Hence, the algorithm refreshes bank B_2 at time slots $t = 7$ and $t = 8$ reducing its deficit to zero. Note that in these time slots, the bank pointer, BP, does not advance. With the deficit counter at zero, the refreshes continue in round-robin order according to BP.

3.2 Enhancement

In the basic design, we required that the Back-pressure module guarantee (at least) X idle time slots in every sliding window of Y time slots. Here, we briefly discuss a simple enhancement that allows us to relax this requirement and improve system throughput.

In each time slot, the Tracking module has a bank that it *prefers* to refresh, ignoring potential conflicting memory accesses. The preferred bank is either the bank with a deficit, if one exists, or the bank pointed to by the bank pointer. Now the main observation is that any time slot with a memory access to a bank other than the preferred bank is equivalent to an idle slot for the refresh scheduling algorithm. In both cases, the preferred bank is refreshed. We refer to such time slots as *no-conflict* slots. For example, in Figure 3, time slot $t = 8$ is a no-conflict even though there is a memory access to bank B_3, because the preferred bank, B_2, is idle. In fact, all the time slots $t = 1, 2, 7, 8, 9, 10$ are no-conflict slots in this example.

Therefore, instead of requiring the Back-pressure module to enforce idle slots, it suffices that it guarantee that there are (at least) X no-conflict slots in every Y consecutive slots. This can be done using exactly the same bit-map structure as before. The only difference is that the Tracking module needs to inform the Back-pressure module which slots are no-conflict slots. The memory access is back-pressured only if the number of no-conflict slots in the last Y cycles is about to drop below X.

[2]Note that this is necessary (and will be enforced by the Back-pressure module) to ensure that there is at least $X = 1$ idle slot in every $Y = 7$ slots.

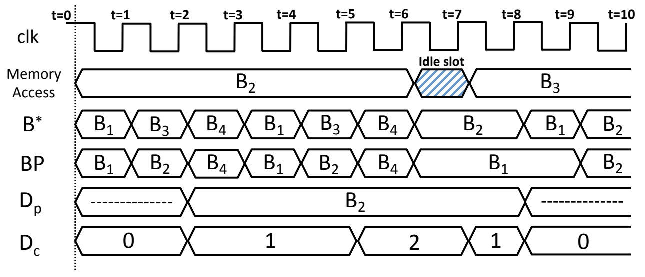


Figure 3: Timing diagram for the VR algorithm with $B = 4$, $X = 1$, and $Y = 7$. The algorithm chooses bank B^* for refresh in each cycle.

This enhancement can greatly reduce the amount of back-pressure required for workloads that multiplex memory accesses across multiple banks, because even without idle slots, there may be adequate no-conflict slots. In fact, as our simulations in §7 show, even with moderate levels of multiplexing, the overhead of refresh can almost entirely be hidden.

Note: In the rest of the paper, we always consider Versatile Refresh with the 'no-conflict' enhancement.

Remark 2. It is important to note that a *strong adversary* who knows the internal state of the algorithm at each step can always create conflicts by accessing the preferred bank. In this case, the no-conflict slots are exactly the idle slots that are enforced by back-pressure and the enhancement does not provide any benefits. Hence, in the *worst case*, there are exactly X back-pressures required in every Y cycles.

3.3 Implementation Complexity

The VR algorithm has a very low state requirement. The tracking module needs $B + 1$ pointers (B row pointers and one bank pointer), and an extra counter and pointer for tracking deficits. The implementation of Algorithm 1 is a simple state-machine. The back-pressure module also requires a Y-bit bitmap and a counter. Overall, the complexity is $O(B + Y)$. (The next section discusses the choice of X and Y in detail). We estimate that a typical instantiation of VR requires ~10K gates and occupies ~0.02mm^2 in area on a 45nm process node. Even on a relatively small instantiation with say 1Mb eDRAM, this is only ~2.5% of the eDRAM area.

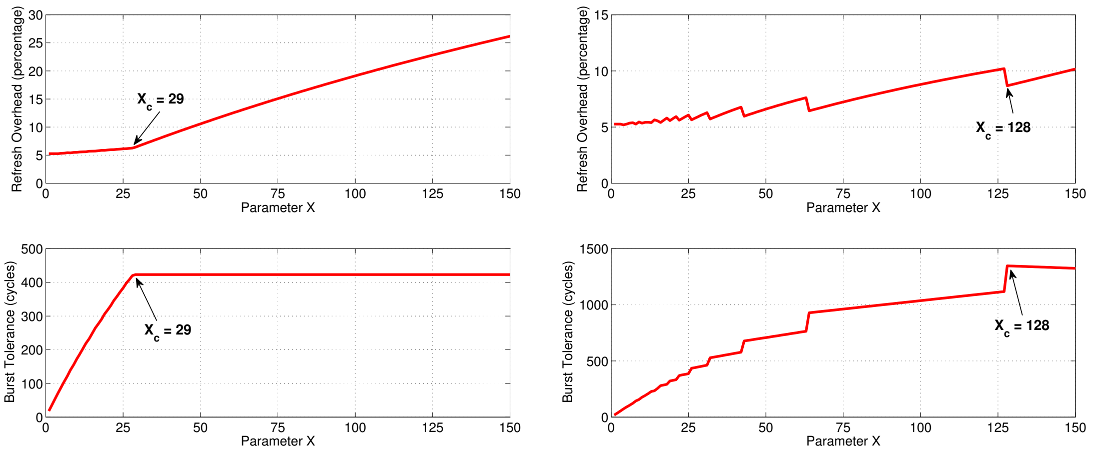
4. ANALYSIS OF THE VERSATILE REFRESH ALGORITHM

In this section we provide a mathematical analysis of the VR algorithm. The aim of our analysis is to determine the conditions under which the algorithm can guarantee data integrity—always refresh all rows in time—*for any input memory access pattern*. By virtue of our analysis, we discuss how the parameters X and Y should be chosen based to the macro configuration and the workload characteristics. We further show that the VR algorithm is nearly optimal in the sense of worst-case refresh overhead by proving a lower-bound on the refresh overhead of any refresh scheduling algorithm that ensures data integrity.

Notation: In this paper, $\lfloor x \rfloor$ and $\lceil x \rceil$ denote the floor and ceiling functions respectively. Also, $\mathbb{1}(\cdot)$ is the indicator function.

4.1 Data integrity for the VR algorithm

As previously mentioned, the back-pressure module ensures that there are (at least) X no-conflict slots in any Y consecutive time



(a) $B = 16 < \frac{W}{R}$ (b) $B = 8 < \frac{W}{2R}$

Figure 4: Worst case refresh overhead and burst tolerance for VR algorithm in Example 1. Note the different scales of the y-axes in the plots.

slots. The analysis proceeds by considering arbitrarily fixed X and Y, and finding the smallest W for which the algorithm can ensure each row is refreshed at least once in every W time slots.

Theorem 1. *Consider the VR algorithm with parameters X and Y. Let $R = aX + b$, where $1 \leq b \leq X$. Then $W \geq W_{VR}$ is necessary and sufficient for the VR Algorithm to refresh all rows in time, where:*

$$W_{VR} = \begin{cases} RB + Y - X + \lceil \frac{Y-X}{B-1} \rceil & \text{if } Y \leq BX, \\ (a+1)Y + bB + 1 & \text{if } Y > BX. \end{cases} \quad (2)$$

The proof of Theorem 1 is given in §4.3.

4.1.1 *Choosing the parameters* $\mathbf{X}$ *and* $\mathbf{Y}$

In practice, we are typically given B and R. The amount of time we have to refresh each row, W, can also be determined using the retention time specifications of the eDRAM (this typically depends on the vendor, the technology, and the maximum operating temperature). We need to choose X and Y for the VR algorithm.

We consider the following two performance metrics:
(i) **Worst-case Refresh Overhead: OV $= \mathbf{X}/\mathbf{Y}$.** Since there are at most X back-pressures in every Y consecutive slots, the throughput loss due to refresh is at most OV.
(ii) **Burst Tolerance: $\mathbf{Z} = \mathbf{Y} - \mathbf{X}$.** This is the longest burst of memory accesses to a particular bank without back-pressure. It is a measure of the flexibility the algorithm has in postponing refreshes during bursts of memory accesses.

Ideally, we want a small refresh overhead and a large burst tolerance. However, we will see that there is a tradeoff between these two metrics.

For any choice of X, we can use Theorem 1 to find the largest Y for which the VR algorithm can successfully refresh the banks. Denote this by Y^*. Using Eq. (2), we can derive:

$$Y^* = \begin{cases} W - (B-1)R - \left\lceil \frac{W}{B} \right\rceil + X & \text{if } RB \leq W \leq (R+X)B, \\ \left\lfloor \frac{W - bB - 1}{a+1} \right\rfloor & \text{if } W > (R+X)B. \end{cases} \quad (3)$$

Note that for a given X, $OV_X = X/Y^*$ is the smallest refresh overhead, and $Z_X = Y^* - X$ is the largest burst tolerance.

Refresh Overhead vs Burst Tolerance tradeoff. Our analysis indicates that, overall, the flexibility afforded in postponing refreshes by having a large burst tolerance results in a larger worst-case refresh overhead. The parameter X controls this tradeoff. We briefly summarize our findings. The derivation of these facts is simple and omitted.

- **Minimizing Refresh Overhead:** The refresh overhead is not a monotone function of X in general. However, the overall trend is that the refresh overhead is higher for large values of X. In fact, the refresh overhead with $X = 1$, given by:

$$OV_1 = \begin{cases} \frac{1}{W - BR} & \text{if } RB < W \leq (R+1)B, \\ \frac{1}{\lfloor (W-B-1)/R \rfloor} & \text{if } W > (R+1)B, \end{cases} \quad (4)$$

is very close to the smallest possible value.[3]

- **Maximizing Burst Tolerance:** The burst tolerance increases with X and attains its largest value for all $X_c \leq X \leq R$, where:

$$X_c = \min\{R, \lceil \frac{W}{B} \rceil - R\}. \quad (5)$$

In fact, $X > X_c$ only results in higher refresh overhead, without any benefit in burst tolerance and should not be used. The maximum value of burst tolerance is given by:

$$\max\{W - R(B+1) - 1, W - R(B-1) - \lceil \frac{W}{B} \rceil\}. \quad (6)$$

Example. Consider two eDRAM macros with $B = 16$ and $B = 8$ banks. Each bank has $R = 128$ rows. The rows must be refreshed at most every 10μs. Assume clock frequency 250MHz, corresponding to every row requiring a refresh every $W = 2500$ time slots. Note that the first macro corresponds to scenarios with W slightly larger than $RB = 2048$, while the second macro corresponds to scenarios with W moderately larger than $RB = 1024$. We vary X from 1 to 150 (corresponding to increasing levels of flexibility in the refresh pattern), and use Eq. (3) to determine the best Y, and the worst-case refresh overhead and burst tolerance in each case. The results are shown in Figure 4.

Overall, the refresh overhead and burst tolerance both increase as X increases. For the case $B = 16$, the refresh overhead increases

[3]It is easy to show that the refresh overhead of any X is bounded by: $OV_X \geq OV_1/(1 + OV_1)$.

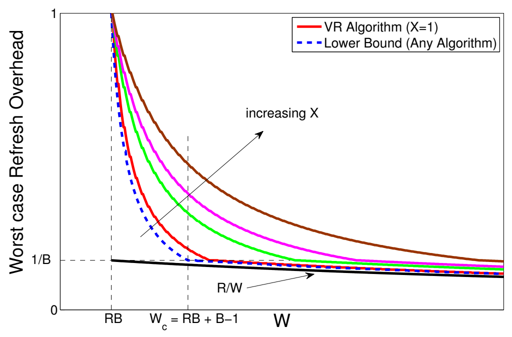


Figure 5: **Behavior of the worst-case refresh overhead for different X as W varies.**

from 5.3% to over 25%, and the burst tolerance increases from 18 to 423 cycles, as X varies from 1 to 150. Note that at $X = X_c = 29$, the burst tolerance reaches its maximum (see Eq. (5)). Also, the refresh overhead at $X = 29$ is 6.4%, only 1.1% higher than at $X = 1$. Therefore, unless the lowest refresh overhead is crucial, $X = 29$ is a good choice in this case. For $B = 8$, the refresh overhead is smallest for $X = 4$ at 5.19%; it is 5.26% at $X = 1$. The largest value of the burst tolerance (1347 cycles) is achieved at $X = X_c = 128$, which has a refresh overhead of 8.68%.

Recall that the refresh overhead for the Period Refresh algorithm (§2.2) is as high as RB/W in the worst case, amounting to 81.9% of memory bandwidth in the $B = 16$ case, and 40.9% in the $B = 8$ case. Hence, compared to Periodic Refresh, the refresh overhead for the VR algorithm (in the best setting) is 76.6% lower in the $B = 16$ case, and 35.71% lower in the $B = 8$ case.

4.2 Near-optimality of the VR algorithm

In this section we demonstrate that the VR algorithm is near-optimal in the sense of worst-case refresh overhead. The following theorem provides a lower bound for the worst-case refresh overhead of *any* algorithm that guarantees data integrity. This allows us to quantify how far the VR algorithm is from the 'optimal' algorithm.

Theorem 2. *The worst-case refresh overhead for any algorithm that ensures data integrity is lower bounded by :*

$$OV_{LB} = \max\Big\{\frac{1}{W - BR + 1}, \frac{R}{W - B + 1}\Big\}. \tag{7}$$

The proof of Theorem 2 is deferred to Appendix E. Comparing Eq. (7) with Eq. (4) shows that the VR algorithm with $X = 1$ (small X) achieves a near-optimal worst case overhead.

Figure 5 illustrates the behavior of the necessary refresh overhead (the lower bound), and the overhead of the VR algorithm for different choices of X as we vary W. Note that this plot is for a fixed B, and R. An obvious requirement to refresh all the rows in time is $W \geq RB$. It is important to note the two distinct regimes for W: (i) $RB \leq W \leq W_c$ and (ii) $W > W_c$, where:

$$W_c = RB + B - 1. \tag{8}$$

For $RB \leq W \leq W_c$, the necessary refresh overhead rapidly drops to $1/B$. For larger W, the necessary overhead decreases much more gradually and is very close to the trivial bound of R/W (the required refresh overhead when a single bank is always read.). Observe that for small values of X, the refresh overhead of the VR algorithm is close to the necessary lower bound for any algorithm. Larger values of X increase the worst-case refresh overhead of the algorithm. However, they also increase the burst tolerance, allowing larger burst of memory accesses without back-pressure. We explore this tradeoff in choosing the value of X further in §7 using simulations.

4.3 Proof of Theorem 1

In order to prove that $W \geq W_{VR}$ is necessary, we introduce a special (adversarial) memory access pattern and prove that $W \geq W_{VR}$ is necessary to refresh all rows in time for this pattern. The details are provided in Appendix A.

In the following, we prove that $W \geq W_{VR}$ is sufficient. We first establish some definitions and lemmas.

Definition 1. *The bank pointer progress during an interval J, denoted by $P(J)$, is the total number of bank positions it advances in the interval. For example, if the bank pointer is at bank 1 at the start of J, and ends at bank 2 after one full rotation across the banks, $P(J) = B + 1$. Furthermore, the number of times the bank pointer passes some bank $\tilde{B}$ is denoted by $p_{\tilde{B}}(J)$. In the previous example, $p_1(J) = 2$ and $p_{\tilde{B}}(J) = 1$ for $2 \leq \tilde{B} \leq B$.*

The following lemma is the key ingredient in the proof. It provides a precise characterization of the number of refreshes to each bank in a given time interval.

Lemma 1. *Consider an interval J, spanning $T \geq 1$ time slots. Let C_0 and C_1 be the initial and final values of the deficit counter for J, and assume these deficits correspond to banks B_0 and B_1 respectively. If the deficit counter does not reach $D_{MAX} = X + 1$ at any time during J, then:*

$$P(J) = T + C_1 - C_0. \tag{9}$$

Furthermore, the number of refreshes to any bank, $\tilde{B}$, during J is exactly:

$$f_{\tilde{B}}(J) = p_{\tilde{B}}(J) + C_0\mathbb{1}\{\tilde{B} = B_0\} - C_1\mathbb{1}\{\tilde{B} = B_1\}. \tag{10}$$

PROOF. In each time slot, the bank pointer advances by either zero, one, or two positions. Let N_0, N_1, and N_2 denote the number of times each of these occur in the interval J. Hence,

$$N_0 + N_1 + N_2 = T. \tag{11}$$

Since the deficit counter never reaches D_{MAX} during J, it is incremented exactly N_2 times and decremented exactly N_0 times. Therefore, we must have:

$$C_0 + N_2 - N_0 = C_1. \tag{12}$$

Using (11) and (12), we obtain:

$$P(J) = N_1 + 2N_2 = T + C_1 - C_0.$$

Now consider an arbitrary bank, $\tilde{B}$. Note that the deficit for $\tilde{B}$ varies from $C_0\mathbb{1}\{\tilde{B} = B_0\}$ to $C_1\mathbb{1}\{\tilde{B} = B_1\}$ during J. In each of the $p_{\tilde{B}}(J)$ times the bank pointer passes $\tilde{B}$, it either skips over it (if it's blocked), or refreshes it. Let s be the number of times it skips over it. The deficit for bank $\tilde{B}$ is incremented exactly s times (since it never reaches D_{MAX}). Therefore, it must also have been decremented exactly $C_0\mathbb{1}\{\tilde{B} = B_0\} + s - C_1\mathbb{1}\{\tilde{B} = B_1\}$ times.

Note that the refreshes which decrement the deficit for $\tilde{B}$ are different from those due to the bank pointer passing $\tilde{B}$, since the bank pointer does not advance in the former while it advances in the latter. As discussed, bank $\tilde{B}$ is refreshed $C_0\mathbb{1}\{\tilde{B} = B_0\} + s - C_1\mathbb{1}\{\tilde{B} = B_1\}$ times in the first manner. Also, it is refreshed $(p_{\tilde{B}} - s)$ times in the second manner. This results in a total of $p_{\tilde{B}}(J) + C_0\mathbb{1}\{\tilde{B} = B_0\} - C_1\mathbb{1}\{\tilde{B} = B_1\}$ refreshes to bank $\tilde{B}$ during J. $\square$

The gist of the proof of Theorem 1 is to divide the evolution of the system into appropriate intervals, so that Lemma 1 can be applied to bound the number of refreshes for a given bank, relative to the length of each interval. For the rest of the proof, we focus on an arbitrary bank, $\tilde{B}$.

Definition 2. *At time t, the* state *of the VR scheduling system, $S(t)$, is the current values of the bank pointer and deficit register; i.e., $S(t) = (BP(t), D_p(t), D_c(t))$. The system is in state S_D (for **Deficit State**), if bank $\tilde{B}$ has a positive deficit: $D_p(t) = \tilde{B}$ and $D_c(t) > 0$. The system is in state S_{ND} (for **No Deficit State**), if bank $\tilde{B}$ does not have a deficit and is next in turn for a refresh according to the bank pointer: either $D_c(t) = 0$ or $D_p(t) \neq \tilde{B}$, and $BP(t) = \tilde{B}$.*

Definition 3. *Assume at time t_0, the system is in one of the states S_D or S_{ND}. An* epoch*, starting at t_0, proceeds until bank $\tilde{B}$ either gets at least one refresh and the system is in state S_{ND}, or bank $\tilde{B}$ gets X refreshes and the system is in S_D. Hence, the duration of the epoch, starting at $t_0 = 0$ is given by:*

$$T = \min\{t \geq 1 | f(t) \geq 1, S(t) = S_{ND}; \\ \text{or } f(t) = X, S(t) = S_D\}, \tag{13}$$

where $f(t)$ is the number of refreshes to bank $\tilde{B}$ until time t.

First note that T is finite. To see this, assume $T = \infty$, and consider the time slot t such that $f(t) = X$. There are two cases: (i) The system is currently at state S_D. (ii) $D_c(t) = 0$ or $D_p(t) \neq \tilde{B}$. In the latter case, when bank $\tilde{B}$ is met by BP for the next time, it still has no deficit and the system reaches the state S_{ND}. Therefore, after a finite time, there is a transition either to S_D or S_{ND}, showing that T is finite. Second, the same argument implies that the number of refreshes to bank $\tilde{B}$ in each epoch is at most X. In the following lemma, we use Lemma 1 to bound the duration of an epoch relative to how many refreshes occur to bank $\tilde{B}$. We defer its proof to Appendix B.

Lemma 2. *Consider an epoch starting at state $S_0 \in \{S_D, S_{ND}\}$ and ending at state $S_1 \in \{S_D, S_{ND}\}$. Let C_0 and C_1 be the values of the deficit counter at the beginning and the end of the epoch respectively. Also, let δ be the epoch duration and f be the number of refreshes to bank $\tilde{B}$ during this epoch. Then f and δ satisfy:*

$$\begin{array}{lll} f \in \{1, \ldots, X\} & \delta \leq C_0 + fB - C_1 & \textit{if } (S_0, S_1) = (S_{ND}, S_{ND}), \\ f = X & \delta \leq C_0 + Y & \textit{if } (S_0, S_1) = (S_{ND}, S_D), \\ f \in \{1, \ldots, X\} & \delta \leq fB - C_1 & \textit{if } (S_0, S_1) = (S_D, S_{ND}), \\ f = X & \delta \leq Y & \textit{if } (S_0, S_1) = (S_D, S_D). \end{array}$$

The following lemma extends the result of Lemma 2 to a sequence of consecutive epochs.

Lemma 3. *Consider a sequence of consecutive epochs starting at state $S_0 \in \{S_D, S_{ND}\}$. Let $S_l \in \{S_D, S_{ND}\}$ be the state of the system and C_l be the value of the deficit counter at the end of the l^{th} epoch. Also, let F_l be the total number of refreshes to bank $\tilde{B}$ and Δ_l be the total number of slots during the first l epochs. If $F_l = \psi_l X + \xi_l$, where $0 \leq \xi_l \leq X - 1$. Then:*

$$\Delta_l \leq \begin{cases} C_0 + F_l B - C_l \mathbb{1}\{S_l = S_{ND}\} & \textit{if } Y \leq BX, \\ C_0 + \psi_l Y + \xi_l B - C_l \mathbb{1}\{S_l = S_{ND}\} & \textit{if } Y > BX. \end{cases} \tag{14}$$

The proof of Lemma 3 is straight forward by induction and is given in Appendix C. The following lemma provides a tight bound on the maximum value of the deficit counter. Its proof is given in Appendix D.

Lemma 4. *At any time, the deficit counter satisfies the bound:*

$$D_c(t) \leq \begin{cases} \lceil \frac{Y-X}{B-1} \rceil & \textit{if } Y \leq BX, \\ X + 1 & \textit{if } Y > BX. \end{cases} \tag{15}$$

We are now ready to prove the theorem.

Proof of Theorem 1 (sufficiency). We will show that at any time, the number of slots it takes for all the rows of $\tilde{B}$ to get refreshed (equivalently, for $\tilde{B}$ to get R refreshes) is at most W_{VR}. Since $\tilde{B}$ is an arbitrary bank, this proves the sufficiency part.

Assume the system is in an arbitrary initial state and the value of the deficit counter is C_i. It is easy to see that it takes

$$\Delta_i \leq C_i + B - 1 - C_0 \tag{16}$$

slots for the system to enter state $S_0 \in \{S_D, S_{ND}\}$ for the first time, where C_0 is the value of the deficit counter in S_0. Now consider a sequence of epochs beginning in state S_0, and define F_l and Δ_l as in Lemma 3. Let

$$l^* = \max\{l | F_l < R\}. \tag{17}$$

Since the number of refreshes to bank $\tilde{B}$ in one epoch is at most X, we have $F_{l^*} = R - e$, where $1 \leq e \leq X$. Let

$$b + X - e = uX + v, \tag{18}$$

where $u \in \{0, 1\}$ and $0 \leq v \leq X - 1$. We obtain:

$$F_{l^*} = (a-1)X + b + X - e = (a + u - 1)X + v. \tag{19}$$

Now, invoking Lemma 3, we can bound the total number of slots for the l^* epochs as:

$$\Delta_{l^*} \leq \\ \begin{cases} C_0 + (R-e)B - C_{l^*}\mathbb{1}\{S_{l^*} = S_{ND}\} & \text{if } Y \leq BX, \\ C_0 + (a+u-1)Y + vB - C_{l^*}\mathbb{1}\{S_{l^*} = S_{ND}\} & \text{if } Y > BX. \end{cases} \tag{20}$$

Bank $\tilde{B}$ has received $R - e$ refreshes until the end of the $(l^*)^{th}$ epoch. We only need to bound how long it takes for it to receive the next e refreshes. Let this take Δ_e slots. By definition of l^* (see Eq. (17)), all e refreshes must occur during the $(l^* + 1)^{th}$ epoch. This implies that the deficit for bank $\tilde{B}$ cannot be zero after it gets refresh $j < e$ during epoch $l^* + 1$. Otherwise, the next time BP meets $\tilde{B}$, the system transits to S_{ND}, and the $(l^* + 1)^{th}$ epoch contains only j refreshes. Therefore, if $S_{l^*} = S_D$, then the next e subsequent no-conflict slots will refresh bank $\tilde{B}$. If $S_{l^*} = S_{ND}$, at most C_{l^*} slots can be initially spent reducing the deficit of some bank other than $\tilde{B}$ before the first refresh to bank $\tilde{B}$ occurs. (Note that $BP(l^*) = \tilde{B}$). Afterward, any subsequent no-conflict slots will refresh bank $\tilde{B}$ until its e^{th} refresh. Since it takes at most $Y - X + e$ slots to get e no-conflict slots, we have:

$$\Delta_e \leq C_{l^*}\mathbb{1}\{S_{l^*} = S_{ND}\} + Y - X + e. \tag{21}$$

The total number of slots for bank $\tilde{B}$ to receive R refreshes is bounded using (16), (20), (21) by:

$$\Delta_i + \Delta_{l^*} + \Delta_e \leq \\ \begin{cases} C_i + RB + Y - X + (B-1)(1-e) & \text{if } Y \leq BX, \\ C_i + (a+1)Y + bB - X & \text{if } Y > BX. \\ +(u-1)(Y - BX) + (B-1)(1-e) & \end{cases} \tag{22}$$

For the $Y > BX$ case in the above, we have used (18) to write $v = b - e - (u-1)X$.

Consider the two cases separately:

1)$\mathbf{Y} \le \mathbf{BX}$: Using $(B-1)(1-e) \le 0$ (because $1 \le e \le X$), and Lemma 4 to bound C_i, we have:

$$\Delta_i + \Delta_{l^*} + \Delta_e \le RB + Y - X + \lceil \frac{Y-X}{B-1} \rceil = W_{VR}. \quad (23)$$

2)$\mathbf{Y} > \mathbf{BX}$: Using $(u-1)(Y-BX) \le 0$ (since $u \in \{0,1\}$), and $(B-1)(1-e) \le 0$, and Lemma 4, we have:

$$\Delta_i + \Delta_{l^*} + \Delta_e \le (a+1)Y + bB + 1 = W_{VR}. \quad (24)$$

Therefore, it takes at most W_{VR} slots for bank $\tilde{B}$ to get R refreshes. Since bank $\tilde{B}$ was arbitrary, $W \ge W_{VR}$ is sufficient for the VR algorithm to refresh all the rows in time. □

5. GENERALIZATION TO MULTIPLE READ/WRITE PORTS

We describe how the VR algorithm can be modified for the general setting with $m > 1$ read/write ports, and extend our analysis to derive the conditions under which the algorithm can refresh all rows in time for any input memory access pattern using m read/write ports (extension of Theorem 1). We also lower bound the worst case refresh overhead of any algorithm (extension of Theorem 2) in this setting to the determine the optimality gap of the Generalized VR (GVR) algorithm.

5.1 VR with multiple read/write ports

With $1 < m < B$ read/write ports, the user can potentially block multiple banks at each time slot. The GVR algorithm is obtained by modifying VR as follows. The Tracking module maintains the state of m deficit registers, $D^{(1)}, \cdots, D^{(m)}$. The registers keep track of which banks have been recently blocked and have a deficit of refreshes. In addition, a new pointer, CP, is needed to choose which bank to refresh. At any time, CP points to one of the banks with a positive deficit (if one exists). The bank that should be refreshed, B^*, is chosen as follows:

- If the bank that CP points to is blocked, then B^* is chosen according to the BP pointer.
- If the bank that CP points to is not blocked, then it is refreshed and its deficit is decremented. The pointer CP then advances to the next bank (in round robin order) with a deficit. This bank is found by consulting the deficit registers.

The Back-pressure module ensures that there are (at least) X no-conflict slots in every Y consecutive slots, similar to the case $m = 1$.

The following properties of the algorithm are worth noting:

(i) If at a time slot the banks $BP, BP+1, \cdots, BP+j-1$ are blocked ($1 \le j \le m$), then BP advances by $j+1$ positions and each of the corresponding deficit counters are incremented in that time slot.

(ii) At each time slot, the *preferred bank* for refresh is the bank pointed to by CP, if one exists, or the bank pointed to by the bank pointer, BP. Then, a no-conflict slot is a slot that the preferred bank is not blocked by any of the m read/write ports.

(iii) While a bank has a positive deficit, it is guaranteed to be refreshed at least once in every m no-conflict slots. This is because every no-conflict slot advances the CP pointer to the next bank with a deficit in round-robin order, and there are at most m such banks at any time.

5.2 Analysis

The following theorem provides a sufficiency condition for the GVR algorithm to ensure data integrity. Its proof proceeds along the same lines as for the case $m = 1$ (Theorem 1) and is omitted.

Theorem 3. *Consider the GVR algorithm with parameters X and Y. Let $mR = aX + b$, $b = pm + q$, and $m = gX + h$, where $1 \le b \le X$, $1 \le q \le m$ and $1 \le h \le X$. Then $W \ge W_{VR}$ is sufficient for the GVR Algorithm to refresh all rows in time, where:*

$$W_{VR} = \begin{cases} RB + (g+1)Y - X + \frac{B}{m}(h-1) & \text{if } mY \le BX, \\ +m\lceil \frac{Y-X}{B-m} \rceil + m - 1 & \\ & \\ (a+1)Y + (p+1)B - X & \text{if } mY > BX. \\ +\frac{B}{m}(q-1) + m(X+1) & \end{cases} \quad (25)$$

Note that the lower bound obtained on W in Theorem 1 (case $m = 1$) is both necessary and sufficient. However, for multiple read/write ports, Theorem 3 only provides a sufficiency bound.

We can use Theorem 3 to determine the largest Y for a given X such that the GVR algorithm can ensure data integrity, following the same approach discussed in §4.1.1 for the $m = 1$ case. Also, the parameter X controls the same tradeoff between worst-case refresh overhead and burst tolerance as before.

Lower bound on refresh overhead of any algorithm. The result of Theorem 2 can also be extended for the setting of m read/write ports. This allows us to quantify the gap between the refresh overhead of the GVR algorithm and the optimal refresh overhead.

Theorem 4. *The worst case refresh overhead of any algorithm that ensures data integrity is lower bounded by:*

$$OV_{LB} = \max\{\frac{m}{W - BR + m}, \frac{mR}{W - B + m}\}. \quad (26)$$

For any choice of X, the worst case refresh overhead of the GVR algorithm can be found using Theorem 3. For example, with $X = 1$, after some simple algebraic manipulations, we have:

$$OV_1 \le \max\left\{\frac{1}{\lfloor \frac{W-RB}{m} \rfloor - 2}, \frac{1}{\lfloor \frac{W-B-2m+1}{mR} \rfloor}\right\}. \quad (27)$$

By comparing Eqs. (26) and (27), it is evident that the GVR algorithm with $X = 1$ (more generally, small X) achieves a near optimal worst case refresh overhead; especially when $W - RB \gg m$, which is the case in practice.

6. FORMAL VERIFICATION

In this section we use the formal property checking tool Magellan [3] to independently verify our analytical results for an implementation of the VR algorithm. Magellan formally verifies user-specified properties for a given design. Properties are specified using SystemVerilog [15] assertions and Magellan's formal property checkers try to mathematically prove whether the behavior of a particular design conforms to the given assertions. If the assertions are static properties (i.e. invariants) of the design, and if Magellan can validate that these properties are universally true (irrespective of the input parameters), then this leads to a formal proof that a particular design works.

The tool either returns a `Proven` or `Falsified` outcome when it converges. The `Proven` outcome means that the design has the correct behavior for any input satisfying the assertions. The `Falsified` outcome returns a counter example in which the design violates the given property. However, if the state space is large, the formal tool may not converge. The key is to identify and isolate static properties that are independent of each other in order to facilitate convergence.

Y	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Formal tool (m=1)	128	130	131	132	133	134	135	136	153	169	185	201
Analysis (m=1)	128	130	131	132	133	134	135	136	153	169	185	201
Formal tool (m=2)	128	133	135	138	171	203	235	267	299	331	363	395
Analysis (m=2)	130	134	136	138	171	203	235	267	299	331	363	395

Table 1: Comparison of the smallest feasible W for the VR algorithm derived by the Formal tool and analysis. The parameters used are $B = 8$, $R = 16$, $X = 1$, $m = 1, 2$ and $Y = 1, \ldots, 12$.

For the VR algorithm, we isolate the behavior of one row and one bank. We create an abstract model for memory, and track the time since the last refresh for every memory row in every bank. In our particular example, we consider a macro with $B = 8$ banks, $R = 16$ rows per bank, and $m = 1$ or 2 read/write ports. We develop RTL for the VR algorithm with $X = 1$ in Verilog. For a given value of Y and W, the formal tool verifies whether the RTL satisfies the constraint that each row is refreshed in every W cycles (as well as several intermediate constraints).

We use the formal tool to determine the smallest value of W for which the VR algorithm, with $X = 1$ and $Y = 1, \ldots, 12$, can successfully refresh the banks in time for any memory access pattern. This is done by sweeping W for each Y and finding the W for which the formal tool returns `Proven` with W, and `Falsified` with $W - 1$. In Table 1, we compare the results from the formal tool with the bound obtained by our analysis (Theorems 1 and 3). The results of the formal tool agree with the predictions from analysis. As previously discussed, with $m = 1$, the bound in Eq. (2) is tight (both necessary and sufficient). With $m = 2$, in some cases, our analysis gives a slightly larger lower bound for W than necessary, as proven by the formal tool.

Remark 3. The formal tool cannot be used to verify VR when the state space is large (e.g., large X or Y). For example, the formal tool did not converge even after 48 hours for $Y \geq 13$ in this design. This highlights the usefulness of having an exact formula that immediately shows whether a design is feasible, greatly simplifying the design process and reducing verification time.

7. PERFORMANCE EVALUATION

The analytical results of the previous sections determined the worst-case refresh overhead of the VR algorithm and showed that for small values of X, it is very close to a lower bound for any algorithm. It was also shown that increasing X results in an increase in the worst case refresh overhead, but provides tolerance to larger bursts of memory accesses because of increased flexibility in postponing refreshes. In this section we use simulations to explore the consequences of this tradeoff in choosing the value of X for a number of (non-adversarial) synthetic and trace-driven workloads.

7.1 Apex-Map Synthetic Benchmarks

Apex-Map [17] is a synthetic benchmark that generates memory accesses with parametrized degrees of spatial and temporal locality. Spatial locality is the tendency to access memory addresses near other recently accessed addresses and temporal locality refers to accessing the same memory addresses that have been recently referenced [22]. Apex-Map has been used to characterize various scientific and HPC workloads [16, 5].

The workload generated by Apex-Map is based on a stochastic process defined using two parameters, L and α. These parameters respectively control the degree of spatial and temporal locality, and can be varied independently between extreme values. Memory accesses occur in strides (contiguous blocks) of length L. When one stride is complete, a new one begins at a random starting address sampled from the distribution:

$$X = M u^{\frac{1}{\alpha}}, \tag{28}$$

where M is the total memory size and u is uniform on (0, 1). Note that with $\alpha = 1$, the starting addresses are uniformly spread, and as $\alpha \to 0$ they become increasingly clustered near address 0, leading to higher temporal locality.

Simulation Setup. We consider a 1Mb eDRAM macro organized as $B = 8$ banks of $R = 128$ rows. Each row has 8 words of 128 bits. The macro has $M = 8192$ words in total. The eDRAM operates at 250MHz and has a retention time of 10μs, giving $W = 2500$ cycles. We generate memory access patterns according to Apex-Map with $L = 1, 64, 4096$ and $\alpha = 0.001, 0.01, 0.1, 0.25, 0.5, 1$. Each simulation continues until 100,000 memory accesses have been completed. We measure the total memory overhead for refresh (the fraction of cycles with a back-pressure). The results presented in each case are the average of 20 runs.

Schemes Compared. As a baseline, we consider the conventional Periodic Refresh scheme which periodically performs a high-priority refresh operation (see §2.2). We use two configurations for Versatile Refresh: $X = 4, Y = 77$ and $X = X_c = 128, Y = 1475$. These values are derived based on the analysis in §4.1.1. The first configuration gives the lowest worst-case refresh overhead of 5.19%. The second configuration has a refresh overhead of 8.68% in the worst-case, but maximizes the burst tolerance by allowing refreshes to be postponed the longest.

The results are shown in Figure 6. The refresh overhead for Periodic Refresh is fairly consistent around 6.25%. VR with $X = 4$ achieves a lower refresh overhead than Periodic Refresh in all cases (at most 5.19%). VR with $X = 128$ has a higher refresh overhead than the other schemes when memory accesses are highly concentrated on a single bank, but has the lowest overhead when memory accesses are (even moderately) multiplexed across banks. For example, with $\alpha = 0.001$ and $L = 1, 64$, essentially all memory accesses are to the first bank and the refresh overhead for both VR schemes is almost the same as the worst case overhead. But with $\alpha \geq 0.1$ or $L = 4096$, VR with $X = 128$ shows almost no refresh overhead. This is because with $X = 128$, VR can tolerate up to 1347 consecutive memory accesses to a single bank without back-pressuring. (For $X = 4$, a back-pressure is forced after only 73 consecutive memory accesses to one bank.)

Note: The Apex-Map memory access patterns do not cause the worst-case refresh overhead for the Periodic Refresh algorithm. Indeed, the worst-case overhead for Periodic Refresh occurs if memory accesses become synchronized with the periodic refreshes, requiring a back-pressure for each refresh. In this case, the refresh overhead of Periodic Refresh would be $RB/W \approx 41\%$, while the overhead with VR is *always* at most 5.19% with $X = 4$, and 8.68% with $X = 128$.

Remark 4. Conventional DRAMs exploit locality in memory accesses by using a Fast page mode (or page mode), where a row can be kept "open" in a row buffer so that successive reads or writes

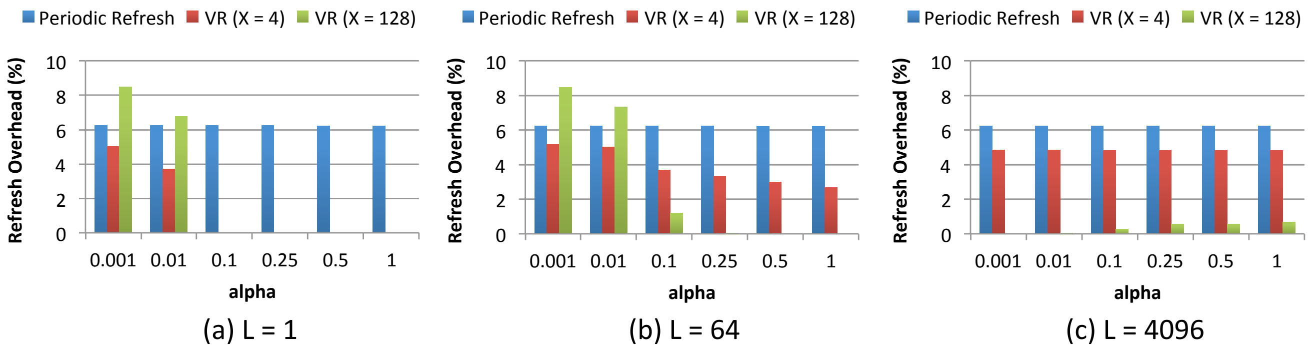


Figure 6: Apex-Map synthetic benchmark results. The results are the average of 20 runs and are within 0.1% of the true value with 95% confidence. Note that in some experiments the refresh overhead is zero and the bar is not visible.

within the row do not need to access the DRAM core and, therefore, do not suffer the delay of precharge. However, because of the wide IO typically used in eDRAM (128 bits or more), page mode is less effective and is not supported in recent eDRAM macros [10]. Therefore, we do not consider page mode in our simulations.

7.2 Switch Lookup Table

Embedded DRAM is extensively used in networking gear for packet buffers and lookup tables [24, 9, 10]. In this section we evaluate the performance implications of refresh for a lookup table stored in eDRAM in a high speed switch. We consider a 1Mb macro similar to the previous section ($B = 8, R = 128, W = 2500$), and use publicly available enterprise packet traces [12] to derive the memory access pattern. We assume a simple L2 lookup based on destination MAC address. For each packet, the destination MAC address is hashed to get the address that needs to be read from the lookup table.

The entire trace consists of 125,244,082 packets. We choose 10 random blocks of 1 million consecutive packets, and simulate the corresponding memory accesses. We assume one packet arrives every clock cycle requiring a lookup and measure the total overhead for refreshes (the fraction of cycles with a back-pressure) over each block. Table 2 shows the results with Periodic Refresh and Versatile Refresh with $X = 4$ and $X = 128$. The VR algorithm shows almost no refresh overhead. As suggested by the Apex-Map benchmark (§7.1), this is because the packet lookups are sufficiently multiplexed across the memory banks to allow VR to hide the refreshes. In fact, with $X = 128$, we did not observe even a single back pressure for the 10 million packet lookups simulated.

	Periodic Refresh	VR ($X = 4$)	VR ($X = 128$)
Refresh Overhead	6.65%	0.39%	0%

Table 2: Lookup table memory refresh overhead.

8. RELATED WORK

The research literature has mostly focused on the problems associated with refresh for DRAMs. We briefly discuss the most relevant work.

Concurrent refresh. Kirihata *et. al.* proposed concurrent refresh [11] to allow simultaneous memory access and refresh operations at different banks of an eDRAM. They also describe a concurrent refresh scheduler for a macro with 16 banks and one read/write port. However, the refresh scheduling algorithm is not fully analyzed and not readily generalizable to arbitrary configurations (e.g. multiple read/write ports). Moreover, the algorithm relies on a periodic external refresh signal to guarantee data integrity, which may unnecessarily increase refresh overhead. In contrast, VR is flexible in the placement of refreshes and allows configurable burst tolerance through the parameter X.

Delayed refresh. The JEDEC DDRx standards allow flexibility in the spacing of refresh operations. In the current DDR3 standard, it allows a maximum of eight refresh operations to be delayed [13] as long as the average refresh rate is maintained. Elastic Refresh [18] takes advantage of the flexibility and improves performance by delaying refreshes based on the predicted workload; it also uses the rule of thumb given in the standard for maximum number of refreshes. Our algorithm allows similar flexibility in refreshes to avoid clashes with read/write operations, while prescribing universal bounds on necessary number of refreshes and giving worst-case guarantees. Hence it improves performance by optimizing the minimum number of refreshes.

Per-word monitoring. Another approach to reducing refresh overhead is to monitor each memory cell, or a combination of cells, to decide a refresh schedule. For instance, Smart Refresh [4] keeps a time-out counter next to each memory row to monitor its urgency for refreshes. It uses this to eliminate unnecessary refreshes to rows that are automatically refreshed during a recent read or write operation. ESKIMO [7] uses program semantics information to decide which memory rows are inconsequential to the correct execution of the program (e.g. have been freed) and avoid refreshing them. These methods can improve throughput or save power for specific workloads, but require complex memory cell monitoring and application analysis. Our algorithm is orthogonal to these approaches and can benefit all workloads.

9. FINAL REMARKS

We introduced the Versatile Refresh (VR) algorithm, which exploits concurrent refresh to solve the eDRAM refresh scheduling problem. Our work can potentially be extended in a number of directions: (i) We could consider its applicability for DRAMs. This would mean adapting it to fit within the instantaneous power budget of a DRAM, and possibly slowing down the rate at which VR performs concurrent refreshes. (ii) We could consider the case where the refresh takes more than one cycle (as is common in current DRAMs). This can be easily incorporated in our analysis.

Our work lays a mathematical foundation in understanding the memory refresh scheduling problem. It can be used as an additional analytical tool by design engineers to complement system performance evaluation using simulations and workload analyses.

Acknowledgments

Mohammad Alizadeh and Adel Javanmard are supported by the Caroline and Fabian Pease Stanford Graduate Fellowship. We would like to thank Rob Busch from IBM Microelectronics; and also the anonymous reviewers for their helpful comments.

10. REFERENCES

[1] J. Barth, D. Plass, E. Nelson, C. Hwang, G. Fredeman, M. Sperling, A. Mathews, T. Kirihata, W. Reohr, K. Nair, and N. Caon. A 45 nm SOI Embedded DRAM Macro for the POWER7TM Processor 32 MByte On-Chip L3 Cache. *Solid-State Circuits, IEEE Journal of*, 46(1):64 –75, jan. 2011.

[2] C.-M. Chang, M.-T. Chao, R.-F. Huang, and D.-Y. Chen. Testing Methodology of Embedded DRAMs. In *Test Conference, 2008. ITC 2008. IEEE International*, pages 1 –9, oct. 2008.

[3] Magellan Formal Verification Tool. http://www.synopsys.com/tools/verification/functionalverification/pages/magellan.aspx.

[4] M. Ghosh and H.-H. S. Lee. Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 134–145, Washington, DC, USA, 2007. IEEE Computer Society.

[5] K. Ibrahim and E. Strohmaier. Characterizing the Relation Between Apex-Map Synthetic Probes and Reuse Distance Distributions. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 353 –362, sept. 2010.

[6] E. Ipek, O. Mutlu, J. F. Martń?nez, and R. Caruana. Self-Optimizing Memory Controllers: A reinforcement learning approach. In *International Symposium on Computer Architecture*, page 39?50, 2008.

[7] C. Isen and L. K. John. ESKIMO: Energy savings using Semantic Knowledge of Inconsequential Memory Occupancy for DRAM subsystem. In *Proc. of MICRO 42*, MICRO 42, pages 337–346, New York, NY, USA, 2009. ACM.

[8] S. S. Iyer, J. E. Barth, Jr., P. C. Parries, J. P. Norum, J. P. Rice, L. R. Logan, and D. Hoyniak. Embedded DRAM: Technology platform for the Blue Gene/L chip. *IBM Journal of Research and Development*, 49(2.3):333 –350, march 2005.

[9] D. Keitel-Schulz and N. Wehn. Embedded DRAM Development: Technology, Physical Design, and Application Issues. *IEEE Design and Test of Computers*, 18:7–15, 2001.

[10] T. Kirihata. High performance embedded dynamic random access memory in nano-scale technologies. In K. Iniewski, editor, *CMOS Processors and Memories*, volume 0 of *Analog Circuits and Signal Processing*, pages 295–336. Springer Netherlands, 2010.

[11] T. Kirihata, P. Parries, D. Hanson, H. Kim, J. Golz, G. Fredeman, R. Rajeevakumar, J. Griesemer, N. Robson, A. Cestero, B. Khan, G. Wang, M. Wordeman, and S. Iyer. An 800-MHz embedded DRAM with a concurrent refresh mode. *Solid-State Circuits, IEEE Journal of*, 40(6):1377 – 1387, June 2005.

[12] LBNL/ICSI Enterprise Tracing Project. http://www.icir.org/enterprise-tracing.

[13] Micron, TN-47-16 Designing for High-Density DDR2 Memory Introduction, 2005.

[14] L. Minas and B. Ellison. The problem of power consumption in servers. *Intel Press Report*, 2009.

[15] C. Spear. *SystemVerilog for Verification, Second Edition: A Guide to Learning the Testbench Language Features*. Springer Publishing Company, Incorporated, 2nd edition, 2008.

[16] E. Strohmaier and H. Shan. Architecture Independent Performance Characterization and Benchmarking for Scientific Applications. In *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 467–474, Washington, DC, USA, 2004. IEEE Computer Society.

[17] E. Strohmaier and H. Shan. Apex-Map: A Global Data Access Benchmark to Analyze HPC Systems and Parallel Programming Paradigms. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 49–, Washington, DC, USA, 2005. IEEE Computer Society.

[18] J. Stuecheli, D. Kaseridis, H. C.Hunter, and L. K. John. Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory. In *Proc. of MICRO 43*, pages 375–384. IEEE Computer Society, 2010.

[19] M. Takahashi, T. Nishikawa, M. Hamada, T. Takayanagi, H. Arakida, N. Machida, H. Yamamoto, T. Fujiyoshi, Y. Ohashi, O. Yamagishi, T. Samata, A. Asano, T. Terazawa, K. Ohmori, Y. Watanabe, H. Nakamura, S. Minami, T. Kuroda, and T. Furuyama. A 60-MHz 240-mW MPEG-4 videophone LSI with 16-Mb embedded DRAM. *Solid-State Circuits, IEEE Journal of*, 35(11):1713 –1721, nov 2000.

[20] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi. A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 51–62, Washington, DC, USA, 2008. IEEE Computer Society.

[21] N. Wehn and S. Hein. Embedded DRAM architectural trade-offs. In *Design, Automation and Test in Europe, 1998., Proceedings*, pages 704 –708, feb 1998.

[22] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely. Quantifying Locality In The Memory Access Patterns of HPC Applications. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 50–, Washington, DC, USA, 2005. IEEE Computer Society.

[23] C.-W. Yoon, R. Woo, J. Kook, S.-J. Lee, K. Lee, and H.-J. Yeo. An 80/20-MHz 160-mW multimedia processor integrated with embedded DRAM, MPEG-4 accelerator and 3-D rendering engine for mobile applications. *Solid-State Circuits, IEEE Journal of*, 36(11):1758 –1767, nov 2001.

[24] NEC Electronics Embedded DRAM over 10 Gbps SONET/SDH and Ethernet takes off. http://www2.renesas.com/process/ja/pdf/eDRAM-SPI4.2.pdf.

APPENDIX

A. PROOF OF THEOREM 1 (NECESSITY)

PROOF. We construct an adversarial memory access pattern containing (at least) X idle slots in any Y consecutive time slots. Clearly, the Back-pressure module does not enforce any back-pressure for this pattern (since it satisfies the X no-conflict slots in every Y slots requirement). The adversarial pattern is as follows (see Figure 7 for an illustration). Up to time slot t_0, it consists of periodically X consecutive idle slots followed by $Y - X$ non-idle slots. The adversary blocks bank B_1 (by reading a row from it) in all non-idle slots up to time t_0. Choose t_0 sufficiently large such that the deficit counter has its maximum possible value at time t_0. It is immediate to see that the maximum value of D_c is $\lceil \frac{Y-X}{B-1} \rceil$ for $Y \leq BX$, and $X + 1$ for $Y > BX$ (see also Lemma 4). We can further assume that t_0 is the last time slot in a block of non-idle slots.

For a fixed (arbitrary) k, the adversary does not block any bank for time slots $t = t_0 + 1, \cdots, t_0 + m$, where $m = D_c(t_0) + B(R - k) + 1$. Note that the VR algorithm uses the idle slots $t_0 + 1, \cdots, t_0 + D_c(t_0)$ to reduce the deficit counter, and the bank pointer does not advance in these slots. Hence, the bank pointer advances by at most $B(R - k + 1) - 1$ in total over the interval $[t_0 - (B - 3), \cdots, t_0 + m]$. The banks different from B_1 are refreshed in this interval only when the bank pointer meets them. Since the bank pointer moves in a round robin fashion, there exists a bank (different from B_1), say B^*, which is refreshed at most $R - k$ times in this interval. In other words, there are at least k

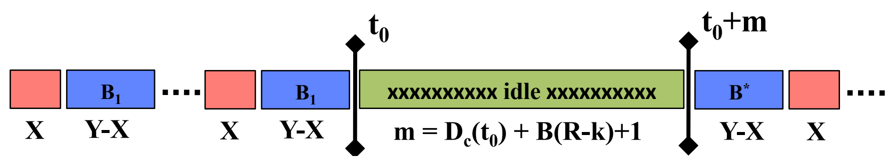


Figure 7: The adversarial memory access pattern in the proof of Theorem 1 (necessity part).

rows in bank B^* which are not refreshed in the interval $[t_0-(B-3),\cdots,t_0+m]$, and all of them have age at least $m+B-2$ at the end of the interval. Denote the set of these rows by A.

After time slot t_0+m, the input pattern again consists of periodically $Y-X$ consecutive non-idle slots followed by X idle slots, and the adversary blocks bank B^* in all subsequent non-idle slots. Therefore, the rows in set A are refreshed only at idle slots. We proceed by finding the largest subsequent interval with $k-1$ idle slots. Since A has at least k elements, one of its rows is not refreshed in this interval, whence we obtain a lower bound on W.

Let $k=k_qX+k_r$ with $1\le k_r\le X$. Note that the k^{th} idle slot arrives in $n(k)$ slots after t_0+m, where

$$n(k)=(k_q+1)Y-X+k_r. \tag{29}$$

Therefore, the interval $[t_0+m+1,t_0+m+n(k)-1]$ contains exactly $k-1$ idle slots, and one of the rows in A is not refreshed during this interval. Recall that all rows in set A have age at least $m+B-2$ at time slot t_0+m. Hence, for this row to be refreshed in time, we must have:

$$(m+B-2)+n(k)-1\le W-1. \tag{30}$$

The above inequalities must hold for any $1\le k\le R$. Plugging in (29) and rearranging, we obtain:

$$\begin{aligned}W\ge RB+Y-X+k_q(Y-BX)\\ +(B-1)(1-k_r)+D_c(t_0),\end{aligned} \tag{31}$$

for $(k_q,k_r)\in\Gamma$, where

$$\Gamma=\{(k_q,k_r):1\le k_r\le X,\ 1\le k_qX+k_r\le R\}. \tag{32}$$

It is easy to see that the right hand side of (31) is maximized for $(k_q,k_r)=(0,1)$ if $Y\le BX$, and $(k_q,k_r)=(a,1)$ if $Y>BX$. Using these values of k_q and k_r, and the value of $D_c(t_0)$, we obtain the expression for W_{VR}. $\square$

B. PROOF OF LEMMA 2

We prove the statement for the case $S_0=S_D$. The case $S_0=S_{ND}$ is similar. With $S_0=S_D$, the deficit is initially $C_0\ge1$ for bank $\tilde{B}$. Consider the following cases.

• **Case 1**: The f^{th} refresh to bank $\tilde{B}$, with $f\le X$, reduces its deficit to zero. (Note that $C_0\le f$). Assume this occurs at the t^{th} time slot of the epoch. In this case, the next time $\tilde{B}$ is met by BP, the system transits to $S_1=S_{ND}$. Let δ be the epoch duration. Firstly, note that although the deficit for bank $\tilde{B}$ is zero at the end of the epoch, some other bank may have non zero deficit, i.e., C_1. Secondly, the deficit counter did not reach $D_{MAX}=X+1$ during the epoch; otherwise, $f\le X$ refreshes were not adequate to reduce the deficit for bank $\tilde{B}$ to zero. Hence, we can apply Lemma 1 to the epoch. Using (9), the progress of the bank pointer during the epoch is:

$$P=\delta+C_1-C_0. \tag{33}$$

Also, using (10), we have:

$$f=p_{\tilde{B}}+C_0, \tag{34}$$

where $p_{\tilde{B}}$ is the number of times the bank pointer passes bank $\tilde{B}$. This implies $P\le(f-C_0)B+B-1$, since if $P\ge(f-C_0+1)B$, the bank pointer will pass $\tilde{B}$ at least $f-C_0+1$ times; i.e., $p_{\tilde{B}}\ge f-C_0+1$, which along with (34) leads to a contradiction. Therefore, using (33), we have:

$$\begin{aligned}\delta&=P+C_0-C_1\\&\le(f-C_0)B+B-1+C_0-C_1,\\&=fB-C_1+(B-1)(1-C_0),\\&\le fB-C_1,\end{aligned} \tag{35}$$

proving the desired result.

• **Case 2**: The deficit for bank $\tilde{B}$ remains positive, up to and after it receives its X^{th} refresh in the epoch. Then $S_1=S_D$ and the epoch ends right after the X^{th} refresh to $\tilde{B}$. Hence, $f=X$. Also, every no-conflict slot during the epoch is guaranteed to refresh $\tilde{B}$ because it has positive deficit. Since there are at least X no-conflict slots in every Y time slots, $\delta\le Y$. The result follows. $\square$

C. PROOF OF LEMMA 3

The proof is by induction on l. For $l=1$ the statement follows from Lemma 2. Assume it's true for $l\ge1$, we prove it for $l+1$.

Let δ be the duration of the $(l+1)^{th}$ epoch, and f be the number of refreshes to bank $\tilde{B}$ in this epoch. Consider the following two separate cases.

1)$\mathbf{Y\le BX}$: Clearly, $F_{l+1}=F_l+f$. Also, the values of the deficit counter at the beginning and the end of $(l+1)^{th}$ epoch are respectively C_l and C_{l+1}. Applying Lemma 2 in conjunction with the assumption $Y\le BX$ yields

$$\delta\le C_l\mathbb{1}\{S_l=S_{ND}\}+fB-C_{l+1}\mathbb{1}\{S_{l+1}=S_{ND}\}. \tag{36}$$

Therefore:

$$\begin{aligned}\Delta_{l+1}&\le\Delta_l+\delta\\&\le C_0+(F_l+f)B-C_{l+1}\mathbb{1}\{S_{l+1}=S_{ND}\}\\&=C_0+F_{l+1}B-C_{l+1}\mathbb{1}\{S_{l+1}=S_{ND}\}.\end{aligned} \tag{37}$$

2)$\mathbf{Y>BX}$: Let $f+\xi_l=uX+v$, with $0\le v\le X-1$. Note that $F_{l+1}=F_l+f=\psi_{l+1}X+\xi_{l+1}$, where $\psi_{l+1}=\psi_l+u$ and $\xi_{l+1}=v$. There are two cases:

- $S_{l+1}=S_D$: In this case $f=X$, $u=1$, $v=\xi_l$. Therefore: $\psi_{l+1}=\psi_l+1$, $\xi_{l+1}=\xi_l$. Also, using Lemma 2, $\delta\le C_l\mathbb{1}\{S_l=S_{ND}\}+Y$. Hence:

$$\begin{aligned}\Delta_{l+1}&\le\Delta_l+\delta\\&\le C_0+\psi_lY+\xi_lB-C_l\mathbb{1}\{S_l=S_{ND}\}\\&\quad+C_l\mathbb{1}\{S_l=S_{ND}\}+Y\\&=C_0+(\psi_l+1)Y+\xi_lB\\&=C_0+\psi_{l+1}Y+\xi_{l+1}B.\end{aligned} \tag{38}$$

- $S_{l+1}=S_{ND}$: In this case $\delta\le C_l\mathbb{1}\{S_l=S_{ND}\}+fB-$

C_{l+1}, and we have:

$$\begin{aligned}\Delta_{l+1} &\leq \Delta_l + \delta \\ &\leq C_0 + \psi_l Y + (\xi_l + f)B - C_{l+1} \\ &= C_0 + \psi_l Y + uBX + vB - C_{l+1} \\ &\leq C_0 + (\psi_l + u)Y + vB - C_{l+1} \\ &= C_0 + \psi_{l+1} Y + \xi_{l+1} B - C_{l+1}.\end{aligned} \tag{39}$$

In all cases, we have shown (14) holds for $l+1$. □

D. PROOF OF LEMMA 4

Consider the evolution of $D_c(t)$. Its value starts from zero and is always nonnegative. Assume an interval of time slots, $J = [t_0, t_1]$, such that $D_c(t_0) = 0$ and D_c is (strictly) positive for $t_0 < t \leq t_1$. Clearly, it suffices to prove the bound only for $t \in J$, since the timeline can be partitioned into so many of these intervals. We further assume that $t_0 = 0$, and $D_p = B_1$ during J, with no loss of generality.

For $0 < t \leq t_1$, let $I(t)$ be the number of no-conflict slots in $[1, t]$. In the following, we bound the number of increments and decrements to D_c in the interval $[1, t]$ separately, whence we obtain a bound on the value of $D_c(t)$.

Since $D.c$ is positive over the interval $[1, t]$, any no-conflict slot in this interval is used to reduce the deficit counter. Therefore, there are at most $t - I(t)$ slots in which D_c is potentially increased. To get a bound on the maximum increments to D_c, we track the bank pointer. Since the bank pointer does not advance in the no-conflict slots, we confine ourselves to the $t - I(t)$ slots. In these slots, D_c is increased by one every time the bank pointer meets B_1 and it is blocked. Since the bank pointer advances in a round robin fashion, it skips bank B_1 at most once in every $B - 1$ slots. Consequently, D_c is increased at most once in every $B - 1$ slots, implying that the maximum increments to D_c is $\lceil \frac{t-I(t)}{B-1} \rceil$. On the other hand, since D_c is always positive in the interval $[1, t_1]$, any no-conflict slot yields a decrease in the value of D_c. Accordingly, the value of counter at time t is bounded as follows.

$$D_c(t) \leq \left\lceil \frac{t - I(t)}{B-1} \right\rceil - I(t) = \left\lceil \frac{t - BI(t)}{B-1} \right\rceil. \tag{40}$$

Let $I(t) = k$, and write $k + 1 = k_q X + k_r$ with $1 \leq k_r \leq X$. Note that the $(k+1)^{th}$ no-conflict slot occurs by the $n(k+1)^{th}$ slot, where:

$$n(k+1) = (k_q + 1)Y - X + k_r. \tag{41}$$

This is because there are at least X no-conflict slots in every Y consecutive time slots. Hence, we have

$$t \leq n(k+1) - 1. \tag{42}$$

Otherwise, we get $I(t) \geq k+1$. Substituting for t and $I(t)$ in Eq. (40), we obtain

$$D_c(t) \leq \left\lceil \frac{(Y - X) + k_q(Y - BX) - (B-1)(k_r - 1)}{B-1} \right\rceil.$$

If $Y \leq BX$, the above bound is clearly at most $\lceil \frac{Y-X}{B-1} \rceil$. For the case $Y > BX$, we have the trivial bound $X + 1$ on $D_c(t)$ as set by the algorithm. The result follows. □

E. PROOF OF THEOREM 2

We propose two adversarial patterns each of which provides a lower bound on the worst case refresh overhead. Throughout, t_0 is sufficiently large such that all the rows have distinct ages from t_0 onward.

First pattern: At time slot t_0, consider the youngest row in each bank. Denote the oldest row among them by r^* and its bank by B^*. Clearly, $Age(r^*) \geq B - 1$. The adversary blocks bank B^* in the subsequent slots. By time $t_1 = t_0 + W - B + 1$, the algorithm must back pressure the memory at least R times. Otherwise, one of the rows in bank B^* is not refreshed in the interval $[t_0, t_1]$, and loses its data by time t_1. (Note that all the rows in bank B^* have age at least $Age(r^*) \geq B - 1$ at time slot t_0). Now, repeat this pattern starting at the first time slot $\hat{t}_0 \leq t_1$ that bank B^* is refreshed R times. At time slot $\hat{t}_0$, let r^* be the oldest row in the set of the youngest rows in each bank, and let B^* be its bank. By a similar argument to the above, the memory is back pressured at least R times within the next $W - B + 1$ time slots, and so on. Accordingly, the refresh overhead for this input read pattern is at least $R/(W - B + 1)$.

Second pattern: At time slot t_0, let r^* be the oldest row and let B^* be its bank. Therefore, $Age(r^*) \geq BR - 1$. The adversary blocks bank B^* in the following slots. By time $t_1 = t_0 + W - BR + 1$, the algorithm must back pressure the memory at least once. Otherwise, r^* loses its data. Now, repeat this pattern starting at the first time slot $\hat{t}_0 \leq t_1$ that row r^* is refreshed. At time slot $\hat{t}_0$, consider the oldest row, r^*, and its bank B^*. The adversary blocks bank B^* in the subsequent slots. Therefore, the memory is back pressured at least once within the next $W - BR + 1$ slots, and so on. Hence, the refresh overhead for this input pattern is at least $1/(W - BR + 1)$.

A refresh scheduling algorithm that ensures data integrity for any input pattern must satisfy both bounds, completing the proof. □