Refresh Triggered Computation: Improving the Energy Efficiency of Convolutional Neural Network Accelerators

SYED M. A. H. JAFRI, KTH Royal Institute of Technology
HASAN HASSAN, ETH Zürich
AHMED HEMANI, KTH Royal Institute of Technology
ONUR MUTLU, ETH Zürich

To employ a Convolutional Neural Network (CNN) in an energy-constrained embedded system, it is critical for the CNN implementation to be highly energy efficient. Many recent studies propose CNN accelerator architectures with custom computation units that try to improve energy-efficiency and performance of CNNs by minimizing data transfers from DRAM-based main memory. However, in these architectures, DRAM is still responsible for half of the overall energy consumption of the system, on average. A key factor of the high energy consumption of DRAM is the *refresh overhead*, which is estimated to consume 40% of the total DRAM energy.

In this paper, we propose a new mechanism, *Refresh Triggered Computation (RTC)*, that exploits the memory access patterns of CNN applications to reduce the number of *refresh operations*. RTC uses two major techniques to mitigate the refresh overhead. First, *Refresh Triggered Transfer (RTT)* is based on our *new* observation that a CNN application accesses a large portion of the DRAM in a predictable and recurring manner. Thus, the read/write accesses of the application inherently refresh the DRAM, and therefore a significant fraction of refresh operations can be skipped. Second, *Partial Array Auto-Refresh (PAAR)* eliminates the refresh operations to DRAM regions that do not store any data.

We propose three RTC designs (min-RTC, mid-RTC, and full-RTC), each of which requires a different level of aggressiveness in terms of customization to the DRAM subsystem. All of our designs have small overhead. Even the most aggressive RTC design (i.e., full-RTC) imposes an area overhead of only 0.18% in a 16 *Gb* DRAM chip and can have less overhead for denser chips. Our experimental evaluation on six well-known CNNs show that RTC reduces average DRAM energy consumption by 24.4% and 61.3%, for the least aggressive and the most aggressive RTC implementations, respectively. Besides CNNs, we also evaluate our RTC mechanism on three workloads from other domains. We show that RTC saves 31.9% and 16.9% DRAM energy for *Face Recognition* and *Bayesian Confidence Propagation Neural Network (BCPNN)*, respectively. We believe RTC can be applied to other applications whose memory access patterns remain predictable for a sufficiently long time.

CCS Concepts: • **Computer systems organization → Processors and memory architectures**; • **Hardware → Dynamic memory**.

Additional Key Words and Phrases: DRAM, DRAM Refresh Overhead, Convolution Neural Networks

ACM Reference Format:
Syed M. A. H. Jafri, Hasan Hassan, Ahmed Hemani, and Onur Mutlu. 2020. Refresh Triggered Computation: Improving the Energy Efficiency of Convolutional Neural Network Accelerators. *ACM Trans. Arch. Code Optim.* 1, 1, Article 1 (January 2020), 25 pages. https://doi.org/10.1145/3417708

Authors' addresses: Syed M. A. H. Jafri, KTH Royal Institute of Technology; Hasan Hassan, ETH Zürich; Ahmed Hemani, KTH Royal Institute of Technology; Onur Mutlu, ETH Zürich.

XXXX-XXXX/2020/1-ART1
https://doi.org/10.1145/3417708

1 INTRODUCTION

Neural Networks (NNs) are becoming a critically important class of mainstream machine learning algorithms, as they provide high prediction accuracy and are easily parallelizable [10, 62]. However, such benefits come at the cost of high computational power and intensive memory usage, which require high energy consumption. Convolutional Neural Networks (CNNs), a widely used type of NNs, try to reduce computation and memory usage by sharing *synaptic weights* in each layer of the neural network. Despite their relatively efficient design, CNNs still require a significant amount of energy. Furthermore, to process the information that is continuously received from various sensors, emerging autonomous systems, e.g., self-driving vehicles, typically require multiple simultaneously operating CNNs, which makes the energy consumed by CNNs even more important. Hence, achieving low-power CNN implementations remains as a challenging task.

As DRAM-based memory provides high capacity with decent latency, it is typically used as main memory in systems that implement CNNs. Although DRAM achieves high density by storing a single bit of data in the form of charge in a DRAM cell, data stored in DRAM is volatile due to charge leakage from the cell. To ensure data integrity, the charge of a cell needs to be periodically replenished by refresh operations. DRAM refresh consumes significant amount of energy and its overhead is expected to further increase in future DRAM devices as DRAM capacity increases [3, 9, 13, 24, 46, 48, 49, 51, 72, 73, 81, 83–85, 87–94, 114]. For example, Liu et al. [72] show that a single 4 *Gb* DDR3 DRAM chip spends 15% of the total DRAM energy for refresh operations and project refreshes to consume approximately half of the total DRAM energy in future 64 *Gb* DRAM chips. Thus, a DRAM device spends significant amount of energy only to ensure data is stored correctly, even during idle periods where no DRAM accesses occur.

CNNs typically have a large memory footprint [11], mainly due to a large number of synaptic weights that they maintain. Storing and accessing the synaptic weights from the DRAM constitute the dominant portion of energy consumption in CNNs [11]. To tackle this problem, recently-proposed accelerators focus on reducing the DRAM accesses by exploiting data locality [10, 11, 40, 102, 104]. Another approach compresses in-memory data to reduce the memory footprint and data transfer overheads of CNNs [104]. Although these approaches improve energy consumption by reducing DRAM accesses, a CNN accelerator still suffers from high DRAM refresh overhead. Figure 1 shows the energy breakdown of three well-known CNNs, AlexNet [62], LeNet [64], and GoogleNet [106], which are implemented on an architecture similar to the state-of-the-art Eyeriss [11] CNN accelerator.[1] The figure shows that the DRAM refresh overhead constitutes a portion as large as 15% for AlexNet and GoogleNet, which are examples of large CNNs, and 47% for LeNet, which is a relatively smaller CNN. For these evaluations, we assume 2 GB total DRAM capacity. For higher capacity DRAM, which is common in systems today, the refresh overhead is responsible for even larger portions of the overall DRAM energy consumption [72] (see Section 6.2). Thus, it is critical to investigate and develop techniques that reduce the DRAM refresh overhead for implementing energy-efficient CNNs.

Various mechanisms have been proposed to mitigate the DRAM refresh overhead. Du et al. [17] eliminate the refresh overhead by implementing a CNN accelerator using only SRAM-based memory. Such an approach not only restricts the applicability of the accelerator to small CNNs, as a majority of CNNs typically require significant memory capacity [11, 102, 104], but also increases the energy consumption for storing synaptic weights as SRAM has higher leakage power compared to DRAM with the same capacity. Smart Refresh [24] can reduce the refresh overhead by skipping the refresh operation for a recently-accessed row. However, to keep track of the time when a row was last accessed, SmartRefresh introduces additional storage overhead by employing a counter for each row. With the increase in DRAM capacity, the total storage required by the counters exceeds one megabyte, overshadowing the energy savings by reducing the number of refresh operations,

[1] Our methodology and accelerator architecture are described in Section 3.

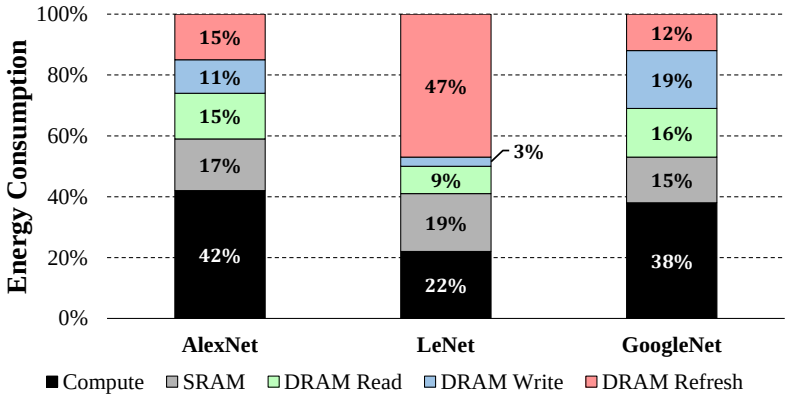


Fig. 1. Energy consumption breakdown of three CNNs on a modern CNN accelerator

as shown in [72]. There are also other works [1, 72, 95, 97] that propose mechanisms to reduce the DRAM refresh energy overhead. However, they have high implementation cost or limited applicability, as they require additional storage or can be applied only to embedded DRAM [1].

Our goal is to reduce the DRAM refresh overhead by eliminating the unnecessary refresh operations with minimal overhead in CNN accelerators. To achieve this, we propose a new technique that we call *Refresh Triggered Computation (RTC)*. In RTC, we take advantage of two *new* observations to develop two orthogonal mechanisms for reducing DRAM energy consumption by eliminating unnecessary refresh operations. First, we observe that large CNNs, such as AlexNet [62], access DRAM periodically, *with a fixed pattern*. As a read or write access implicitly refreshes the accessed DRAM cells, we can exploit the access pattern of such CNNs to overlap and replace read/write operations with the refresh operations. To this end, we propose and implement a new *Refresh Triggered Transfer (RTT)* mechanism to coalesce the read/write accesses with refresh operations. Second, we observe that smaller CNNs, such as LeNet, leave most of the DRAM capacity unused. We propose and implement *Partial-array Auto Refresh (PAAR)*, which eliminates the refresh operations to the portions of DRAM that are not used. We find that large CNNs typically benefit more from RTT than from PAAR, while the opposite is typically true for small CNNs that leave a large portion of DRAM unallocated.

In this work, we implement and evaluate three variants of RTC that differ in the level of customization required on the DRAM device and the memory controller. The first variant, min-RTC, requires changes only in the memory controller, and is useful when the read/write requests are frequent, such that they can be coalesced with the refresh operations. For the second variant, mid-RTC, we slightly modify the implementation of the already-available *Partial-array Self Refresh (PASR)* feature in modern DRAM chips [43, 80], to enable that feature not only in *self-refresh mode*, but also during normal operation of the DRAM. For the third variant, full-RTC, we propose internal DRAM modifications that fully exploit the capabilities of RTC. In particular, we add an Address Generation Unit and a Finite State Machine (FSM) to skip refreshes of recently accessed rows. In our evaluations, we find that RTC reduces the DRAM refresh energy by 25% to 96% across six different CNNs, depending on the used RTC variant, DRAM capacity, and the access pattern of the application.

Although we apply RTC to mainly CNNs in the scope of this paper, a wide class of applications with a *pseudo-stationary spatio-temporal memory access pattern* can take advantage of the RTC mechanism. RTC reduces DRAM refresh overhead when the memory access pattern of a workload is stationary for a time interval sufficiently long enough to reconfigure the RTC logic,

and otherwise remains inactive with negligible system performance and energy overhead. We believe that such long intervals with stationary memory accesses are prevalent for a wide variety of streaming applications (e.g., pattern recognition, signal processing, computer vision) that operate on large amounts of data. We demonstrate that multiple other applications, i.e., Face Recognition and Bayesian Confidence Propagation Neural Network (BCPNN), significantly benefit from RTC (Section 6.7). We hope that future work finds other use cases for RTC.

We make the following major contributions:

- We observe that the regular memory access patterns of CNNs can be exploited to reduce the DRAM refresh overhead by replacing periodic refresh operations with read and write accesses.
- We propose Refresh Triggered Computation (RTC) as a general technique to reduce the number of refresh operations based on applications memory access patterns. RTC includes two mechanisms: Refresh Triggered Transfer (RTT) for coalescing the read/write accesses with refresh operations, and Partial-array Auto Refresh (PAAR) for eliminating refreshes to portions of DRAM that are not being used.
- To improve the adoption of RTC, we implement three variants of it that differ in the amount of modifications required to the DRAM device and the memory controller. We evaluate refresh overhead reduction of all three variants for six widely used CNN applications (i.e., AlexNet [62], LeNet [64], GoogleNet [106], Winograd [63], ResNet [29], and Generative Adversarial Network [25]). We show that RTC, in its most aggressive variant, reduces DRAM refresh energy in a state-of-the-art CNN accelerator by up to 96% (on average 61.3% across multiple CNNs). We show that RTC is also effective for Face Recognition and Bayesian Confidence Propagation Neural Network (BCPNN) applications.

2 BACKGROUND

In this section, we provide background on DRAM and CNNs, necessary to understand the RTC framework that we propose. We refer the reader to past works in DRAM for more details [6–9, 22, 23, 26–28, 34, 36, 38, 47, 53–56, 58, 59, 65–69, 76, 99–101, 116, 117, 120].

2.1 DRAM Organization and Operation

Dynamic Random Access Memory (DRAM) offers high memory density at relatively low latency, which makes it the most preferable alternative for implementing main memory on mobile, desktop, and warehouse-scale systems. DRAM is also a viable option for CNN accelerators, as it provides enough capacity to fit large CNNs.

DRAM stores data in a hierarchical structure, as we show in Figure 2. As the smallest component of the hierarchy, a DRAM *cell* stores a single bit of data in a *capacitor* that is accessed by enabling the *access transistor* of the cell. As the cell capacitor leaks its charge over time, to correctly maintain the data, the capacitor needs to be *periodically refreshed*, commonly once every 64*ms*. Typically 2K to 16K cells are organized as a *row*, where all cells share the same *wordline* connected to their access transistors. Therefore, all cells in a row are refreshed simultaneously. The refresh operation involves the *sense amplifiers*, which are units that connect to the cells via *bitlines* and read the data out of the corresponding cells based on the charge amount their capacitors store, and correspondingly replenish the capacitor charge afterwards. As the area of a sense amplifier is much higher than that of a DRAM cell [68], a large number of cells from different rows share the same sense amplifier to provide high memory density. However, as having an extremely large number of rows that share a sense amplifier would negatively affect the access latency due to increased parasitic capacitance on the bitline, the rows are grouped into multiple *banks*, where each bank has its own set of sense amplifiers, referred to as *row-buffer*. Besides improving access latency, a banked structure also improves the memory throughput by providing parallelism at bank-level (i.e., multiple banks can operate simultaneously as they have separate row-buffers). Finally, at the top level of the hierarchy, multiple chips are organized as a *rank*, where the chips operate in lock step (i.e., perform the same

operation concurrently). There might be one or more ranks per *channel*. In the latter case, multiple ranks share the same memory bus to interface with the processor, reducing I/O pin requirements, but limiting parallelism.

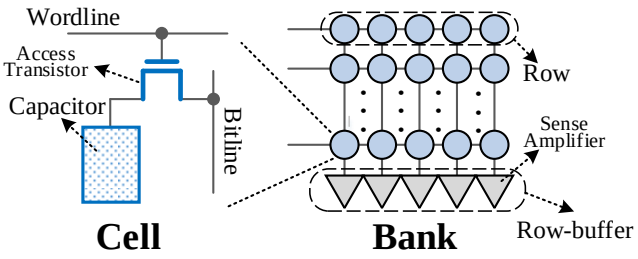


Fig. 2. DRAM cell and bank

DRAM commands are the interface between the memory controller and DRAM. There are four main DRAM commands involved in a DRAM access. First, to service a *demand request* (i.e., a load or store request) the memory controller issues an *Activate (ACT)* command to select a row from a bank, and copy its data to the row-buffer. After completion of that operation, the memory controller can issue multiple *READ* and *WRITE* commands to access the data in the row-buffer at a granularity equal to the data bus width of the DRAM chip. In order to access data from another row in the same bank, the memory controller first closes the currently active row by issuing a *Precharge (PRE)* command.

In addition to these four commands used to access DRAM, the memory controller also periodically issues a *Refresh (REF)* command to replenish the charge stored in DRAM cell capacitors and ensure data integrity. For the chips available in the market today, the entire DRAM chip has to be refreshed every 64 *ms* [42] (or 32 *ms* when operating at temperatures exceeding 85 °C [42]). As there is a large number of rows in the chip, the memory controller issues a refresh command once every 7.8 *us* to complete the refresh cycle for the entire DRAM in 64 *ms*. A single refresh command typically refreshes multiple rows in batch in hundreds of nanoseconds.[2] DRAM refresh consumes significant amount of energy and its overhead is expected to further increase in future DRAM devices as DRAM capacity increases [3, 9, 13, 24, 46, 48, 49, 51, 72, 73, 81, 83, 87, 92–94, 114]. For example, Liu et al. [72] show that refreshes constitute 15% of the total DRAM energy for a 4 *Gb* DDR3 chip and the fraction of DRAM energy spent on DRAM refresh is projected to increase as DRAM chips become denser (e.g., refreshes would consume about 50% of the total DRAM energy in future 64 *Gb* DRAM chips). Additionally, although a DRAM device may not be always accessed with maximum throughput while executing a workload, all DRAM rows have to be refreshed at a constant rate. Thus, when DRAM is accessed infrequently, energy spent on DRAM refresh accounts for a significant portion of the overall DRAM energy. We observe that typical CNN workloads access DRAM regularly but not frequently enough such that refresh operations consume significant DRAM energy compared to the energy consumed by DRAM accesses.

An ACT-PRE command pair, which the memory controller issues to service a demand request, also fundamentally performs the same operation as refresh. Both, first transfer the charge stored in the capacitor to the sense amplifier, which later fully restores the capacitor back to its original level (i.e., fully-charged or empty). As a result, both refresh and demand requests have the ability to replenish the charge stored in the DRAM cells. We exploit this observation in the design of our mechanism to save DRAM refresh energy.

[2]For an 8 Gb DDR3 chip, a DRAM refresh command takes 350 ns to complete, during which all banks are unavailable for access [9, 42].

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [64] are machine learning algorithms that achieve state-of-the-art learning accuracy. The basic idea of CNNs is to extract low-level features from the input data at high resolution, and later combine those features to build more complex ones.

As we show in Figure 3, a CNN consists of multiple layers, which contain feature maps at different abstraction levels of the input data, and synaptic weights (i.e., convolutional kernels), which are used for extracting the features of the next layer by performing convolution on the output of the previous layer. There are two main computational phases in a CNN: *training* and *inference*. During the training phase, to learn what to infer from the input data, the CNN processes a large amount of reference data using error back-propagation [98]. Later, during the inference phase, the CNN classifies the input data by using the information that it has learned during the training phase. In general, it is sufficient to perform the training phase offline, before the inference phase [10, 102]. Since the offline training does not affect the performance of the end-application, we focus on the inference phase, similar to prior work [10, 11, 102, 104]. However, we observe that the training phase exhibits similar memory access patterns as the inference phase, and thus the techniques we propose can also be applied to the training phase.

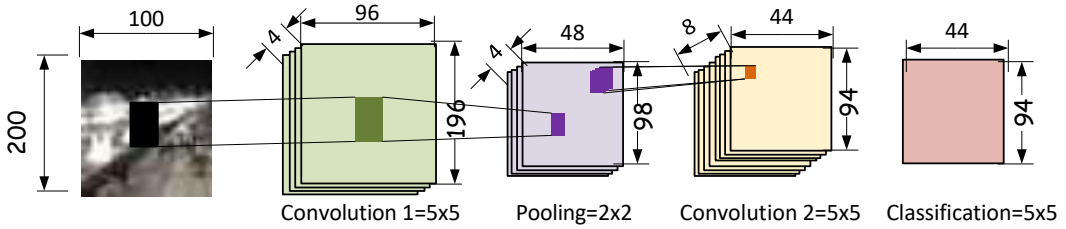


Fig. 3. The general structure of a CNN

The high-level goal of the inference phase is to infer the required information (e.g., whether a particular object is available in the image) from raw input. CNNs have multiple layers between the input data and the final classification output. Primarily, there are three types of layers: (i) *convolution*, (ii) *pooling*, and (iii) *classification* layers. The convolutional layer extracts various features (e.g., edges and corners) by convolving a 2D mask (of synaptic weights) with the input data from the previous layer. For each feature that is being extracted, the CNN applies a different set of synaptic weights to the input, producing multiple feature maps. For example, in Figure 3, the first convolutional layer (Conv1) produces four output feature maps by convolving the input image with a 5x5 mask. The pooling layer extracts the salient features from the previous layer, usually by applying a max or averaging function. After several layers of convolution and pooling, the input image is classified in the classification layer, which provides the probability that the input belongs to a particular class.

The inference phase of the CNN is largely memory intensive. When processing an input image, the CNN needs to read the large data (i.e., synaptic weights and outputs of previous layer) of each layer from the memory. For each layer, the CNN runs multiple convolution or pooling operations and writes back the results to memory. Thus, the inference phase yields a large read and write traffic that could not be entirely filtered out by the caches, and requires the data to be serviced from DRAM. For example, AlexNet [62] performs about 3 billion DRAM accesses when processing a single image. Modern CNN accelerators [11] reduce this requirement to 60 million DRAM accesses per input image by exploiting data locality. However, despite that huge reduction, DRAM is still major contributor to the overall energy consumption of a system, as we see in Figure 1.

3 REFRESH TRIGGERED COMPUTATION

As we explained in Section 2.1, the memory controller periodically issues refresh commands to DRAM, in order to ensure data integrity. Such frequent and time-consuming refresh operations often conflict with read and write requests that are issued by the workloads running on the system [9, 72]. As a result, a refresh operation not only consumes significant amount of energy, but also negatively affects system performance by delaying read and write requests.

Refresh Triggered Computation (RTC) is based on the high-level observation that for applications with regular memory access patterns, such as CNNs, it is possible to synchronize the refresh operations and the read/write requests, such that read/write requests of the application naturally refresh DRAM. RTC not only eliminates conflicts between refresh and read/write, but also reduces the number of refresh commands that the memory controller needs to issue by eliminating redundant refresh operations. In this section, we first introduce the RTC concepts before elaborating on RTC's implementation details in Section 4.

3.1 Making Refresh Unnecessary

In Section 2.1, we explain that both refresh and access requests perform similar operations (i.e., activating and precharging a row) in the DRAM circuitry that replenish the charge of the DRAM cells in a row. We observe that, in many cases, the *explicit* refresh operations can be eliminated, since i) the DRAM access requests are at least as frequent as the periodic refresh operations and ii) such requests continuously cover a very large portion of the DRAM. Hence, there is potential to eliminate most of the *explicit* refresh operations since a large fraction of the DRAM is already being *implicitly* refreshed when accessed.

We aim to make refresh unnecessary by ensuring that the row to be refreshed is accessed at the same time it is supposed to be refreshed. However, performing such an alignment is not straightforward due to two reasons. First, the periodic refresh operation is performed using an in-DRAM counter that points to the next row to be refreshed. Thus, an application (or even the memory controller) does *not* have control on *which* row will be refreshed next. Second, the access requests are *not* as regular as the refresh operations, in terms of their row access pattern. Therefore, aligning refresh operations with accesses is a challenging problem.

3.2 Alignment in a Controlled Environment

To develop a feasible and efficient solution for the problem of aligning the access requests with the periodic refresh operations, we first make three simplifying assumptions. *i)* We assume that the access pattern of the application is known in advance and it is periodic. In other words, the application has an iterative execution flow and, in each iteration, it generates requests in a fixed order. *ii)* The period of the access requests is lower than (or same as) the period of the refreshes. This assumption ensures that the refresh period (e.g., 64ms) of a DRAM row is not exceeded between two consecutive accesses to the row. *iii)* We assume that the entire working data set of the application is accessed in each iteration. In Sections 3.3 and 3.4, we introduce our techniques to handle the cases where these assumptions do *not* hold true.

In the process of making refreshes unnecessary, we first design a scheme that aligns refreshes with reads when the three assumptions about the applications access pattern hold. In Figure 4, we explain how such a scheme works by plotting a timeline of accesses that an application performs and refreshes that the memory controller issues during three refresh periods. The refresh requests iterate through rows r_1 to r_4 in the first two refresh periods. Close to the end of the first refresh period, the application starts to issue access requests to all of these four rows, but in different order. In the second period, the refresh operations are still required to ensure data integrity because if we eliminate the refresh operations, the time since r_1 was last refreshed would exceed the refresh period. In contrast, all refresh operations in the third period are redundant as the rows are already refreshed due to the accesses in the same period. Next, we introduce our techniques to align refreshes and access request when the three simplifying assumptions are relaxed.

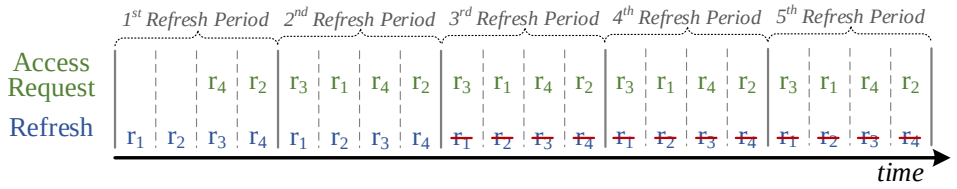


Fig. 4. Periodic application access pattern vs. refresh pattern.

3.3 Refresh Triggered Transfer

The simple scheme we proposed in Section 3.2 assumes that the rate of the refresh operations and accesses always match. However, in real applications, the access requests can be more or less frequent than the refreshes.

To solve the problem of matching the rates of accesses and refreshes, we propose *Refresh Triggered Transfer (RTT)*. The key idea of RTT is to alter the existing periodic refresh scheme to align the refreshes with the access requests. We achieve this by slightly modifying the DRAM auto-refresh circuit, as we explain in Section 4.3.

Algorithm 1 describes how RTT handles the mismatch in the rates of accesses and refreshes.[3] If the application generates access requests to its entire allocated memory as frequently as the refresh rate or faster, RTT completely removes the refresh overhead and ensures data integrity as an access (i.e., DRAM row activation and precharge) replenishes the charge of a row it accesses. However, when the accesses are not frequent enough, the problem of matching the rates of the accesses and refreshes becomes more challenging. To tackle this problem, RTT eliminates refreshes *partially* by performing refresh only on rows that are *not* accessed within the refresh period.

Algorithm 1 takes N_a and N_r as input[4], which are the number of rows that the access requests and refreshes target during a single refresh period, respectively. The output of the algorithm is the explicit refresh (exp_ref) signal, which determines whether a row will be explicitly refreshed or implicitly replenished when accessed to read/write data. Thus, when $N_r \leq N_a$, exp_ref is set as 0 to indicate that an access occurs to all rows frequently enough (line 4). When the opposite is the case, i.e., $N_r > N_a$, then the algorithm needs to output additional refresh operations to compensate for the rows that are *not accessed* during the refresh period. To find which rows to refresh using explicit refresh requests, the algorithm starts with a credit c, equal to N_r (line 7). For each implicit refresh, c is reduced by $N_r - N_a$, until the credit becomes less than $N_r - N_a$. At this point, the algorithm signals $exp\_ref = 1$ to indicate an explicit refresh, and increments the credit by N_a.

To understand how Algorithm 1 operates, consider an example where $N_a = 2$ and $N_r = 4$. Within a refresh period, only half of the rows will be refreshed using an explicit refresh operation, as we illustrate in Figure 5. Initially, $P = 1$ and $c = 4$ (lines 6-7). In the first iteration of the loop (line 8), c is greater than $N_r - N_a = 2$. Thus, the row is implicitly refreshed, and the credit is decreased (lines 10-11). In the next iteration, as the credit is not greater than $N_r - N_a$, an explicit refresh will be triggered (line 13). Thus, the algorithm will interleave between an implicit and an explicit refresh operation. We implement the RTT scheme in DRAM with minor modifications to existing circuitry as we explain in Section 4.

Generating Memory Access Patterns. The existing refresh scheme implements a counter in the DRAM chip to refresh the rows with a fixed pattern. However, the access pattern of a real application may not follow the same pattern as the refreshes. To adapt the refresh scheme to arbitrary access patterns, RTT implements an *Address Generation Unit (AGU)* that is similar to the proposal in prior work [21]. AGUs are commonly used in Digital Signal Processors (DSPs) to efficiently generate the memory addresses to feed to the functional units [21, 70, 71, 107, 112, 113]. An AGU can typically be programmed to generate various address sequences for a given application.

[3] We adapt the algorithm from a technique [5, 41] that is used to align send and receive processes operating at rationally related clock frequencies.

[4] N_r is equal to the number of rows in DRAM, as the entire DRAM needs to be refreshed in a single refresh period.

Algorithm 1 Rate matching algorithm

```
▷ N_a: the number of rows accessed by read/write during a refresh period
▷ N_r: the number of rows refreshed during a refresh period
procedure RateMatching(N_a, N_r)
    for every refresh period do
        if N_r ≤ N_a then
            exp_ref ← 0                                   ▷ implicit refresh
        else
            P ← N_r / gcd(N_r, N_a)
            c ← N_r
            for i ← 1, P do
                if c > N_r − N_a then
                    exp_ref ← 0                           ▷ implicit refresh
                    c ← c − (N_r − N_a)
                else
                    exp_ref ← 1                           ▷ explicit refresh
                    c ← c + N_a
                end if
            end for
        end if
    end for
end procedure
```

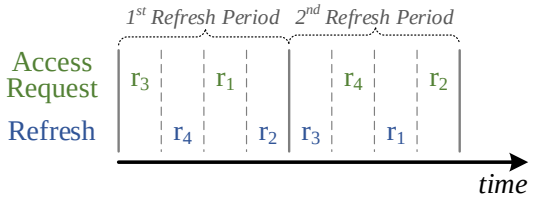


Fig. 5. Accesses and refreshes generated by RTT for $N_r = 4$ and $N_a = 2$

Prior works propose a broad range of AGU designs that can generate address sequences with various amounts of complexity (e.g., commonly used DSP addressing modes such as bit-reverse and circular buffer addressing [33, 118], piece-wise affine address pattern generation [32], two-dimensional affine address generation [79], complex addressing modes that include multiplication, modulo, and shift [108]). We observe that the memory access patterns of the workloads we focus on in the scope of this work are relatively regular, and thus, to keep the design simple, we adopt an AGU design that can generate address sequences based on an arbitrary affine function. We explain the details of AGU's implementation in Section 4.3.

3.4 Partial-Array Auto Refresh

For many applications, a significantly large portion of the DRAM may not always be in use (i.e., portions may be unallocated). For example, the memory footprint of LeNet [102], which is a small CNN, is only $1.06MB$ (e.g., when 100*100 image is used for character recognition). Hence, depending on the DRAM capacity, a large number of unallocated DRAM rows would unnecessarily be refreshed, consuming significant energy. In RTC, we implement a technique, *Partial-Array Auto Refresh (PAAR)*, which ensures that refreshes are generated only for rows that are allocated.

PAAR should not be confused with a technique called *Partial-Array Self Refresh (PASR)* [43], which already exists in low-power DRAM chips and is used to refresh only certain DRAM banks while in self-refresh mode (i.e., power-saving mode in which DRAM cannot be accessed). As PASR operates at coarse bank granularity, no data should be allocated in an entire bank that PASR turns refresh off. PAAR differs from PASR mainly in two ways. First, to enable PASR, the memory controller needs to switch the DRAM to a special low-power mode. Besides switching in and out of this mode is a relatively slow process [78], another downside of PASR is that the DRAM cannot serve access requests while in PASR mode. In contrast, PAAR can be enabled during the normal operation of the DRAM. Second, PASR can eliminate refreshes only at bank-granularity. In order to eliminate refreshes using such a scheme, an entire bank should be unallocated. Leaving one or more banks out of data allocation limits bank-level parallelism [78, 86], and reduces the memory bandwidth. In contrast, PAAR operates at row-granularity and thus provides a more practical scheme to eliminate redundant refreshes compared to PASR.

3.5 Limitations of RTC

Our RTC framework has two limitations.

Access Patterns. RTC can eliminate redundant refresh operations when the access pattern of an application is stationary for sufficiently long time. Configuring the AGU of RTC can take approximately 100 cycles. To compensate for this latency overhead, the access pattern of the application should not change very frequently. Fortunately, there are many applications from different domains (e.g., signal processing, neural networks, bioinformatics) that exhibit regular access patterns. In this work, we expect the programmer to determine the memory access pattern of an application. However, a profiling-based or compiler-assisted approach can potentially be used to automatically determine access patterns of applications and take advantage of RTC without involving the programmer. We leave this study to future work. For other applications that have frequently changing access patterns, RTC can be disabled to operate DRAM in the conventional way with negligible performance and energy overhead.

Simultaneously Running Applications. Even though two different applications have regular access patterns, running them simultaneously on the same system may lead to irregularity in the memory access pattern. To support multiple applications, we propose to map applications to separate DRAM banks or channels, each with its own RTC control logic. Note that such an approach does not reduce the bank-level parallelism, since all banks continue to receive memory requests, but from different applications. In fact, prior work shows that partitioning the applications to separate banks or channels improves overall system performance by reducing the bank/channel conflicts [44, 74, 82].

4 THE RTC ARCHITECTURE

In this section, we present the RTC architecture, which implements the concepts we introduced in Section 3. We propose three variants of RTC, differing in the level of customization that they require. First, *Min-RTC* does not require any changes to the DRAM chip, but it only slightly changes the memory controller. Second, besides the changes to the memory controller, *Mid-RTC* also introduces minimal modifications the DRAM peripheral logic. Third, our most aggressive implementation, *Full-RTC*, exploits the full potential of the RTC concepts.

4.1 Min-RTC

For this implementation, we restrain ourselves from making any changes to the DRAM chip. By modifying only the memory controller, we can implement RTT partially and cannot implement PAAR at all. Thus, *Min-RTC* is only useful when the accesses are more frequent than the refreshes such that all refresh operations can be eliminated.

With min-RTC, the memory controller receives information about the access period directly from the application. Based on the information, the memory controller decides whether to operate

in *normal* or *min-RTC mode*. If the application accesses the memory slower than the refresh rate, the memory controller disables min-RTC, and operates in normal mode. Otherwise, it enables min-RTC to eliminate the overhead of the refresh operations. To achieve this, first, the memory controller aligns the accesses with the refreshes as we describe in Section 3.2. Later, the memory controller stops issuing refresh commands to DRAM, as the access requests implicitly refresh DRAM. The memory controller disables min-RTC when the application completes execution or another application is invoked. According to our evaluations (Section 6.1), even such a simple mechanism saves significant energy.

4.2 Mid-RTC

In mid-RTC, besides the changes required for min-RTC, we also apply minor modifications to the DRAM control logic to enable a coarse-grained (bank-granularity) implementation of PAAR. Particularly, we modify the logic that enables PASR, which is already available in low power DRAM chips [110], but is used only when the DRAM chip is in low-power stand-by mode. To enable PAAR, we reuse the PASR logic and make it possible to activate even when the DRAM is in normal mode of operation. In mid-RTC, we avoid adding additional registers to define the range of rows that will be refreshed with PAAR, and thus PAAR operates at bank-granularity in this implementation.

Mid-RTC can mitigate the refresh overhead by eliminating unnecessary refreshes, as min-RTC does, and by disabling the refreshes for the DRAM banks that do not have any allocated portions.

4.3 Full-RTC

As we show in Figure 6, the most aggressive implementation, full-RTC, requires mainly three modifications in the DRAM chip and the memory controller. ❶ To prevent a subset of non-allocated DRAM rows from being refreshed, full-RTC modifies the in-DRAM refresh logic to be configurable by the memory controller. ❷ To *fully* implement the RTT scheme as described in Section 3.3, full-RTC adds an *Address Generation Unit (AGU)*, which is implemented in two levels (i.e., *Row AGU* ❷a and *column AGU* ❷b). An application can configure the AGU at runtime to generate access and refresh requests using an arbitrary affine function. ❸ Full-RTC implements *RTC Frontend Controller* to enable reconfiguration of the AGUs and the *refresh counter*; and executes Algorithm 1 to determine which addresses generated by the AGU will transfer data from/to DRAM, and which will only refresh the corresponding row. Full-RTC implements this algorithm in the memory controller (i.e., in the RTC Frontend Controller) but it also introduces a small modification to the DRAM command decoder to handle the explicit refresh (exp_ref) signal generated by the RTC Frontend Controller. We explain our design in more detail.

4.3.1 Modifications to the Memory Controller. The memory controller is the interface between the accelerator/processor and DRAM. We modify the memory controller to support our changes in the DRAM architecture that enables full-RTC.

In full-RTC, applications need to provide their memory access patterns to the RTC Frontend Controller, which reconfigures the AGU and the *refresh counter*. Once the RTC Frontend Controller completes reconfiguring the AGU, the AGU starts generating DRAM row and column addresses to access data according to how the application has configured the AGUs.

The address generation unit (AGU) incorporated inside the RTT counter logic can be configured with an arbitrary affine function to generate various memory access patterns that applications typically exhibit. In our implementation, the memory controller uses special commands to configure the AGU.

4.3.2 Modifications to the DRAM chip. PAAR improves DRAM energy efficiency by eliminating the refresh operations to DRAM regions that are not allocated. In conventional DRAM, periodic refresh operations are performed on all DRAM rows with a fixed pattern. We slightly modify the conventional control logic for the periodic refresh operations to limit the refreshed address range

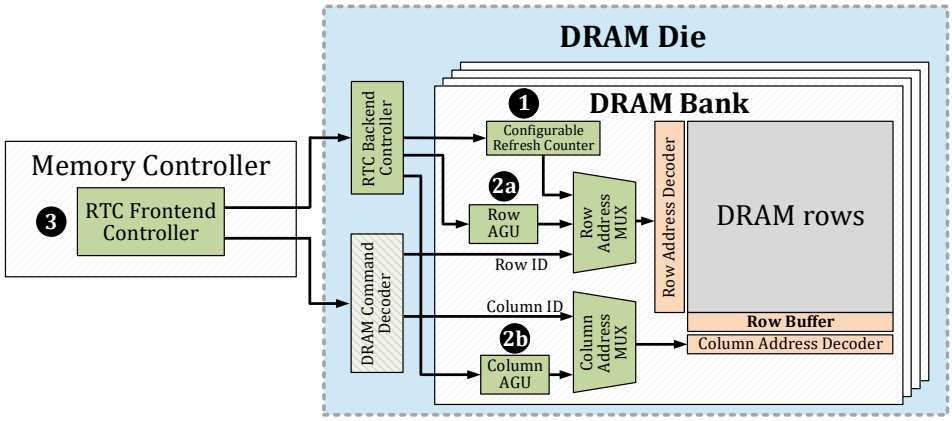


Fig. 6. Modifications in the memory controller and DRAM to support Full-RTC

such that only a specific region of DRAM is refreshed. As we illustrate in Figure 6, we implement this feature by introducing a *Configurable Refresh Counter*, which incorporates a register for start and end row addresses of the region to refresh. The *RTC Backend Controller* provides an interface to the memory controller to configure start and end addresses of the DRAM rows to refresh.

4.3.3 RTC Controller Operation. The RTT technique aligns memory accesses with refresh requests such that explicit refresh operations can be eliminated as accesses already implicitly refresh DRAM rows. To achieve the alignment of accesses and refreshes, the RTC Frontend Controller implements Algorithm 1 that we explain in Section 3.3. By running the algorithm, the RTC Frontend Controller determines whether DRAM should perform an access using the next address generated by the AGU or a refresh operation using the refresh counter.

In Figure 7, we describe the operation of the *RTC Frontend Controller* using a state diagram. During the initial *idle* state, the *RTC Frontend Controller* expects signals for reconfiguring one of its three components (shaded with different colors). Once reconfigured, it transitions into the *Active* state, where RTT is enabled (we describe operation in *Active* state in Figure 8).

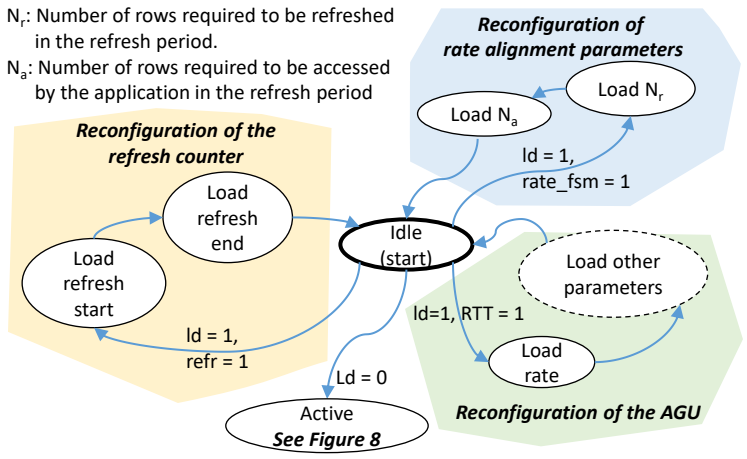


Fig. 7. Operation of the RTC Frontend Controller

To enter a reconfiguration state, the *load* signal (*ld*) has to be asserted along with one of the three signals that indicate which of the reconfiguration states to enter. First, when `refr=1`, the *RTC Controller* reconfigures the start and end row addresses of the *Configurable Refresh Counter*. Second, when `rtt=1`, the *RTC Controller* reconfigures the Row and Column AGUs. Third, when `rate_fsm=1`, the *RTC Controller* reconfigures the N_a and N_r parameters that we describe in Section 3.3.

In Figure 8, we show a diagram that describes the operation of RTT. While RTC reconfiguration is in progress, the *CKE* signal remains low to keep RTT in *idle* state. After reconfiguration finishes, the memory controller starts RTT operation by setting `CKE` and `ld` to 0.

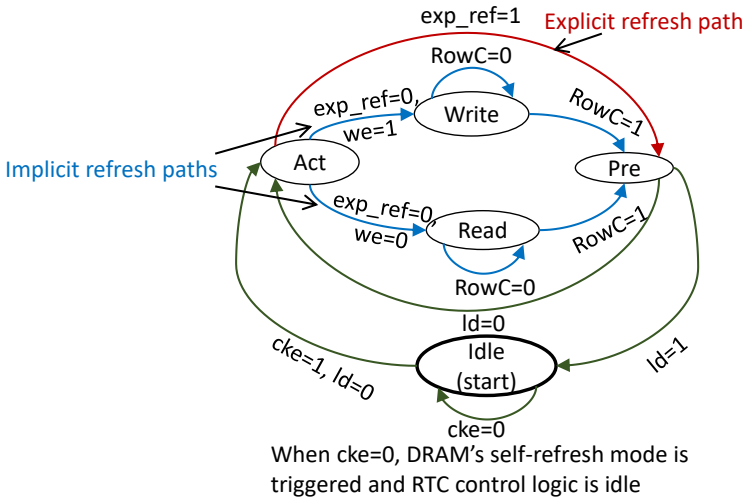


Fig. 8. State machine that describes full-RTC operation

In the *Act* state, the *RTC Backend Controller* generates a DRAM command either to activate the row at the address that the Row AGU provides or to refresh the row that the *Configurable Refresh Counter* points to depending on the exp_ref signal sent by the memory controller. First, if the time interval between two consecutive read/write requests is greater than the required refresh interval (i.e., the memory controller sends $exp\_ref = 1$), the RTC Backend controller *explicitly* refreshes the row that the *refresh counter* points to. We show this in Figure 8 with a red line from the *Act* to the *Pre* state. During this state transition, the memory controller issues a precharge command to close the open row. Second, when $exp\_ref = 0$, control is transferred to either the *Read* or the *Write* state depending on whether the write enable (*we*) signal is set to 1 or 0. This is because if the read/write path is taken, the rows are implicitly refreshed. RTT remains operational as long as `ld=0`. When `ld=1`, the control returns to the *idle* state in Figure 7, which allows the RTC to be reconfigured.

5 METHODOLOGY

We implement the RTC framework on a system that consists of a LEON3-based open-source processor, which is connected to a state-of-the-art CNN accelerator MOCHA [40], similar to Eyeriss [11], via an AMBA AHB bus [2]. As we illustrate in Figure 9, the accelerator is implemented in the logic-layer of a DRAM-based 3D-stacked memory. We evaluate DRAM capacities of 16 *Gb*, 32 *Gb*, and 64 *Gb*. The CNN accelerator has a private 108KB scratch-pad memory, as in Eyeriss [11], and it also incorporates a memory controller to interface the upper DRAM layers of the 3D-stacked memory. The Eyeriss architecture uses row-stationary dataflow, which aims to maximize reuse

of filter weights and feature maps in the processing engines' local storage to minimize DRAM accesses.

To analyze the effectiveness of RTC at saving DRAM refresh energy, we evaluate six widely-used CNN applications, GoogleNet [106], AlexNet [62], LeNet [64], Winograd [63], ResNet [29], and Generative Adversarial Network (GAN) [25]. We adjust the batch size for each CNN individually depending on how many kernels we can accommodate at most in the register files of the MOCHA accelerator.

Winograd is an algorithm for efficiently performing convolution operations. Winograd promises reduction in multiplication operations but it does not affect main memory access characteristics. Reducing the number of multiplications is beneficial for architectures such as GPUs and FPGAs that perform convolutions as matrix multiplication. However, custom CNN accelerator architectures already implement hardware optimizations to perform convolution efficiently, and therefore using Winograd has negligible impact on performance and energy consumption of CNNs in the system we model. We implement Winograd on only AlexNet but we expect Winograd to have limited benefits when used with other CNNs in our system.

We evaluate each CNN with two different use cases: 1) a real-time video application that requires 30 frames per second (fps), and 2) a robotic vision application that requires 60 fps. Thus, in our evaluation, the accelerator in the system we model invokes CNN inference either at 30 fps or 60 fps, and we do not have any other performance requirements. Because of this, although RTC can improve system performance by eliminating a significant fraction of refresh operations and perform more accesses instead, we do not quantitatively evaluate potential performance benefits of RTC in the scope of this work.

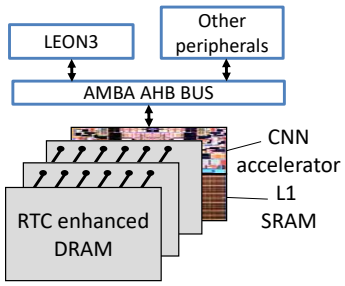


Fig. 9. System-level view of the proposed architecture

Tools, Technology, Area, and Energy Models. We use commercial EDA tools for all of our designs. We synthesize our designs to run at 200 MHz frequency using the 40 nm technology node for both CMOS and DRAM logic. To quantify memory controller area and energy overhead of the RTC logic, we use a Micron-compatible DRAM controller available from Gaisler [14] as the baseline. We extend this controller as we discuss in Section 4.3.1. We report area and energy overheads based on post-layout data. The energy estimation for the CMOS logic is based on gate-level simulation, back annotated with post-layout data. To quantify area and energy overheads in DRAM, we use the Rambus DRAM model [115] for different DRAM dimensions and traces of access patterns.

We create three different datapaths, one for each of the three variants of RTC. For the full- and mid-RTC, we modify the DRAM peripheral logic to reflect the RTC-enabled DRAM datapath. For both models, we use technology parameters for 40 nm DRAM from ITRS [37]. By supplying the Rambus model a trace of operations, in terms of activate, read, write, and precharge, the Rambus model provides the energy numbers. We generate traces using an in-house simulator [40] for the workloads we evaluate.

6 EVALUATION

In this section, we analyze DRAM energy savings and area overhead of each variant of RTC compared to conventional low-power DRAM, LPDDR4 [43]. We evaluate six different workloads in total. These include various CNNs (i.e., AlexNet (AN), LeNet (LN), GoogleNet (GN), and ResNet50 (RN)), a Generative Adversarial Network (GAN), and Winograd, which is an optimization for performing faster convolution in CNNs. All these workloads vary in their memory footprints and memory access patterns. We evaluate these workloads on systems with different DRAM capacities. Furthermore, we provide a breakdown of the benefits of the RTT and PAAR techniques that are part of the RTC framework.

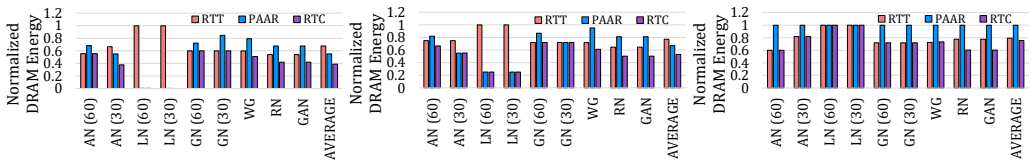
6.1 Energy Savings on Different Workloads

We evaluate the DRAM energy savings of the three different implementations of RTC on six CNN workloads in comparison to standard low-power 16 Gb LPDDR4 DRAM.

Full-RTC. Figure 10a plots DRAM energy with full-RTC, normalized to the baseline DRAM with conventional refresh. We break down the individual benefits of the RTT and PAAR techniques that RTC combines. The DRAM energy savings of RTT primarily depend on how well the DRAM refresh and access rates match: the closer they match, the greater is the energy reduction. On average, RTT saves 32.3% DRAM energy across all workloads. RTT saves more DRAM energy at 60 fps than at 30 fps because running inference on the CNN more frequently results in a larger number of DRAM accesses at 60 fps, which in turn creates more opportunity for the accesses to align with the refreshes, and thus makes the refreshes redundant. At 30 fps, the DRAM access rate is almost the half of at 60 fps and the refresh rate remains the same, which results in an insufficient number of DRAM accesses to cover all DRAM rows that contain workloads' data before the 64 ms refresh period. Therefore, at 30 fps, the memory controller needs to issue more explicit refreshes that come with DRAM energy cost. Specifically, for LeNet, the effectiveness of RTT is minimal because of the small memory footprint and fewer read/write DRAM accesses of this workload.

The PAAR technique saves DRAM energy by eliminating refreshes to DRAM regions that are not allocated. Therefore, PAAR significantly favors low-memory-footprint workloads, such as LeNet. PAAR alone saves 96% DRAM energy when running LeNet, as LeNet's working data set mostly fits into accelerator's on-chip memory, and DRAM remains mostly idle. In such a case, PAAR eliminates almost all refresh operations as very few DRAM rows are allocated by LeNet.

Full-RTC takes advantage of both RTT and PAAR at the same time and it reduces DRAM energy consumption to 0.39x, achieving greater DRAM energy savings than each technique achieves alone.



(a) Full-RTC (b) Mid-RTC (c) Min-RTC

Fig. 10. DRAM energy consumption of different RTC implementations normalized to the baseline LPDDR4 DRAM with standard refresh for AlexNet (AN), LeNet (LN), GoogleNet (GN), Winograd (WG), ResNet50 (RN), and Generative Adversarial Network (GAN).

Mid-RTC. Figure 10b plots the DRAM energy savings of mid-RTC. Mid-RTC implements a low-overhead version of PAAR that operates at DRAM bank granularity, and thus PAAR in mid-RTC eliminates refresh operations *only* if a bank does not have any allocated rows. As a result, mid-RTC

PAAR saves less DRAM energy compared to full-RTC PAAR. Similarly, mid-RTC implements a lighter version of RTT that is effective *only* when memory access rate is higher than refresh rate, i.e., when memory accesses activate all rows that contain data at least once in every 64 *ms* refresh period. Mid-RTC reduces average DRAM energy consumption to 0.53x compared to the baseline system. Therefore, mid-RTC RTT is not as effective as full-RTC RTT, which can partially align memory accesses with refreshes and issue explicit refresh only when necessary.

Min-RTC. Figure 10c plots the DRAM energy savings for min-RTC. Min-RTC is the most lightweight RTC implementation that only employs the same RTT technique as in mid-RTC. On average, it reduces DRAM energy consumption to 0.76x compared to the baseline. Min-RTC provides the largest benefits for AlexNet at 60 fps, reducing DRAM energy consumption by 40.0%.

We conclude that all three variants of RTC save DRAM energy and the system designer can choose the variant that fits best the energy and area constraints.

6.2 Sensitivity to DRAM Chip Capacity

Figure 11 plots the energy savings of full-RTC when employed in systems with different DRAM capacities. On average, full-RTC provides higher DRAM energy saving as the DRAM capacity increases, consuming 78.8% less DRAM energy for 64 Gb DRAM. This is because high-capacity DRAM contains a large number of unallocated rows, which PAAR skips refreshing. We note that two workloads, RN and GAN, consume more energy with 32 Gb DRAM than with 16 Gb DRAM. This is because the 32 Gb DRAM we evaluate has the same number of DRAM rows as the 16 Gb DRAM but each row contains double the number of DRAM cells compared to the 16 Gb DRAM. We notice that RN and GAN allocate slightly more DRAM rows *partially* when using 32 Gb DRAM, which slightly increases the DRAM energy consumption compared to 16 Gb DRAM.

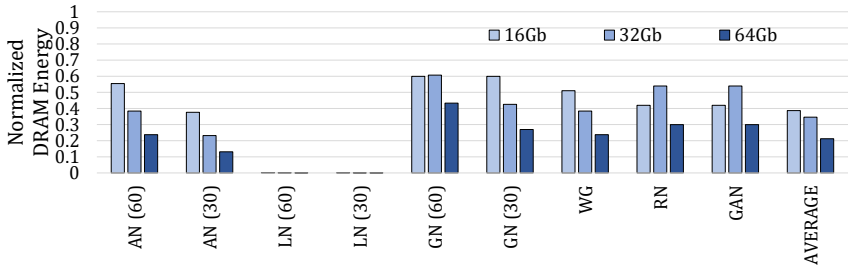


Fig. 11. DRAM energy savings of Full-RTC when using DRAM chips with different densities.

6.3 Sensitivity to Data Locality Exploitation

Data Locality exploitation refers to the ability of the system to cache the data read from DRAM. For example, a data locality exploitation of 100% implies that once the data is read from DRAM during an iteration in a CNN layer, the data never leaves the CPU cache, and thus it is not read from the DRAM again during the same iteration. Similarly, a data locality exploitation of 50% implies that the data set is read twice from the DRAM during each iteration. For many CNN applications, it is likely to achieve a data locality exploitation of approximately 100%, as reported in [11].

We now elaborate on the impact of data locality exploitation on the effectiveness of RTC. Figure 12 plots the normalized DRAM energy consumption for RTC with 50% and 100% data locality exploitation. The absolute energy savings of the PAAR components of RTC are not dependent on data locality exploitation. This is because PAAR eliminates refreshes to unallocated regions in DRAM and the rate at which allocated regions are accessed does not affect PAAR. However, overall DRAM energy reduction with PAAR reduces when data locality exploitation is low because

frequent DRAM accesses increase the access energy proportionally to the refresh energy, which remains constant.

The RTT component of RTC benefits more from low data locality exploitation. As we explain in Section 3.2, RTT eliminates refresh overhead when an application accesses its data frequently enough. Therefore, low data locality exploitation causes the DRAM to be accessed more frequently and this enables more of the refresh requests to be synchronized with accesses.

Overall, full-RTC saves 41.3% and 61.3% DRAM energy for 50% and 100% data locality exploitation, respectively.

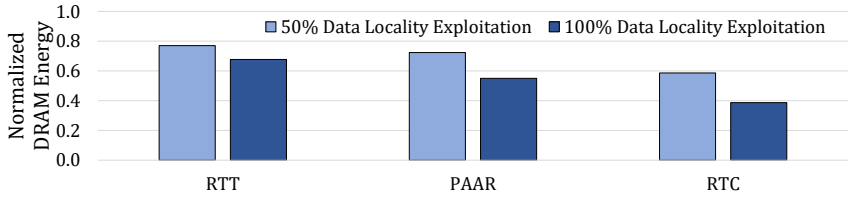


Fig. 12. Average DRAM energy savings of full-RTC vs. data locality exploitation ratio.

6.4 Comparison to the Most Relevant Works

In this section, we compare our RTC mechanism against prior works that attempt to reduce the DRAM refresh overhead. In particular, we compare our work with SmartRefresh [24], the most closely-related work to RTC. The key idea of SmartRefresh is to keep a history of the recently-accessed rows and avoid refreshing these rows as their cells' charge is already replenished when they were recently accessed. SmartRefresh maintains 3-bit counters for each row. Using the counters, it ensures that a row is not refreshed if it had been accessed recently. To compare RTC against SmartRefresh, we implement a DRAM controller with additional row counters (needed for SmartRefresh). For this evaluation, we assume an 8 GB DRAM module with a row size of 2048 B. To utilize the DRAM bandwidth, we run multiple instances of LeNet (LN), GoogleNet (GN), and AlexNet (AN). We assume that each CNN requires operation at 60 fps. We calculate the access patterns using state-of-the-art row stationary data flow [11]. Figure 13 shows the energy savings of RTC over SmartRefresh. The figure shows that RTC provides from 28% to 96% energy reduction, compared to SmartRefresh.

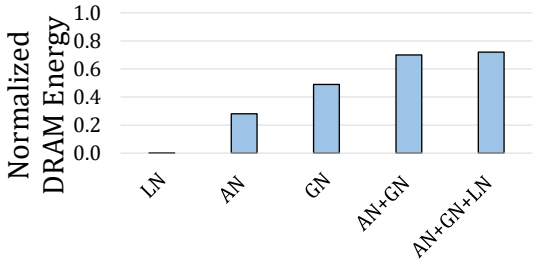


Fig. 13. DRAM energy consumption with RTC normalized to DRAM energy consumption with SmartRefresh.

RTC outperforms SmartRefresh using three optimizations to reduce the refreshes. First, RTC aligns the refresh with reads. In this way it ensures that the energy spent on both refresh and read is not wasted. Second, RTC prevents the refresh of the DRAM rows that are *not* being used (i.e., not allocated). Third, RTC does not refresh the rows that have been recently accessed. However, SmartRefresh applies *only* the third optimization by not refreshing recently-accessed DRAM rows.

As a result, SmartRefresh is ineffective when data transfer rate is lower than the refresh rate, e.g., when only LeNet is running on the system. In contrast, our RTC mechanism can reduce DRAM accesses regardless of the data rate. SmartRefresh is effective when access rate is greater than the refresh rate, which is the case in the rightmost two bar graphs, where multiple workloads run together. However, even in these two cases, RTC provides a significant ≈ 30% DRAM energy reduction over SmartRefresh. The main reason is the large number (e.g., 4,194,304 in our evaluated system) of SRAM counters that SmartRefresh needs to maintain to keep track of when each row is accessed. These counters consume a significant amount of energy that offsets the benefits of refresh reduction.

Refrint [1] is another refresh reduction technique, which has the advantage of being effective for low data access rates. However, Refrint has the downside of being applicable to only embedded DRAM that is used as a cache. This is because Refrint is based on the idea of refreshing only data that will be accessed in near future and flushing the rest back to the main memory. In contrast, our approach is generally applicable to any type of DRAM with small changes in the DRAM chips.

Similar to our PAAR technique, ESKIMO [35] skips refreshes to unallocated memory regions. However, it does not perform any refresh-access synchronization. Hence, ESKIMO does not reduce energy in allocated regions of memory.

6.5 Scalability Benefits

Refresh is a growing major energy and performance bottleneck with the scaling of the DRAM technology [46, 72]. RTC mitigates this negative scaling trend [3, 9, 13, 24, 46, 48, 49, 51, 72, 73, 81, 83, 87, 92–94, 114] for a class of applications by minimizing the need to refresh with its Refresh Triggered Transfer (RTT) and Partial Array Auto Refresh (PAAR) techniques. For a 64 *Gb* DRAM chip, even when working at peak bandwidth, refresh is expected to consume 46% of the total DRAM energy [45, 72]. To understand how RTC mitigates the refresh overhead, consider two extremes of applications' DRAM access characteristics. The first extreme is when the application has a small data set. For this scenario, almost all the DRAM energy will be spent on refresh. The PAAR technique eliminates this refresh overhead. It should be noted that when the memory controller puts the DRAM into self-refresh mode or power-down mode, PAAR still reduces DRAM energy consumption since rows are still refreshed while conventional DRAM is in one of these modes, and PAAR can eliminate unnecessary refreshes. The second extreme is when the application utilizes the *entire* DRAM capacity and has a high DRAM bandwidth demand. Note that, in this scenario, DRAM cannot switch to a low-power mode as it needs to keep servicing access requests. In such a scenario, conventional DRAM still spends a significant amount of energy on refresh in addition to read/write accesses. [45, 72] report that 47% of the total DRAM energy is spent while refreshing a DRAM chip of size 64 *Gb*. However, in an RTC-enabled DRAM, a large portion of refreshes can be eliminated by implicit refreshes for applications that have regular memory access patterns. Thus, in both extremes, RTC reduces the energy spent on refresh, and thus provides better scalability of DRAM in future technology nodes for applications with access patterns that are amenable to it.

To make the above arguments more concrete, we quantify the scalability of RTC for emerging large DRAMs when used for CNN applications. We perform an experiment by utilizing the entire bandwidth of a DRAM module. We show our results in Figure 14. It can be seen that RTC-enabled DRAM almost completely eliminates the DRAM refresh energy for CNN applications. Note that our results are consistent with prior work [45, 72], providing external validity to the experimental setup that we use.

6.6 Overhead of RTC

RTC-enabled DRAM incurs almost none to modest area, energy, and latency overheads. The area overhead mainly stems from 1) the configurable refresh counter (see Section 4), 2) the *AGU*

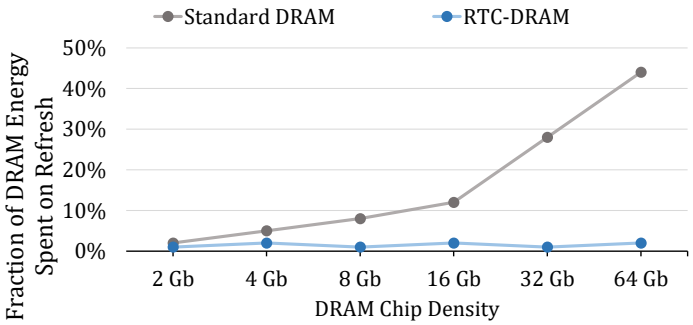


Fig. 14. Fraction of DRAM energy spent on refresh as a function of DRAM chip capacity.

for address generation, 3) the modifications to the data path (see Figure 6), 4) the RTC Backend Controller (see Section 4.3.2) and 5) the RTC Frontend Controller. We quantify the area overhead by synthesizing full-RTC in the standard CMOS 40 *nm* technology. We synthesize the corresponding logic components of the conventional DRAM also in the same process since we do not have access to DRAM process technology parameters. However, to approximate DRAM process in a more fair and accurate way, we restrict the physical design tool to use only three layers as commonly done in DRAM process. Our experiments show that RTC has an area overhead of 0.18% compared to a conventional 2 *Gb* DRAM chip. This area overhead proportionally decreases as DRAM chip density increases. This is because the area of only a few RTC components (e.g., counters) increase with DRAM chip density, whereas the area of a large number of components (e.g., RTC Backend Controller) do not change.

The latency overhead of the RTC logic stems from the extra cycles needed to reconfigure the RTC registers and state machines. However, the latency overhead is negligibly small compared to the execution time of a typical CNN-like application and reconfiguration of the RTC logic likely occurs only once when an application starts.

6.7 Using RTC with Non-CNN Workloads

So far, we have focused on CNNs as an example for discussing and quantifying the benefits and overhead of RTC. However, we believe RTC can be applied to a wide variety of applications that have a regular access pattern. We analyze the access patterns of three such well-known applications and estimate the benefits of RTC while executing them. These applications are: 1) Face recognition algorithm using Eigenfaces [60], 2) Bayesian Confidence Propagation Neural Network (BCPNN), a spiking neural network model of biologically plausible human brain cortex [20], and 3) the bioinformatics sequence alignment algorithm BFAST [31]. The reason for choosing these particular applications is that all of them largely differ in DRAM access characteristics compared to CNNs.

Figure 15 shows the estimated DRAM energy reduction for these three applications when using full-RTC-enabled DRAM chips with different densities. Face recognition is a streaming application that requires multiple filtering stages, which typically access the same data multiple times from DRAM. We evaluate face recognition using images of size $1024 * 1024 * 3$ and frame rate of 60 fps. We find that full-RTC saves 12.2% to 31.9% DRAM energy for face recognition depending on the DRAM chip density.

BCPNN is a memory- and compute-intensive application that requires approximately 740 teraflop/s computational bandwidth and 30 TB memory storage with a bandwidth of 112 TB/s [20]. During a single iteration, the BCPNN workload accesses its entire allocated memory four times. Because of such high rate of access to all of the allocated memory, the RTT technique largely eliminates the need for refresh in BCPNN, whereas PAAR provides small benefits since BCPNN allocates

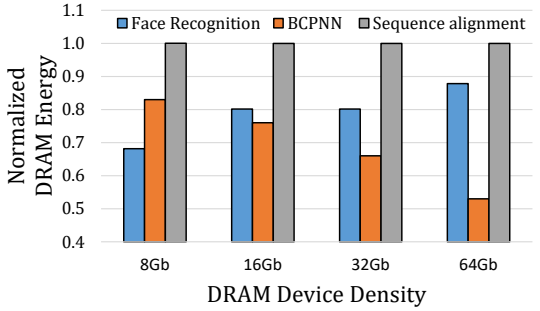


Fig. 15. DRAM energy savings of RTC on applications from different domains

the majority of the system's memory. Full-RTC saves 17.0% to 47.3% DRAM energy for BCPNN depending on the DRAM chip density.

BFAST is based on the well-known Smith-Waterman local DNA sequence alignment algorithm [103]. BFAST has a mix of random- and linear-access patterns. For this application, the RTC circuitry is bypassed as neither PAAR nor RTT is effective. By evaluating BFAST, we show that RTC can be disabled when it is not effective, which incurs less than 0.01% DRAM energy overhead, as shown in the rightmost bars of Figure 15.

7 RELATED WORK

To our knowledge, this work is the first to methodically synchronize applications' memory accesses with DRAM refreshes, so that the overhead caused by refresh operations is significantly reduced in Convolutional Neural Networks (CNNs). We briefly describe related work in DRAM refresh optimization and CNN storage optimization.

7.1 DRAM Refresh Optimization

Several previous works change the DRAM refresh scheduling policy to improve DRAM energy efficiency or performance. Bhati et al. [4] present a flexible refresh mechanism to reduce the refreshes. Stuecheli et al. [105] propose a technique that avoids interfering requests by altering the refresh schedule. It delays a refresh depending on the number of postponed refreshes and the predicted rank idle time. Mukundan et al. [81] propose various scheduling techniques to tackle command queue contention. Chang et al. [9] provide mechanisms to parallelize accesses and refreshes via scheduling and DRAM changes. However, all these techniques consider refresh and memory access as two disjoint processes and attempt to reduce the collisions between them as opposed to synchronizing accesses and refreshes like we do.

Various works [3, 12, 15, 16, 18, 24, 35, 48–52, 57, 72, 73, 75–77, 93, 97, 114] reduce unnecessary refreshes by exploiting the properties of DRAM cells and stored data. These works require expensive mechanisms to discover the retention times of different DRAM cells [3, 12, 15, 16, 18, 24, 48–51, 72, 73, 76, 93, 97, 114] or require knowledge of how tolerant stored data is to retention failures [15, 35, 52, 57, 75, 77]. RTC does not require such methods.

Zulian et al. [121] propose a mechanism that creates a mask of recently-accessed rows for each bank and introduces a modified refresh command to skip refreshing the masked rows. Their mechanism, likely concurrently developed with RTC, achieves a similar goal as RTC but has a large area overhead as it stores one bit for every row in DRAM.

SmartRefresh [24], Refrint [1], and Refree [95] are techniques that reduce the refresh overhead based on the memory access patterns of applications. These techniques are closely related to RTC. SmartRefresh [24] reduces refresh energy in DRAM by maintaining a timeout counter for each row.

This mechanism avoids unnecessary refreshes of recently accessed rows. However, SmartRefresh does not skip refreshing rows that do not store useful data. Thus, SmartRefresh is not effective for applications that have a small memory footprint where a significant number of DRAM rows do not contain useful data. Furthermore, SmartRefresh requires significant additional energy to maintain the large number of counters (see Section 6). *Refrint* [1] eliminates refresh to unused DRAM rows. However, its overheads are evaluated only for embedded DRAMs. Implementing this technique on off-chip DRAMs would require changing the memory arrays (i.e., it would be even more invasive than Full-RTC). Furthermore, similar to SmartRefresh, Refrint suffers from the overhead of maintaining the state of each DRAM row. Refree [95] combines a non-volatile PCM memory with conventional DRAM to eliminate DRAM refresh by moving a row to PCM when the row needs to be refreshed. Refree requires retention timeout counters and incurs overhead of moving data between PCM and DRAM. Compared to these approaches, RTC does not require any per row state. RTC improves the energy efficiency with small overhead on the DRAM chip and the memory controller.

ESKIMO [35] eliminates refreshes in unallocated memory regions. However, ESKIMO does not synchronize memory accesses with refreshes, and thus it does not reduce refresh energy in memory regions that allocate data. Using RTT and PAAR, RTC reduces the energy overhead of refresh operations on both allocated and unallocated portions of the memory.

7.2 CNN Storage Optimization

Driven by the success of CNNs as a machine learning technique, many researchers have focused on implementation aspects of CNN. While initially researchers focused on speeding up and improving the energy efficiency of the computational aspects [19, 96, 109], recently, the research have shifted towards improving the efficiency of the memory [10, 11, 104].

Chen et al. [10] show that CNNs can be viewed as nested loops. They present an accelerator that reduces memory footprint using loop tiling. Du et al. [17] build on top of [10] and propose an accelerator architecture that uses only SRAM to store application data, eliminating DRAM completely. While their approach is applicable some application domains, many accelerators [11, 104] are designed to work with a DRAM to meet the memory requirements of large neural networks. Chen et al. [11] show a technique to optimize the data movement between the memory and the computational units. Song et al. [104] present a technique to reduce the number of memory accesses using compression in classification layers. However, even after fully exploiting data locality, most of the energy is still spent on data transfers between DRAM and SRAM. Overall, these prior works aim to mitigate DRAM overhead in NN applications by exploiting data locality to better utilize SRAM-based memories. However, such techniques do not reduce DRAM refresh energy, and thus, DRAM refresh incurs significant overhead.

RANA [111] employs embedded DRAM (eDRAM) as an additional on-chip buffer to SRAM. RANA mitigates the refresh overhead of eDRAM by disabling refresh when data lifetime in an eDRAM bank is shorter than the retention time of the DRAM. RTC is complementary to this work as RTC mitigates the refresh overhead when data stored in DRAM has a long lifetime by synchronizing accesses to data with refresh operations.

EDEN [61] implements energy-efficient approximate DRAM for neural network inference by exploiting the error tolerance property of neural networks. EDEN has the limitation of being only applicable to data that has error tolerance. In contrast, RTC can mitigate DRAM refresh without causing bit flips due to retention failures in DRAM. EDEN and RTC can be combined for higher energy savings than each can achieve alone.

To the best of our knowledge, RTC is the first work that provides architectural solution for mitigating DRAM refresh energy in CNNs by synchronizing applications' memory accesses with DRAM refresh operations.

8 FUTURE WORK

We envision at least two major avenues of future work.

First, we plan to evaluate more applications and check if they can benefit from the new Refresh Triggered Computation model we propose. For example, we expect various applications from domains such as deep learning, computer vision, bioinformatics, and high-performance computing to highly benefit from RTC. We believe RTC has the potential to be applied to a wide range of applications from a variety of domains, as long as the access patterns can be regularized and synchronized with refresh.

Second, we plan to use RTC with a new class of neural networks, *Self-Organizing Maps* that a prior work [119] uses for rapid and accurate identification of bacterial genomes and their resistance to antibiotics. We also plan to incorporate the RTC technique as a part of the SiLago [30] framework, which is a Lego-inspired VLSI design framework that we develop. We plan to expand such a synthesis framework to map multiple complex workloads to custom SiLago design instances that will use DRAM enhanced with RTC as main memory.

9 CONCLUSION

We introduce a new software/hardware cooperative DRAM refresh optimization technique, which we refer to as Refresh Triggered Computation (RTC). RTC significantly reduces DRAM refresh overhead using two key concepts. First, it synchronizes DRAM refreshes with application read/write accesses to reduce the number of required refresh operations by exploiting the fact that application DRAM accesses implicitly replenish the charge of the DRAM cells. Second, RTC eliminates refreshing of rows that do not have any data allocated. We propose three variants of RTC, which differ in the level of area overhead incurred in the memory controller and the DRAM chip. Our extensive evaluations using commonly-used Convolutional Neural Networks (CNNs) show that the most aggressive variant of RTC reduces average DRAM energy by 61.3% while incurring only 0.18% area overhead over a conventional DRAM chip. We also show that RTC improves DRAM energy consumption of workloads from different domains. We conclude that RTC largely mitigates DRAM refresh overhead in both CNN applications and various other applications by synchronizing applications' DRAM accesses with DRAM refresh operations. We hope that RTC inspires other software/hardware cooperative mechanisms to reduce DRAM energy in data-intensive workloads.

ACKNOWLEDGMENTS

We thank the anonymous TACO 2020 reviewers for their feedback and the SAFARI group members for the stimulating intellectual environment they provide. We acknowledge the generous gifts provided by our industrial partners: ASML, Facebook, Google, Huawei, Intel, Microsoft, and VMware. This research was supported in part by the Semiconductor Research Corporation. An earlier version of this article was placed on arxiv.org [39] in October 2019.

REFERENCES

[1] A. Agrawal, et al. 2013. Refrint: Intelligent Refresh to Minimize Power in on-chip Multiprocessor Cache Hierarchies. In *HPCA*.
[2] AMBA Specification. 1999. Rev. 2.0. *ARM, http://www.arm.com* (1999).
[3] S. Baek, et al. 2014. Refresh Now and Then. *IEEE TC* (2014).
[4] I. Bhati, et al. 2015. Flexible Auto-Refresh: Enabling Scalable and Energy-Efficient DRAM Refresh Reductions. In *ISCA*.
[5] J. M. Chabloz et al. 2014. Low-Latency Maximal-Throughput Communication Interfaces for Rationally Related Clock Domains. *TVLSI* (2014).
[6] K. K. Chang, et al. 2016. Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization. In *SIGMETRICS*.

[7] K. K. Chang, et al. 2016. Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM. In *HPCA*.

[8] K. K. Chang, et al. 2017. Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms. In *SIGMETRICS*.

[9] K. K.-w. Chang, et al. 2014. Improving DRAM Performance by Parallelizing Refreshes with Accesses. In *HPCA*.

[10] T. Chen, et al. 2014. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *ASPLOS*.

[11] Y. Chen, et al. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *ISCA*.

[12] H. Choi, et al. 2020. Reducing DRAM Refresh Power Consumption by Runtime Profiling of Retention Time and Dual-Row Activation. *Microprocessors and Microsystems* (2020).

[13] C. Chou, et al. 2015. Reducing Refresh Power in Mobile Devices with Morphable ECC. In *DSN*.

[14] Cobham. 2017. GRLIB IP Library User's Manual. http://www.gaisler.com/products/grlib/grlib.pdf.

[15] Z. Cui, et al. 2014. DTail: A Flexible Approach to DRAM Refresh Management. In *ICS*.

[16] A. Das, et al. 2018. VRL-DRAM: Improving DRAM Performance via Variable Refresh Latency.. In *DAC*.

[17] Z. Du, et al. 2015. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *ISCA*.

[18] P. G. Emma, et al. 2008. Rethinking Refresh: Increasing Availability and Reducing Power in DRAM for Cache Applications. *IEEE Micro* (2008).

[19] H. Esmaeilzadeh, et al. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *MICRO*.

[20] N. Farahini, et al. 2014. A Scalable Custom Simulation Machine for the Bayesian Confidence Propagation Neural Network Model of the Brain. In *ASP-DAC*.

[21] N. Farahini, et al. 2014. Parallel Distributed Scalable Runtime Address Generation Scheme for a Coarse Grain Reconfigurable Computation and Storage Fabric. *Microprocessors and Microsystems* (2014).

[22] S. Ghose, et al. 2019. Demystifying Complex Workload-DRAM Interactions: An Experimental Study. In *SIGMETRICS*.

[23] S. Ghose, et al. 2018. What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study. In *SIGMETRICS*.

[24] M. Ghosh et al. 2007. Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-stacked DRAMs. In *MICRO*.

[25] I. Goodfellow, et al. 2014. Generative Adversarial Nets. In *NIPS*.

[26] H. Hassan, et al. 2019. CROW: A Low-Cost Substrate for Improving DRAM Performance, Energy Efficiency, and Reliability. In *ISCA*.

[27] H. Hassan, et al. 2016. ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality. In *HPCA*.

[28] H. Hassan, et al. 2017. SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies. In *HPCA*.

[29] K. He, et al. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.

[30] A. Hemani, et al. 2017. Synchoricity and NOCs Could Make Billion Gate Custom Hardware Centric SOCs Affordable. In *NOCS*.

[31] N. Homer, et al. 2009. BFAST: An Alignment Tool for Large Scale Genome Resequencing. *PLoS ONE* (2009).

[32] M. Ilić et al. 2011. Address Generation Unit as Accelerator Block in DSP. In *TELSIKS*.

[33] T. Instruments. 2002. TMS320C55x DSP Mnemonic Instruction Set Reference Guide. *Literature Number: SPRU374G October* (2002).

[34] E. Ipek, et al. 2008. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *ISCA*.

[35] C. Isen et al. 2009. ESKIMO - Energy Savings Using Semantic Knowledge of Inconsequential Memory Occupancy for DRAM Subsystem. In *MICRO*.

[36] K. Itoh. 2013. *VLSI Memory Chip Design*. Vol. 5. Springer Science & Business Media.

[37] ITRS. 2011. International Technology Roadmap for Semiconductors 2011 Edition: Executive Summary. http://www.itrs.net/Links/2011ITRS/2011Chapters/2011ExecSum.pdf.

[38] B. Jacob, et al. 2010. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann.

[39] S. M. Jafri, et al. 2019. Refresh Triggered Computation: Improving the Energy Efficiency of Convolutional Neural Network Accelerators. In *arXiv preprint arXiv:1910.06672*.

[40] S. M. A. H. Jafri, et al. 2017. MOCHA: Morphable Locality and Compression Aware Architecture for Convolutional Neural Networks. In *IPDPS*.

[41] S. M. A. H. Jafri, et al. 2013. Energy-Aware CGRAs using Dynamically Reconfigurable Isolation Cells. In *ISQED*.

[42] JEDEC. 2007. DDR3 SDRAM Standard. *JESD79-3* (2007).

[43] JEDEC. 2014. Low Power Double Data Rate 4 (LPDDR4). Standard No. JESD209-4.

[44] M. K. Jeong, et al. 2012. Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems. In *HPCA*.

[45] M. Jung, et al. 2015. Omitting Refresh: A Case Study for Commodity and Wide I/O DRAMs. In *MEMSYS*.

[46] U. Kang, et al. 2014. Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling. In *The Memory Forum*.

[47] B. Keeth, et al. 2007. *DRAM Circuit Design: Fundamental and High-Speed Topics.* John Wiley & Sons.
[48] S. Khan, et al. 2014. The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study. In *SIGMETRICS*.
[49] S. Khan, et al. 2016. PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM. In *DSN*.
[50] S. Khan, et al. 2016. A Case for Memory Content-Based Detection and Mitigation of Data-Dependent Failures in DRAM. In *CAL*.
[51] S. Khan, et al. 2017. Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content. In *MICRO*.
[52] D.-Y. Kim et al. 2020. Smart Adaptive Refresh for Optimum Refresh Interval Tracking using in-DRAM ECC. In *MWSCAS*.
[53] J. S. Kim, et al. 2018. Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines. In *ICCD*.
[54] J. S. Kim, et al. 2018. The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern Commodity DRAM Devices. In *HPCA*.
[55] J. S. Kim, et al. 2019. D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput. In *HPCA*.
[56] J. S. Kim, et al. 2020. Revisiting RowHammer: An Experimental Analysis of Modern Devices and Mitigation Techniques. In *ISCA*.
[57] S. Kim, et al. 2020. Charge-Aware DRAM Refresh Reduction with Value Transformation. In *HPCA*.
[58] Y. Kim, et al. 2014. Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA*.
[59] Y. Kim, et al. 2012. A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM. In *ISCA*.
[60] M. Kirby et al. 1990. Application of the Karhunen-Loeve Procedure for the Characterization of Human Faces. *IEEE TPAMI*.
[61] S. Koppula, et al. 2019. EDEN: Energy-Efficient, High-Performance Neural Network Inference Using Approximate DRAM. In *MICRO*.
[62] A. Krizhevsky, et al. 2012. Imagenet Classification with Deep Convolutional Neural Networks. In *NIPS*.
[63] A. Lavin et al. 2016. Fast Algorithms for Convolutional Neural Networks. In *CoRR*.
[64] Y. Lecun, et al. 1998. Gradient-based Learning Applied to Document Recognition. *Proc. of the IEEE*.
[65] D. Lee, et al. 2016. Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost. In *TACO*.
[66] D. Lee, et al. 2017. Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms. In *SIGMETRICS*.
[67] D. Lee, et al. 2015. Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case. In *HPCA*.
[68] D. Lee, et al. 2013. Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture. In *HPCA*.
[69] D. Lee, et al. 2015. Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM. In *PACT*.
[70] R. Leupers et al. 1996. Algorithms for Address Assignment in DSP Code Generation. In *ICCAD*.
[71] D. Liu. 2008. *Embedded DSP Processor Design: Application Specific Instruction Set Processors.* Elsevier.
[72] J. Liu et al. 2012. RAIDR: Retention-Aware Intelligent DRAM Refresh. In *ISCA*.
[73] J. Liu et al. 2013. An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms. In *ISCA*.
[74] L. Liu, et al. 2012. A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems. In *PACT*.
[75] S. Liu, et al. 2011. Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning. In *ASPLOS*.
[76] H. Luo, et al. 2020. CLR-DRAM: A Low-Cost DRAM Architecture Enabling Dynamic Capacity-Latency Trade-Off. In *ISCA*.
[77] Y. Luo, et al. 2014. Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory. In *DSN*.
[78] K. T. Malladi, et al. 2012. Rethinking DRAM Power Modes for Energy Proportionality. In *MICRO*.
[79] B. Mathew et al. 2004. A Loop Accelerator for Low Power Embedded VLIW Processors. In *CODES*.
[80] Micron. 2014. Mobile LPDDR3 SDRAM. https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/mobile-dram/low-power-dram/lpddr3/253b_12-5x12-5_2ch_8-16gb_2c0f_mobile_lpddr3.pdf?rev=1b66d5710434460eb13dc3be8faa6d77
[81] J. Mukundan, et al. 2013. Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM Systems. In *ISCA*.
[82] S. P. Muralidhara, et al. 2011. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In *MICRO*.

[83] O. Mutlu. 2013. Memory Scaling: A Systems Architecture Perspective. *IMW*.
[84] O. Mutlu. 2017. The RowHammer Problem and Other Issues we may Face as Memory Becomes Denser. In *DATE*.
[85] O. Mutlu et al. 2019. RowHammer: A Retrospective. In *TCAD*.
[86] O. Mutlu et al. 2008. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *ISCA*.
[87] O. Mutlu et al. 2015. Research Problems and Opportunities in Memory Systems. *SUPERFRI*.
[88] P. Nair, et al. 2013. A Case for Refresh Pausing in DRAM Memory Systems. In *HPCA*.
[89] P. J. Nair, et al. 2014. Refresh Pausing in DRAM Memory Systems. In *TACO*.
[90] P. J. Nair, et al. 2013. ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates. In *ISCA*.
[91] P. J. Nair, et al. 2016. XED: Exposing On-Die Error Detection Information for Strong Memory Reliability. In *ISCA*.
[92] M. Patel, et al. 2019. Understanding and Modeling On-Die Error Correction in Modern DRAM: An Experimental Study Using Real Devices. In *DSN*.
[93] M. Patel, et al. 2017. The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions. In *ISCA*.
[94] M. Patel, et al. 2020. Bit-Exact ECC Recovery (BEER): Determining DRAM On-Die ECC Functions by Exploiting DRAM Data Retention Characteristics. *MICRO*.
[95] B. Pourshirazi et al. 2016. Refree: A Refresh-free Hybrid DRAM/PCM Main Memory System. In *IPDPS*.
[96] W. Qadeer, et al. 2013. Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing. In *ISCA*.
[97] M. Qureshi, et al. 2015. AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems. In *DSN*.
[98] D. E. Rumelhart, et al. 1985. *Learning Internal Representations by Error Propagation.* Technical Report.
[99] V. Seshadri, et al. 2013. RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization. In *MICRO*.
[100] V. Seshadri, et al. 2017. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *MICRO*.
[101] V. Seshadri et al. 2019. In-DRAM Bulk Bitwise Execution Engine. *arXiv preprint arXiv:1905.09822* (2019).
[102] R. Shi, et al. 2015. A Locality Aware Convolutional Neural Networks Accelerator. In *DSD*.
[103] T. F. Smith, et al. 1981. Identification of Common Molecular Subsequences. *Journal of molecular biology*.
[104] H. Song, et al. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *ISCA*.
[105] J. Stuecheli, et al. 2010. Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory. In *MICRO*.
[106] C. Szegedy, et al. 2015. Going Deeper with Convolutions. In *CVPR*.
[107] I. Taniguchi, et al. 2009. Systematic Architecture Exploration Based on Optimistic Cycle Estimation for Low Energy Embedded Processors. In *ASP-DAC*.
[108] I. Taniguchi, et al. 2009. Reconfigurable AGU: An Address Generation Unit Based on Address Calculation Pattern for Low Energy and High Performance Embedded Processors. *IEICE*.
[109] O. Temam. 2012. A Defect-tolerant Accelerator for Emerging High-performance Applications. In *ISCA*.
[110] TN-46-15. 2007. *Low-Power Versus Standard DDR SDRAM.* Technical Report. Micron Technology, Inc.
[111] F. Tu, et al. 2018. RANA: Towards Efficient Neural Acceleration with Refresh-Optimized Embedded DRAM. In *ISCA*.
[112] S. Udayanarayanan et al. 2001. Address Code Generation for Digital Signal Processors. In *DAC*.
[113] G. T. Velilla. 2009. *Scratchpad-Oriented Address Generation for Low-Power Embedded VLIW Processors.* Ph.D. Dissertation.
[114] R. Venkatesan et al. 2006. Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM. In *HPCA*.
[115] T. Vogelsang. 2010. Understanding the Energy Consumption of Dynamic Random Access Memories. In *MICRO*.
[116] Y. Wang, et al. 2020. FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching. In *MICRO*.
[117] Y. Wang, et al. 2018. Reducing DRAM Latency via Charge-Level-Aware Look-Ahead Partial Restoration. In *MICRO*.
[118] C. Xue, et al. 2005. Optimizing DSP Scheduling via Address Assignment with Array and Loop Transformation. In *ICASSP*.
[119] Y. Yang, et al. 2018. RiBoSOM: Rapid Bacterial Genome Identification Using Self-Organizing Map Implemented on the Synchoros SiLago Platform. In *SAMOS*.
[120] T. Zhang, et al. 2014. Half-DRAM: A High-Bandwidth and Low-Power DRAM Architecture from the Rethinking of Fine-Grained Activation. In *ISCA*.
[121] É. F. Zulian, et al. 2020. Access-Aware Per-Bank DRAM Refresh for Reduced DRAM Refresh Overhead. In *ISCAS*.