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

Syed M. A. H. Jafri† Hasan Hassan‡ Ahmed Hemani‡ Onur Mutlu†
‡KTH Royal Institute of Technology †ETH Zürich

Abstract—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. Recently, many studies proposed CNN accelerator architectures with custom computation units that try to improve energy-efficiency and performance of CNN by minimizing data transfers from DRAM-based main memory. However, in these architectures, DRAM still contributes 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 mainly uses two 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 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, 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 design of RTC imposes an area overhead of 15% for AlexNet and GoogleNet, which are examples of large CNNs, and 47% for LeNet, which is a relatively smaller CNN. Our experimental evaluation on three well-known CNNs (i.e., AlexNet, LeNet, and GoogleNet) show that RTC can reduce the DRAM refresh energy from 25% to 96%, for the least aggressive and the most aggressive designs, respectively. Although we mainly use CNNs in our evaluations, we believe RTC can be applied to a wide range of applications, whose memory access patterns remain predictable for sufficiently long time.

I. 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 [8], [32]. 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 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 a DRAM cell that is composed of a capacitor-transistor pair, such a design makes it volatile due to charge leakage. 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], [7], [17], [35], [42], [44], [45], [46], [59].

CNNs typically have a large memory footprint [9], 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 [9]. To tackle this problem, recently-proposed accelerators focus on reducing the DRAM accesses by exploiting data locality [8], [9], [21], [51], [53]. Another approach compresses in-memory data to reduce the memory footprint and data transfer overhead of CNNs [53]. 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 [32], LeNet [33], and GoogleNet [55], which are implemented on an architecture similar to the state-of-the-art Eyeriss [9] 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 a DRAM as small as 2GB. For higher capacity DRAM, which is common in systems today, the refresh overhead is responsible for even larger portions of the overall energy consumption [35] (See Section VI). Thus, it is critical to investigate and develop techniques for reducing DRAM refresh overhead for implementing energy-efficient CNNs.

Various mechanisms have been proposed to mitigate the DRAM refresh overhead. Du et al. [12] 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 the CNNs typically require significant memory capacity [9], [51], [53], 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 [17] can reduce the refresh overhead by skipping the refresh operation for a

1Our methodology and accelerator architecture are described in Section III.
We propose Refresh Triggered Computation (RTC) as a general technique for reducing the number of refresh operations depending on the 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.

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). RTC reduces DRAM energy consumption by eliminating unnecessary refresh operations using two orthogonal mechanisms that are based on two new observations. First, we observe that large CNNs, such as AlexNet, 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. Our second mechanism, Partial-array Auto Refresh (PAAR), eliminates the refresh operations to the portions of the DRAM that are not used.

In this work, we implement and evaluate three versions of RTC that differ in the level of customization required on the DRAM device and the memory controller. The first version requires changes only in the memory controller, and is useful when the read/write requests are frequent, so that they can be coalesced with the refresh operations. For the second version, we slightly modify the implementation of the already-available Partial-array Self Refresh (PASR) feature in modern DRAM chips [23], [41], to enable that feature not only in self-refresh mode, but also during the normal operation of the DRAM. Third, we propose internal DRAM modifications that fully exploit the capabilities of RTC. In particular, we add an Address Generation Unit and an 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 three different CNNs, depending on its version, DRAM capacity, and the access pattern of the application.

Although we apply RTC to only CNNs in the scope of this paper, RTC can be adapted to a wide class of applications with a pseudo-stationary spatio-temporal access pattern. In other words, RTC is beneficial when the memory access pattern remains stationary for a sufficiently long period. A period is considered sufficiently long if it exceeds the latency for configuring the RTC logic. We believe that this behavior is prevalent for a wide variety of streaming applications (e.g., pattern recognition, sequence alignment) that operate on large amounts of data and hope that future work finds other use cases for RTC beyond CNN acceleration.

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 the read and write accesses.
- We propose Refresh Triggered Computation (RTC) as a general technique for reducing the number of refresh operations depending on the 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 versions 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 versions for three widely used CNN applications (i.e., AlexNet, LeNet, and GoogleNet). We show that RTC, in its most comprehensive version, reduces DRAM refresh energy in a state-of-the-art CNN accelerator by up to 96% (on average 60% across multiple CNNs).

II. BACKGROUND

In this section, we provide background on DRAM and CNNs, necessary to understand the RTC framework that we propose.

A. 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 64ms. 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 [34], 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.

Wordline
Bitline
Capacitor
Access Transistor
Cell

Bank
Row-buffer
Row
Sense Amplifier

![Fig. 2: DRAM cell and bank](image)

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 DRAM to ensure data integrity. 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.

### B. Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [33] 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 [50]. 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 [8], [51]. Since the offline training does not affect the performance of the end-application, we focus on the inference phase, similar to prior work [8], [9], [51], [53]. 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](image)

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 [32] performs about 3 billion DRAM accesses when processing a single image. Modern CNN accelerators [9] 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.

III. REFRESH TRIGGERED COMPUTATION

As we explained in Section II-A, the memory controller periodically issues refresh commands to DRAM, in order to ensure data integrity. For the chips available in the market today, the entire DRAM chip has to be refreshed every 64ms (or 32ms when operating at temperature exceeding 85°C). As there is a large number of rows in the chip, the memory controller issues a refresh command, which refreshes multiple rows in batch, once every 7.8us to complete the refresh cycle for the entire DRAM in 64ms. Such frequent refresh operations often conflict with read and write requests that are issued by the workloads running on the system [7], [35]. 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 such as CNNs, it is possible to synchronize and harmonize the refresh operations and the read/write requests, such that read/write requests of the application naturally refreshes DRAM. RTC not only eliminates conflicts between refresh and read/write, but it 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 IV.

A. Making Refresh Unnecessary

In Section II-A, we explain that both refresh and access requests perform similar operations in the DRAM circuit 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 by the access requests.

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 the refresh with access is a challenging problem.

B. 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 retention time (i.e., refresh period) 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 III-C and III-D 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 refresh 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 the rows refreshed and accessed during three refresh periods. The refresh requests iterate through rows r1 to r4 in the first two refresh periods. Close to the end of the first refresh period, the running 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, since, if we eliminate the refresh operations, the time since r1 was last refreshed would exceed the refresh period. However, all refresh operations in the third period are redundant, as the access requests already refresh the rows. Next, we introduce our techniques to align refreshes and access request when the three assumptions are relaxed.

C. Refresh Triggered Transfer

The simple scheme we proposed in Section III-B assumes that the periods 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 periods of access requests 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 IV-C.

Algorithm 1 describes how RTT handles the mismatch between the periods of the access requests and refreshes.\(^2\) If the application generates access requests as frequently as the refresh rate or faster, RTT completely removes the refresh overhead.

\(^2\) We adapt the algorithm from a technique that is used to align send and receive processes operating at rationally related clock frequencies [5], [22].

Fig. 4: Cyclic application access pattern vs. refresh pattern
and ensures data integrity as the accesses replenish the charge of the rows that they access. However, when the accesses are not frequent enough, the problem of matching the periods of the accesses and refreshes becomes more challenging. To tackle this problem, RTT keeps some of the refresh operations, which refresh the rows that are not accessed within the refresh period.

Algorithm 1 takes \( N_a \) and \( N_r \) as inputs\(^3\), 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 \( xfer \) signal, which determines whether a row will be explicitly refreshed or implicitly replenished when accessed to read/write data. Thus, when \( N_r \leq N_a \), \( xfer \) is set as 1 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 some additional refresh operations to compensate for the rows that are not accessed during the refresh period. Since \( N_r \) and \( N_a \) are rationed related, a pattern that interleaves between data-transfer and explicit refresh requests repeats at period of \( P = N_r / \gcd(N_r, N_a) \). To determine the interleaving between data-transfer and 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 \( xfer = 0 \) to indicate an explicit refresh, and increments the credit by \( N_a \).

**Algorithm 1 Rate matching algorithm**

\[
\begin{align*}
\text{procedure } \text{RATEMATCHING}(N_a, N_r) \quad & \text{for every refresh period do} \\
\text{if } N_r \leq N_a & \text{ then} \\
\quad & xfer \leftarrow 1 \quad \text{> implicit refresh} \\
\text{else} & \\
\quad P \leftarrow N_r / \gcd(N_r, N_a) \quad & c \leftarrow N_r \\
\quad \text{for } i \leftarrow 1 \ldots P \text{ do} & \\
\quad \text{if } c > N_r - N_a & \text{ then} \\
\quad & xfer \leftarrow 1 \quad \text{> implicit refresh} \\
\quad & c \leftarrow c - (N_r - N_a) \\
\text{else} & \\
\quad & xfer \leftarrow 0 \quad \text{> explicit refresh} \\
\quad & c \leftarrow c + N_a \\
\text{end if} & \\
\text{end if} \quad & \\
\text{end if} \quad & \\
\text{end for} \quad & \\
\text{end procedure} \quad &
\end{align*}
\]

To understand how the Algorithm 1 works, consider an example, where \( N_a = 2 \) and \( N_r = 4 \). In this case, 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 for loop

(line 8), \( c \) is greater than the difference \( N_r - N_a = 2 \). 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.

**Arbitrary 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 [16]. We implement AGU as a part of the DRAM circuitry and it can be configured to generate address patterns based on arbitrary affine function.

**D. 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 [51], 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 redundantly 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) [23], which already exists in low-power DRAM chips and is used to refresh only certain DRAM banks while in self-refreshing 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 [40], 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 [40], 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.
E. 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 introduces a small latency overhead. To compensate for this 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. For other applications, which have frequently changing access patterns, RTC can be disabled on-the-fly.

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 [37], [43].

IV. THE RTC ARCHITECTURE

In this section, we present the RTC architecture, which implements the concepts we introduced in Section III. 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.

A. Min-RTC

For this implementation, we restrain ourselves from applying 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 III-B. 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 VI-A), even such a simple mechanism saves significant energy.

B. 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 [57], 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.

C. Full-RTC

The most aggressive implementation, full-RTC, requires three new hardware units in the DRAM chip, as we show in Figure 6. First, to prevent a subset of rows from being refreshed, full-RTC modifies the in-DRAM refresh logic to be configurable by the memory controller. Second, to implement the RTT scheme (Section III-C) properly, full-RTC adds to the DRAM chip a new counter and logic (i.e., address generation unit (AGU)) to manage the counter. An application can configure the AGU at runtime to generate refresh requests with arbitrary address patterns. Third, full-RTC implements Algorithm 1 to determine which requests will transfer data from/to DRAM, and which requests will only refresh a row. Full-RTC implements this algorithm in the memory controller and also introduces modifications to the DRAM control logic to handle the xfer signals generated by the memory controller. We explain our design in more detail.

![Fig. 6: Modified DRAM to support Full-RTC](image)

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 DRAM architecture.

After an application starts execution, it needs to provide memory access characteristics to the memory controller. In our
implementation of the RTC framework on CNN accelerators, we implement a runtime resource manager in the software stack, which monitors the application, and communicates the information that it gathers about the application to the memory controller. Then, the memory controller configures the in-DRAM RTC components accordingly.

We make modifications on the DRAM interface to extend it such that we can configure the RTC hardware components (Section IV-C2). Particularly, we add four I/O signals to the DRAM interface to modify the RTC circuitry. These signals are ld, refr, RTT, and rate_fsm, which we explain in the next subsection.

2) Modifications to the DRAM chip: We implement the necessary circuitry to fully support the RTT and PAAR techniques.

To implement the RTT technique that we propose in Section III-C for aligning refreshes and accesses, we modify the DRAM control circuit further by adding an RTT counter. After the alignment, the RTT counter completely replaces the conventional DRAM refresh counter, and thus the addresses that the RTT counter generates are used for refreshing the rows. The RTT counter logic implements Algorithm 1 to determine whether the address that it generates will be used solely to perform refresh or also to transfer data from/to the memory controller.

The RTT counter does not only generate the address used to perform a refresh inside DRAM, but it also generates the address to perform an access. However, in conventional DRAM, the memory controller issues the DRAM commands along with the address via the DDR interface, which incurs additional energy consumption compared to RTC.

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

PAAR improves DRAM energy efficiency by eliminating the refresh operations to DRAM regions that are not allocated. In standard 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, such that only a specific region of DRAM is refreshed, rather than the entire DRAM. As we illustrate in Figure 6, we implement this feature by introducing two registers that specify the lower and upper bounds of the region to refresh. We also modify the DRAM control circuit to make these registers configurable.

We modify the DRAM architecture to implement RTC control logic, which implements the RTT and PAAR mechanisms and provides an interface for reconfiguring them. In Figure 7, we describe the operation of the RTC control logic using a state diagram. During the initial idle state, the RTC control logic expects signals for reconfiguring one of its three components (shaded with different colors) or transitioning to Active state where RTT is active (operation during Active state is described in Figure 8).

To enter a reconfiguration state, the load signal (ld) has to be asserted along with another signal that indicates which of the three reconfiguration states to enter. First, when refr = 1, the RTC control logic reconfigures the start and end bounds of the refresh counter using new values received from the memory controller in successive DRAM cycles. Second, when rt = 1, the RTC control logic reconfigures the RTT counter. Third, when rate_fsm = 1, the RTC control logic reconfigures the two parameters Na and Nr that we describe in Section III-C.

In Figure 8, we show the state diagram describing the RTT operation. While RTC configuration 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. In the Act state, the RTC control logic generates a DRAM command to activate the row that either the RTT counter or the refresh counter points to as shown in Figure 6.

After Act, the RTC logic performs either implicit or explicit refresh depending on the xfer signal. First, if the time interval between two consecutive reads/writes is greater than the required refresh interval (i.e., xfer = 0), the row is explicitly refreshed. We indicate this case with a red line from the Act to the Pre state, which precharges the open row. Second, when xfer = 1, 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. The underlying rationale is that if the read/write path is taken, the rows are implicitly refreshed. Depending on the IO width, the column AGU, shown in Figure 6, will transfer...
the row address in pieces. The control remains in this state as long as $ld = 0$. When $ld = 1$, the control returns to the idle state, which allows the RTC to be reconfigured as we show in Figure 7.

V. 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 [21], similar to Eyeriss [9], via an AMBA AHB [2] bus. As we illustrate in Figure 9, the accelerator is implemented in the logic-layer of a DRAM-based 3D-stacked memory. The CNN accelerator has a private 108KB scratch-pad memory, as in Eyeriss [9], and it also incorporates a memory controller to interface the upper DRAM layers of the 3D-stacked memory. We evaluate DRAM capacities of 2GB, 4GB, and 8GB.

To analyze the effectiveness of RTC at saving DRAM refresh energy, we evaluate three widely-used CNN applications, GoogleNet [55], AlexNet [32], and LeNet [33]. 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.

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 40nm 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 [10] as the baseline. We extend this controller as we discuss in Section IV-C1. We report area and energy overhead 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 overhead in DRAM, we use the Rambus DRAM model [60] for different DRAM dimensions and trace 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 40nm DRAM from ITRS [20]. 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 in-house simulator [21] for three popular CNNs (AlexNet (AN) [32], LeNet (LN) [33], and GoogleNet (GN) [55]) that have widely different memory requirements.

VI. EVALUATION

In this section, we analyze DRAM energy savings and area overhead of each variant of RTC compared to conventional low-power DRAM, LPDDR4 [23]. We evaluate three widely-used CNNs: AlexNet (AN), LeNet (LN), and GoogleNet (GN), which vary in memory footprint and access patterns. We study these CNNs on systems with different DRAM capacities. Furthermore, we provide a breakdown of the benefits of the RTT and PAAR techniques that our RTC framework implements.

A. Energy Savings

Figure 10 shows DRAM energy reduction for each of the three RTC variants, in comparison to standard low-power LPDDR4 DRAM. The figure shows two sets of plots (a,b,c, and d,e,f) that differ in data locality exploitation, which refers to the ability of the system to cache the data read from the DRAM. A data locality exploitation of 100% implies that once the data is read from DRAM, the data never leaves the CPU cache, and thus it is not read from the DRAM again during an 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 [9].

We plot the DRAM energy savings achieved using RTT and PAAR separately, as well as when the two techniques are combined together. We report results for each CNN at two different frame rates. The reduction in DRAM energy consumption, achieved by RTT, is primarily dependent on the ratio of DRAM refresh rate and access rate. The closer the two are matched, the greater the energy reduction. We first elaborate on the first set of graphs (i.e., a, b, and c) followed by the second set of graphs (i.e., d, e, and f). For the full-RTC implementation, shown in Figure 10a, the energy saving due to RTT is 44% at 60 fps, and 30% at 30 fps. This is because for 60 fps, the accesses match the frequency of the refreshes, and thus they can easily be aligned. However, for 30 fps, the access rate drops to approximately half of the 60 fps case, whereas the refresh frequency remains the same. This results in a mismatch between the access and refresh rates, and thus fewer refreshes overlap with read/write accesses. In other words, there are more explicit refreshes, which result in lower DRAM energy reduction. If we use a larger memory, the number of refresh cycles will increase for both 30 and 60 frames/s scenarios, resulting in a reduction in energy savings for RTT. For LeNet, the effectiveness of RTT is minimal because of the small memory footprint of LeNet, and small number of the read/write accesses in comparison to the refreshes.

For PAAR, the benefits stem from the fact that if the data is no longer needed there is no need to refresh the corresponding rows. As a result, for LeNet, the small data set is read out once and after that there is no further need to maintain those rows, resulting in 96% savings in DRAM energy. It is also worth comparing the benefits of PAAR for AlexNet with 2 GB DRAM for the two frame rates. For 60 fps, since we need relatively more accesses due to the higher frame rate, refresh energy constitutes a smaller portion of the overall DRAM energy. However, for 30 fps, the refresh overhead is proportionally
larger as accesses are less frequent. As a result, the PAAR technique achieves greater energy savings for the 30 fps mode.

Full-RTC combines the RTT and PAAR techniques. In Figures 10a and 10d, we see that RTC achieves the maximum DRAM energy reduction that each technique provides alone. For AN (60), RTT achieves greater DRAM energy reduction compared to PAAR, and thus, RTC uses the RTT technique. In contrast, for LN (60), the memory access rate does not match the required refresh rate. As a result, RTT is not effective in reducing DRAM energy for LN (60). Thus, RTC saves DRAM energy by using PAAR to eliminate accesses to unallocated DRAM rows. We conclude that RTT and PAAR can be combined such that one of the techniques can save DRAM energy when the other is not effective for a given workload.

Figure 10b shows the benefits of mid-RTC. Since mid-RTC implements a simple version of the PAAR technique that operates at bank granularity, it eliminates refresh operations for a bank only if no rows are allocated in the bank. Therefore, in mid-RTC, PAAR achieves less DRAM energy reduction compared to PAAR in full-RTC. In mid-RTC, RTT is only useful when memory access rate is greater than the refresh rate. Therefore, RTT in mid-RTC is not very effective for large DRAMs as the high refresh overhead reduces the energy savings achieved by RTT.

Figure 10c shows DRAM energy reduction of min-RTC. For 2 GB DRAM, min-RTC provides up to 20% reduction in DRAM energy for AN and GN. Min-RTC becomes less effective as DRAM size increases since it is harder to match memory access rate to the higher refresh rate in large DRAMs.

B. 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 [17], 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 8GB DRAM module with a row size of 2048B. 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 [9]. Figure 11 shows the energy savings of RTC over SmartRefresh. The figure shows that RTC provides from 28% to 96% energy reduction, compared to SmartRefresh.

RTC outperforms SmartRefresh using three optimizations to reduce the refreshes: (i) It aligns the refresh with reads. In this
way it ensures that the energy spent on both refresh and read
is not wasted, (ii) it prevents the refresh of the DRAM rows
that are not being used, and (iii) it does not refresh the rows
that have been recently accessed. In contrast, SmartRefresh
applies only the third optimization. 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 platform. 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 networks are 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.

Refrent [1] is another optimization technique, that has the
advantage of being effective also for low data rates. However,
Refrent has the downside of being applicable to only embedded
DRAM that is used as a cache. This is because Refrent 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 applicable to high-density commodity
DRAM that has much larger refresh overhead compared to
embedded DRAM.

Similar to our PAAR technique, ESKIMO [19] skips re-
freshes to unallocated memory regions. However, it does not
perform any refresh-access synchronization. Hence, ESKIMO
does not reduce energy in allocated regions of memory.

C. Scalability Benefits

Refresh is as a major energy and performance cost com-
ponent with the scaling of the DRAM technology [25], [35].
RTC mitigates this negative scaling trend [6], [7], [25], [35],
[36], [44], for a class of applications, by minimizing the need
to refresh with its Partial Array Auto Refresh (PAAR) and
Refresh Triggered Transfer (RTT). For a 64 Gb DRAM chip,
even when working at peak bandwidth, refresh is expected
to consume 46% of the total DRAM energy [24], [35]. To
understand how RTC mitigates the refresh overhead, consider
two extremes of applications’ DRAM requirements. 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 if the DRAM goes to self-refresh or power-down
mode, PAAR will further reduce energy consumption of the
low-power modes by adding another optimization dimension
(i.e., the rows to be refreshed). The second extreme is when
the application utilizes the maximum capacity and bandwidth
of the DRAM. Note that, in this scenario, DRAM cannot shift
to a low power mode as it would remain busy servicing access
requests. In such a scenario, conventional DRAM will still
spend a significant amount of energy on refresh in addition
to read/write accesses. [24], [35] 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, by the virtue of
RTT, almost all of the refreshes will be replaced by implicit
refreshes due to the regular application access patterns. Thus in
both extremes, RTC will reduce the energy spent on refresh and
thus provide better scalability of DRAM in future technology
nodes for applications with access patterns that are amenable.

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 12. It can be seen that RTC-enabled DRAM almost
completely eliminates the refresh energy for CNN applications.
Note that our results are consistent with prior work [24], [35],
providing external validity to the experimental setup that we
use.

![Fraction of DRAM energy spent on refresh, with increasing DRAM chip capacity for CNNs](image)

**Fig. 12:** Fraction of DRAM energy spend on refresh, with increasing DRAM chip capacity for CNNs

D. RTC Overheads

RTC-enabled DRAM incurs almost none to modest area,
energy, and latency overheads. The area overhead stems from
(i) the enhanced refresh counter (see Section IV), (ii) the RTT
counter and control along with the registers that specify the rate,
(iii) the modifications to the data path (see Figure 6), (iv)
the back-end controller FSM (see Section IV-C2) and (v)
the modifications to the front-end controller. We quantify the area
overhead by synthesizing our mechanism in the standard CMOS
40nm technology. We also synthesize the corresponding logic
components of the conventional DRAM in the same process.
Since DRAM commonly uses only three metal layers, we
restrict the physical design tool to use only three layers. This
allows us to have a fair and accurate method to measure the
overhead of RTC. Our experiments show that RTC has an area
overhead of 0.18% compared to a conventional 2Gb DRAM
chip. Obviously for large capacity DRAMs, this overhead would
be even less. This overhead increases less than logarithmically
with the increase in DRAM size because only a few components
(e.g., counters) increase logarithmically with address space,
whereas the remaining logic (e.g., FSM) is independent of the
size of the address space.

The latency overhead of the RTC logic stems from the
extra cycles needed to configure the RTC registers and state
machines. However, these cycles are negligibly small compared
to the total number of DRAM cycles in a typical CNN-like
application.
E. Other Applications

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 wider variety of applications that have a static access pattern in each iteration. We analyze the access patterns of three such well-known applications and estimate the benefits of RTC while executing them. These applications are: (i) Face recognition algorithm using Eigenfaces [30], (ii) Bayesian Confidence Propagation Neural Network (BCPNN), a spiking neural network model of the human cortex [15], and (iii) the bioinformatics sequence alignment algorithm BFAST [18]. Figure 13 shows the estimated DRAM energy reduction for these three applications when using full-RTC-enabled DRAM chips with different densities.

Our reason for choosing these particular applications is that all of them largely differ in terms of their DRAM access characteristics compared to CNNs. Face recognition, similar to the CNN processing phase, is a streaming application that requires multiple filtering stages. However, face recognition usually needs to access the same data from DRAM multiple times, which is a different access characteristic compared to CNN’s. For face recognition, both RTT and PAAR provide benefits. We estimate the benefits for 1024×1024×3 image size with the frame rate of 60 fps. BCPNN is a very memory and compute intensive application that requires performance close to 1 petaflop/s and 30 TBs of storage, at a bandwidth of 120 TB/s [15]. In each iteration, BCPNN accesses its entire allocated memory four times. RTC completely eliminates the refresh need for BCPNN by using RTT, though PAAR provides no benefits since majority of the memory is allocated. Finally, BFAST is based on the well-known Smith-Waterman local sequence alignment algorithm [52]. It has a mix of random- and linear-access patterns. For this application, the RTC circuitry is bypassed as neither PAAR nor RTT is effective.

![Fig. 13: DRAM energy savings of RTC on applications from different domains](image)

**VII. 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.

A. 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. [54] 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. [42] propose various scheduling techniques to tackle command queue contention. Chang et al. [7] 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], [11], [13], [17], [19], [26], [27], [28], [29], [35], [36], [38], [39], [46], [49], [59] reduce unnecessary refresh 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], [11], [13], [17], [26], [27], [28], [29], [35], [36], [46], [49], [59] or retention/approximation requirements of stored data [11], [19], [38], [39]. RTC does not require such methods.

SmartRefresh [17], Refrint [1], and Refree [47] are techniques that reduce the refresh overhead based on the memory access patterns of applications. These techniques are closely related to RTC. SmartRefresh [17] 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 VI). 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 [47] 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.

B. 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 [14], [48], [56], recently, the research have shifted towards improving the efficiency of the memory [8], [9], [53].
Chen et al. [8] show that CNNs can be viewed as nested loops. They present an accelerator that reduces memory footprint using loop tiling. Du et al. [12] build on top of [8] 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 [9], [53] are designed to work with a DRAM to meet the memory requirements of large neural networks. Chen et al. [9] show a technique to optimize the data movement between the memory and the computational units. Song et al. [53] 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 [58] 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 their 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 [31] implements energy-efficient approximate DRAM for neural network inference by exploiting the error tolerance property of neural networks. EDEN has the limitation of being applicable to data that has error tolerance. In contrast, RTC can mitigate DRAM refresh without causing bit flips due to retention failures in DRAM.

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.

VIII. CONCLUSION

In this paper, we describe 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 three commonly-used Convolutional Neural Networks (CNNs) show that the most aggressive variant of RTC reduces DRAM refresh energy from 25% to 96% while incurring only 0.18% area overhead over a conventional DRAM chip. We also show that RTC improves DRAM energy consumption on three other applications, including face recognition and sequence alignment. 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.

REFERENCES

[1] A. Agrawal, P. Jain, A. Ansari, and J. Torrellas, “Refrint: Intelligent Refresh to Minimize Power in on-chip Multi processor Cache Hierarchies,” in HPCA, 2013.
[2] AMBA Specification, “Rev. 2.0,” ARM, http://www.arm.com, 1999.
[3] S. Baek, S. Cho, and R. Melhem, “Refresh Now and Then,” IEEE TC, 2014.
[4] I. Bhati, Z. Chishti, S.-L. Lu, and B. Jacob, “Flexible Auto-Refresh: Enabling Scalable and Energy-Efficient DRAM Refresh Reductions,” in ISCA, 2015.
[5] J. M. Chabioz and A. Hemanis, “Low-Latency Maximal-Throughput Communication Interfaces for Rationally Related Clock Domains,” TVLSI, 2014.
[6] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, “Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization,” in SIGMETRICS, 2016.
[7] K. K.-W. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, “Improving DRAM Performance by Parallelizing Refreshes with Accesses,” in HPCA, 2014.
[8] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, “DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning,” in ASPLOS, 2014.
[9] Y. Chen, J. Emer, and Y. Sze, “Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks,” in ISCA, 2016.
[10] Cobbham, “GRLIB IP Library Users Manual.” [Online]. Available: http://www.gaisler.com/products/grlib/grlib.pdf
[11] Z. Cui, S. A. McKex, Z. Zha, Y. Bao, and M. Chen, “DTail: A Flexible Approach to DRAM Refresh Management,” in ICS, 2014.
[12] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, “ShiDianNao: Shifting Vision Processing Closer to the Sensor,” in ISCA, 2015.
[13] P. G. Emma, W. R. Reoehr, and M. Meterelliyoiz, “Rethinking Refresh: Increasing Availability and Reducing Power in DRAM for Cache Applications,” IEEE Micro, 2008.
[14] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Neural Acceleration for General-Purpose Approximate Programs,” in MICRO, 2012.
[15] N. Farahini, A. Hemanis, A. Lansner, F. Clermidy, and C. Svensson, “A Scalable Custom Simulation Machine for the Bayesian Confidence Propagation Neural Network Model of the Brain,” in ASP-DAC, 2014.
[16] N. Farahini, A. Hemanis, H. Sohofi, S. M. Jafri, M. A. Tajammul, and K. Paul, “Parallel Distributed Scalable Runtime Address Generation Scheme for a Coarse Grain Reconfigurable Computation and Storage Fabric,” Microprocessors and Microsystems, 2014.
[17] M. Ghosh and H.-H. S. Lee, “Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-stacked DRAMs,” in MICRO, 2007.
[18] N. Homer, B. Merriman, and S. Nelson, “BFAST: An Alignment Tool for Large Scale Genome Resequencing,” PLoS ONE, 2009.
[19] C. Isen and L. John, “ESKIMO - Energy Savings Using Semantic Knowledge of Inconsequential Memory Occupancy for DRAM Subsystem,” in MICRO, 2009.
[20] ITRS, “International Technology Roadmap for Semiconductors 2011 Edition: Executive Summary,” http://www.itrs.net/Links/2011ITRS/2011Chapters/2011ExecSum.pdf, 2011.
[21] S. M. A. H. Jafri, A. Hemanis, K. Paul, and N. Abbas, “MOCHA: Morphable Locality and Compression Aware Architecture for Convolutional Neural Networks,” in IPPS, 2017.
[22] S. M. A. H. Jafri, O. Ozbuk, A. Hemanis, N. Farahini, K. Paul, J. Ploisla, and H. Tenhunen, “Energy-Aware CRGs using Dynamically Reconfigurable Isolation Cells,” in ISQED, 2013.
[23] JEDEC, “Low Power Double Data Rate 4 (LPDDR4),” Standard No. JESD209-4, 2014.
[24] M. Jung, E. Zulian, D. M. Mathew, M. Herrmann, C. Brugger, C. Weis, and N. Wehn, “Omitting Refresh: A Case Study for Commodity and 3D I/O DRAMs,” in CODES+ISSS, 2015.
[25] U. Kang, H.-S. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. Choi, “Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling,” in The Memory Forum, 2014.
S. Khan, D. Lee, Y. Kim, A. R. Alameldeen, C. Wilkerson, and O. Mutlu, “The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study,” in SIGMETRICS, 2014.

S. Khan, D. Lee, and O. Mutlu, “PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM,” in ISCA, 2016.

S. Khan, C. Wilkerson, D. Lee, A. R. Alameldeen, and O. Mutlu, “A Case for Memory Content-Based Detection and Mitigation of Data-Dependent Failures in DRAM,” CAL, 2016.

S. Khan, C. Wilkerson, Z. Wang, A. R. Alameldeen, D. Lee, and O. Mutlu, “Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content,” in MICRO, 2017.

M. Kirby and L. Sirovich, “Application of the Karhunen-Loeve Procedure for Signal Characterization of Human Faces,” IEEE TPAMI, 1990.

S. Koppula, L. Orosa, K. Kanellopoulos, T. Shahrozdi, R. Azizi, and O. Mutlu, “EDEN: Energy-Efficient, High-Performance Neural Network Inference Using Approximate DRAM,” in MICRO, 2019.

A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet Classification with Deep Convolutional Neural Networks,” in NIPS, 2012.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based Learning Applied to Document Recognition,” Proc. of the IEEE, 1998.

D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, “Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture,” in HPCA, 2013.

J. Liu et al., “RAIDR: Retention-Aware Intelligent DRAM Refresh,” in ISCA, 2012.

J. Liu et al., “An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms,” in ISCA, 2013.

L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, “A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems,” in PAC, 2012.

S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, “Fikker: Saving DRAM Refresh-power Through Critical Data Partitioning,” in ASPLOS, 2011.

Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu, “Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory,” in DSN, 2014.

K. T. Malladi, I. Shaeffer, L. Gopalakrishnan, D. Lo, B. C. Lee, and M. Horowitz, “Rethinking DRAM Power Modes for Energy Proportionality,” in MICRO, 2012.

http://www.micron.com/-/media/documents/products/data%20sheet/dram/mobile%20dram/low-power%20dram/lpddr3/253bhttp://www.micron.com/-/media/documents/products/data%20sheet/dram/mobile%20dram/low-power%20dram/lpddr3/253b.pdf

J. Mukundan, H. Hunter, K.-h. Kim, J. Stuecheli, and J. F. Martinez, “Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM Systems,” in ISCA, 2013.

S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, “Reducing Memory Interference in Multitasking Multicore Systems via Application-Aware Memory Channel Partitioning,” in MICRO, 2011.

O. Mutlu, “Memory Scaling: A Systems Architecture Perspective,” in IMW, 2013.

O. Mutlu and L. Subramanian, “Research Problems and Opportunities in Memory Systems,” in SUPERFRI, 2015.

M. Patel, J. S. Kim, and O. Mutlu, “The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions,” in ISCA, 2017.

B. Pourshirazi and Z. Zhu, “Refree: A Refresh-free Hybrid DRAM/PCM Main Memory System,” in IPDPS, 2016.

W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, “Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing,” in ISCA, 2013.

M. Qureshi, D.-H. Kim, S. Khan, P. Nair, and O. Mutlu, “AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems,” in ISCA, 2015.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning Internal Representations by Error Propagation,” in Techn., Rep., 1985.

R. Shi, Z. Xu, Z. Sun, M. Peemen, A. Li, H. Corporea, and D. Wu, “A Locality Aware Convolutional Neural Networks Accelerator,” in DSD, 2015.

T. F. Smith, M. S. Waterman et al., “Identification of Common Molecular Subsequences,” Journal of molecular biology, 1981.

H. Song, L. Xingyu, M. Huizi, P. Jing, P. Ardavan, A. H. Mark, and J. D. William, “EIE: Efficient Inference Engine on Compressed Deep Neural Network,” in ISCA, 2016.
Prof. Onur Mutlu is a Professor of Computer Science at ETH Zürich. He is also a faculty member at Carnegie Mellon University, where he previously held the William D. and Nancy W. Strecker Early Career Professorship. His current broader research interests are in computer architecture, systems, and bioinformatics. He is especially interested in interactions across domains and between applications, system software, compilers, and microarchitecture, with a major current focus on memory and storage systems. A variety of techniques he, together with his group and collaborators, have invented over the years have influenced industry and have been employed in commercial microprocessors and memory/storage systems. He obtained his PhD and MS in ECE from the University of Texas at Austin and BS degrees in Computer Engineering and Psychology from the University of Michigan, Ann Arbor. His industrial experience spans starting the Computer Architecture Group at Microsoft Research (2006-2009), and various product and research positions at Intel Corporation, Advanced Micro Devices, VMware, and Google. He was the recipient of the ACM SIGARCH Maurice Wilkes Award, the inaugural IEEE Computer Society Young Computer Architect Award, the inaugural Intel Early Career Faculty Award, faculty partnership awards from various companies, a healthy number of best paper or Top Pick paper recognitions at various computer systems and architecture venues. He is an ACM Fellow, an IEEE Fellow "for contributions to computer architecture research and practice", and an elected member of the Academy of Europe (Academia Europaea). His computer architecture course lectures and materials are freely available on YouTube, and his research group makes software artifacts freely available online. For more information, please see his webpage at http://people.inf.ethz.ch/omutlu/.