ROP: Alleviating Refresh Overheads via Reviving the Memory System in Frozen Cycles

Ping Huang*†, Wenjie Liu*, Kun Tang†, Xubin He†‡✉, and Ke Zhou*
*Huazhong University of Science and Technology,Wuhan, China
†Department of Electrical and Computer Engineering, Virginia Commonwealth University, USA
‡Department of Computer and Information Sciences, Temple University, USA

***Abstract*—DRAM memory performs periodic refreshes to prevent data loss due to charge leakage, while memory refreshes cause performance degradation and energy consumption, referred to as refresh overheads. In this paper, we propose *Refresh-Oriented Prefetching (ROP)* to alleviate memory refresh overheads. Before a refresh starts, *ROP* prefetches cache lines from the to-be-refreshed rank into an added SRAM buffer. In doing so, when a rank is undergoing refresh, memory requests can still be serviced rather than being blocked. At the core of *ROP* is a probabilistic prefetch model determining which cache lines are prefetched for a refresh based on the access patterns appearing in an observational window ahead of the refresh. A *Pattern Profiler* collects statistics about memory traffic occurring before and after the starting time of each refresh operation in a period of training time and it outputs two conditional probabilities which are used to control subsequent prefetch decisions. A *Prefetcher* maintains a prediction table which helps to ascertain access patterns appearing around refresh operations. The prediction table is updated every time an access occurs to the to-be-next-refreshed ran during the observational window and is consulted to decide which cache lines are prefetched. Extensive evaluation results with benchmarks from SPEC CPU2006 on a DDR4 memory have demonstrated that with *ROP* memory performance can be improved by up to 9.2% (3.3% on average) for single-core simulations, while reducing the overall memory energy by up to 6.7% (3.6% on average), relative to an auto-refresh baseline memory. Moreover, it increases the *Weighted Speedup* by up to 2.22X (1.32X on average) for 4-core multiprogram simulations, while reducing energy by up to 48.8% (24.4% on average).**

I. Introduction

DRAM memory has long been used as the primary building blocks to architect main memory systems. As DRAM density has increased dramatically and its cost becomes more affordable, contemporary DRAMs are not only used in memory systems in desktop computers, but also deployed to construct large high-performance DRAM-based storage systems [1]. Though DRAM capacity has significantly enlarged across different generations, the latency has not reduced commensurately [2]. Moreover, the increasing popularity of multicore and multiprogrammed applications has aggravated the "memory wall" problem due to mutual interference [3] and lack of locality [4].

DRAM cells are leaky and the stored data can only persist for a limited period of time, the so-called retention time [5], [6]. To guarantee data integrity, the memory controller periodically issues refresh operations to replenish memory cell charges. However, periodically performing refresh operations brings about two negative ramifications. First, frequently refreshing consumes a significant amount of energy. Even worse, the dissipated power could increase ambient temperature which results in shorter retention time. Shorter retention time in turn leads to more frequent refreshing [5], [7]. Second, refresh operations negatively impact memory latency in at least three ways: *Scheduling Delay*, *Execution Conflict*, and *Resource Contention*. Refresh commands are perceived equally as regular memory requests when the memory controller chooses commands to be issued. When a refresh operation is required, the memory controller needs to enforce a refresh operation before issuing a regular memory request, prolonging request *Scheduling Delay*. Moreover, when a refresh operation is in progress, part of DRAM resources are locked for refreshing and become unavailable. Requests that happen to access the locked resources have to wait until on-going refreshes complete, causing *Execution Conflict*. Furthermore, blocked requests consume resources which could otherwise be used to service other non-conflicting requests while a refresh is being performed, resulting in hogged *Resource Contention*. For example, blocked requests could cause *command queue seizure* [7] phenomenon that might prevent other non-conflicting requests from progressing. Previous researches have reported that refresh operations cause up to 10% performance degradation [8], [9] and the refresh penalty will become even larger in future high density memory [10], [11].

A lot of research efforts have been devoted to attack the "memory wall" problem, which has become increasingly challenging as memory capacity increases [12] and 3D-stacked DRAM emerges [13], [14]. Not surprisingly, refresh operations, which impose significant overheads on memory access latency, have particularly been researched. Existing refresh-overheads mitigation techniques mainly fall into two broad categories: **refresh reduction** and **refresh hiding**. *Refresh reduction* aims to reduce refresh operations via exploiting memory retention time differences in two ways. One is that, the memory controller keeps track of the retention periods of underlying cells and opportunistically skips refreshing cells which still have long enough remaining retention time when a refresh is performed [15], [16]. The other one is to exploit the differences in retention time at software layer to decrease refreshes [17], [18]. *Refresh hiding* is mainly realized via smart scheduling or concurrent refreshing. Smart scheduling prioritizes to schedule memory requests over refresh operations

2332-5690/16 $31.00 © 2016 IEEE
DOI 10.1109/ICPP.2016.26

when conflicts occur [8], [9]. Concurrent refreshing allows refreshes and memory accesses to execute simultaneously by taking advantage of the internally parallel architectural components [19], [20] so as to hide refresh overheads.

In this paper we suggest a new refresh mitigation technique named *Refresh-Oriented Prefetching (ROP)* . The central idea is motivated by several critical observations on how refresh operations affect memory performance (Section III). We propose to add an SRAM prefetch buffer in the memory controller to stage the cache lines which are predicted to be very likely accessed while the parent rank is undergoing a refresh operation in the near future. Cache line prefetching essentially enables a refresh and memory requests to proceed in parallel without delaying neither of them if the cache lines which will be accessed during the refresh period have already been prefetched in the buffer before the refresh starts. We employ a probabilistic prefetch model to decide the cache line candidates for prefetching. The prefetch model mainly consists of two components, i.e., *Pattern Profiler* and *Prefetcher*. The *Pattern Profiler* collects statistics about memory behaviors happening before and after refresh times for a training period and calculates two conditional probabilities which are used to enable/disable prefetching in a probabilistic manner. The *Prefetcher* relies on a prediction table to ascertain access patterns in an observational window before each refresh operation. If prefetching is enabled, it prefetches cache lines according to the observed patterns in the prediction table. It should be noted that prefetching in *ROP* is only enabled for a brief period of time (i.e., an observational window) before refresh operations, which is a distinguishing feature from conventional prefetchers. Our evaluation results have shown that *ROP* is able to significantly alleviate refresh overheads in modern memory systems.

This paper makes the following contributions:

- We present a comprehensive analysis on the memory refresh problem from different angles. The findings motivate the design of our proposed refresh approach.
- We design a new probabilistic prefetch model which prefetches cache lines into an SRAM buffer in the memory controller before a refresh starts such that memory requests can be satisfied even when the target rank is being refreshed.
- We conduct extensive evaluations with benchmarks from SPEC CPU2006. The results have shown that our system can successfully alleviate refresh overheads, improving both performance and energy efficiency relative to an auto-refresh baseline memory.

The rest of this paper is structured as follows. In Section II, we give background knowledge about memory systems and refresh operations. In Section III, we study refresh operations from various aspects via investigating memory behaviors at a microscopic level. Section IV focuses on system design details, particularly the profiling mechanism and prefetching algorithm. Section V presents our evaluation methodology and evaluation results. Section VI discusses related work. Finally, we conclude our paper in Section VII.

II. Background

A. DRAM Basics and Organization

Each DRAM cell consists of one capacitor which stores charges and one access transistor. DRAM cells are arranged as two-dimensional subarrays. All the access transistors in a row share the same enabling signal wire called wordline, and all transistors in a column are connected to the same bitline. The bitlines inside a subarray are all connected to a *sense amplifier* (a.k.a, *row buffer*) where voltage signals are amplified [21]. Each subarray has its own local address decoders and row buffer and can operate independently, providing high access parallelism. To fetch a cache line from DRAM cells, the row address is first asserted to the corresponding wordline, which drives a whole row content to appear in the row buffer. The procedure of driving a whole row of data into the row buffer is called *row activation*. After activating a row, the column address is decoded and the cache line is fetched from the row buffer. In *open page* mode, if the following request hits the same row (i.e., row buffer hit), the row activation latency is obviated, resulting in better performance. However, if the following request accesses a different row, the memory controller need to first close the currently opened row and then activate a new row, increasing access latency.

DRAM system is architected using basic subarrays to form a hierarchical structure, including channels, ranks, banks, and subarrays. Each bank consists of multiple subarrays. All subarrays inside one bank take turns sharing a global row buffer and the same set of I/O buses. Though banks can operate simultaneously to service requests, memory requests that access the same bank can still result in bank conflicts and must be serviced sequentially. Each rank contains multiple banks and all banks inside one rank work in a lockstep manner. At the highest level, DRAM is partitioned into different channels, each of which contains a number of ranks. To service a request, the memory controller first interprets the memory address as a combination of channel, rank, bank, subarray according to address mapping scheme and then uses those information to traverse memory hierarchy to locate the data.

B. Refresh Operation

DRAM refresh can be performed in two modes, *bursty refreshing* and *distributed refreshing*. In bursty refreshing, the whole DRAM device is tied up until all DRAM cells have finished refreshing. This mode is unfavorable as it causes a long time of device unavailability. In distributed refreshing, only a limited number of DRAM rows (called *refresh bundles*) are refreshed by one refresh operation, and multiple refreshes have to be issued in the retention time window to guarantee that every row gets a chance to be refreshed before retention time expires. By dividing the total refreshing time into small pieces, the probability of request blocking is reduced and the average waiting time is decreased. The frequency of refresh operations (T_{REFI}) is mainly determined by retention time[1]

[1] The retention time is generally considered to be 64ms when the temperature is $< 85°$C and 32ms when the temperature is $> 85°$C.

and the number of bundles contained in a memory system. Due to historical reasons [9], modern memory arrays are typically divided into 8K groups. As a result, the refresh interval T_{REFI} ($\frac{64ms}{8K} = 7.8\mu sec$ and $\frac{32ms}{8K} = 3.9\mu sec$ under normal and high temperature, respectively) has remained the same across different generations. Moreover, since the time taken to refresh one row (T_{RC}) remains almost unchanged across generations, the time needed to refresh one bundle (T_{RFC}) has increased dramatically as the number of DRAM rows included in a bundle increases, resulting in decreased DRAM availability [9], i.e., increased *Refresh Duty Cycle* T_{RDC}, which is defined as the ratio of T_{RFC} to T_{REFI}). JEDEC DDR4 standard [21] has suggested Fine-Grained Refresh (FGR) to reduce T_{RFC} by increasing the frequency of refresh operations (i.e., by decreasing T_{REFI}).

III. DRAM Refresh: A Close Look

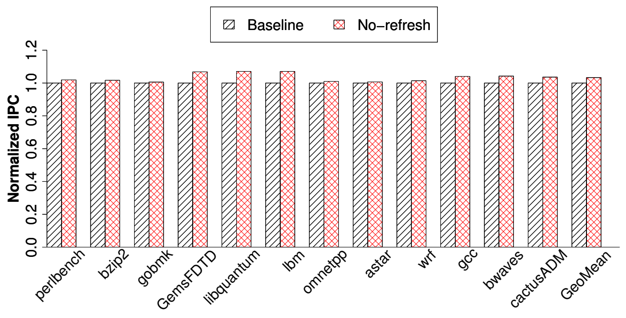
As discussed before, refresh operations cause overheads to memory system in various respects. In this section, we take an in-depth look at the refresh impacts. We run the SPEC CPU2006 benchmarks and collect the memory traces for post-analysis. Each benchmark trace includes the memory request and refresh timing information based on which we can obtain various characteristics of refresh operations. The experimental setup details are referred to Section V.

A. Overall Overheads

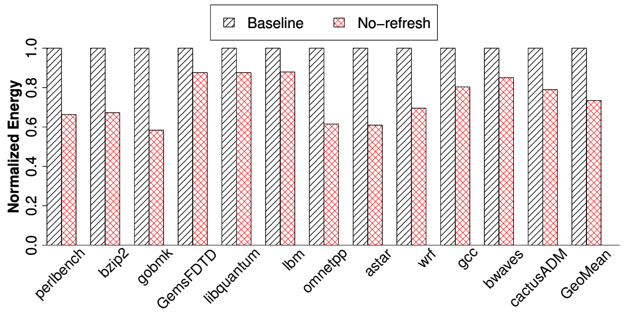
The most noticeable refresh impacts are performance degradation and increased energy consumption, since refreshes interfere with regular memory requests and consume power. We evaluate refresh overheads by running benchmarks on a regular memory system (denoted as *baseline*) and an idealized no-refresh memory (denoted *no-refresh*) and comparing their performance and energy consumption. The performance and energy disparities between *baseline* and *no-refresh* reflect refresh overheads and are the bounded limits of refresh mitigation techniques. Figure 1 shows the comparison results. As it is shown, refresh causes up to 7.3% performance degradation, with an average of 3.3% across the examined benchmarks and incurs extra energy consumption, consuming up to 41.6% more energy, with an average of 26.5%.

B. Requests Blocked by Refresh

We define a memory request as "blocked" if it arrives within the time window of an examined period following the refresh starting time. Suppose the length of refresh cycle is L and a refresh happens at time T, then any requests arriving in the time interval between T and $T + L$ would be blocked and the corresponding refresh is regarded as "blocking" refresh. If there is no request arriving during that time interval, then the corresponding refresh is regarded as "non-blocking". As write requests can be buffered, we assume only read requests would be blocked. For analysis purpose, we investigate three cases, where the length of examined period equals to 1×, 2×, and 4× that of the refresh period, respectively. Figure 2 shows the percentage of non-blocking refresh operations for each benchmark. It shows that an impressive amount of refreshes



(a) Performance degradation.



(b) Energy consumption.

Fig. 1: Performance and energy omparisons between the baseline and an idealized memory. On average, refresh incurs 3.3% performance degradation and 26.5% additional energy.

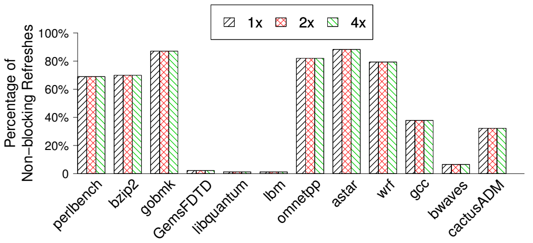


Fig. 2: The percentage of non-blocking refreshes in different examined periods, i.e., 1x, 2x, 4x of the refresh cycle (T). As it is shown, many refreshes do not block memory requests.

actually do not block memory requests in all three scenarios. Particularly, the percentages of non-intensive benchmarks are rather high, achieving an average of 79.3%. One of the reasons is that the last level cache (LLC) in the processor has filtered out many memory traffic and creates bursty access patterns to the memory, reducing conflict occurrences. This observation reveals that refresh overheads most of the time are caused by a small percentage of refreshes. For those blocking refreshes, we calculate the number of requests blocked by each refresh. Figure 3 shows the average number of requests blocked by "blocking" refreshes for each benchmark. As it is shown in the figure, on average each refresh blocks only a small number of requests. We have observed that the maximum number of requests blocked by a refresh is only 12.

This analysis helps us better understand how refresh impacts memory requests: 1) most refreshes do not block memory requests and, 2) only a few requests are blocked by each individual blocking refresh. In spirit, the design of ROP is inspired by the two observations. First, since not every refresh

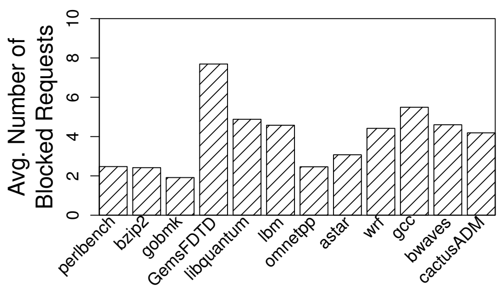


Fig. 3: The average number of blocked requests by refresh operations for each benchmark. On average, each blocking refresh blocks only a marginal number of requests.

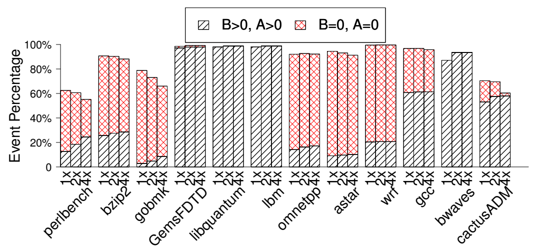


Fig. 4: The percentage of the two dominant types of refreshes. Predicting these two events can achieve a very high level of prediction coverage.

blocks requests, aggressively prefetching cache lines for every refresh would cause unnecessary overheads. Correspondingly, we employ a probabilistic prefetch model to make prefetching decisions based on access patterns appearing before refresh operations. Second, since each refresh blocks a small number of requests, it thus only requires a small amount of SRAM buffer space for prefetching cache lines during refresh periods.

C. Access Correlation

In this section, we explore the relationship between access behaviors occurring before and after refreshes by investigating the occurrences of memory requests. Based on the number of request occurrences, a refresh can be classified as one of the four types defined in Section IV-B. We are interested in the occurrences of the following two events, E1.) $B > 0$ and $A > 0$ and E2.) $B = 0$ and $A = 0$. Please note that B denotes the number of memory requests occurring in a time window before a refresh, and A denotes the number of memory requests occurring in a time window after the refresh. $E1$ indicates the event that there exist requests in the time windows before and after a refresh, and $E2$ represents the event that there is no request occurring in both time windows. In prefetching context, the frequencies of $E1$ and $E2$ indicate the effective prefetches based on $B > 0$ and the correct no-prefetching decisions based on $B = 0$ the prefetcher would achieve, respectively. Figure 4 shows the percentages of these two events for the examined benchmarks under different window lengths. As it is shown, these two events are the dominant events across all benchmarks, meaning the prefetcher can achieve a good prediction coverage by only predicting the two events. Since *ROP* makes predictions based on the value of B, the conditional probabilities of $P\{A > 0/B > 0\}$ (denoted by λ) and $P\{A = 0/B = 0\}$ (denoted by β) represent the prediction accuracies. Table I gives the values of these two conditional probabilities. As can be seen, for most benchmarks, those two probabilities are rather high, meaning that prefetching decisions made based on the value of B could be reasonably accurate. Take *bzip2* for example, if there is at least one request in the observational window of the length of 1x refresh period preceding a refresh starts, the prefether can perform prefetching with the confidence that prefetched cache lines may be accessed during the refresh period at a probability of 0.84. Similarly, if there is no request observed in the observational window, the prefetcher assumes no need for prefetching at a probability of 0.94. *ROP* leverages the two conditional probabilities to make prefetching decisions, which avoids over-prefetching and does not miss potentially useful opportunity with a reasonably high level of confidence. Another conclusion drawn from Table I is that, the values of λ and β are insensitive to the length of observational window. Therefore, we use 1x refresh period length of observational window in our experiments.

IV. Refresh-oriented Prefetching

A. Architecture Overview

As we have seen in Section III, memory refresh affects system performance by interrupting application executions due to the required data being inaccessible during refresh periods. A refresh consumes much longer time than a memory request and memory requests conflicting with a refresh have to stall until the refresh finishes. We propose to reduce refresh overheads by judiciously prefetching cache lines which will be likely read by applications during the following refresh period into an SRAM buffer in the memory controller before the rank is locked up for refreshing. The hope is that memory requests arriving at the rank during refresh period can still be satisfied while the rank is being refreshed, as the requested cache lines have already been prefetched into the buffer ahead of time.

Figure 5 shows a simplified architectural view of our proposed memory system. There are many other functional components (e.g., command queue, transaction queue) in a realistic memory controller, but we omit them for simplicity and only show the two components that interact with newly added components. As it is shown, we have added four new modules in the memory controller to support our idea. The function of *Pattern Profiler* is to observe application memory behaviors occurring before and after refresh operations in training phases. As the result of pattern profiling, it outputs a number of probability values which are provided to the *Prefetcher* to help making prefetching decisions in a probabilistic manner. The *Prefetcher* deploys a prefetching algorithm to predict which cache lines will be needed by requests arriving at the rank during the forthcoming refresh period. It makes predictions based on the access patterns encountered during a configurable length of time preceding the refresh. Finally, when the refresh is about to start, the *Prefetcher* decides whether or not to fetch

TABLE I: The values of λ and β

	perlbench	bzip2	gobmk	GemsFDTD	libquantum	lbm	omnetpp	astar	wrf	gcc	bwaves	cactusADM	Average
1x Refresh Period													
λ	0.40	0.84	0.20	0.99	0.99	0.99	0.78	0.76	0.99	0.97	0.93	0.78	0.80
β	0.73	0.94	0.88	0.68	0.04	0.00	0.95	0.97	1.00	0.96	0.00	0.54	0.64
2x Refresh Period													
λ	0.41	0.79	0.20	0.99	0.99	0.99	0.74	0.67	0.99	0.96	0.93	0.74	0.78
β	0.77	0.96	0.89	1.00	0.77	0.00	0.98	0.98	1.00	0.98	0.00	0.54	0.74
4x Refresh Period													
λ	0.39	0.73	0.22	0.99	0.99	0.99	0.71	0.59	0.98	0.95	0.93	0.66	0.76
β	0.83	0.98	0.93	1.00	0.91	0.00	0.99	0.98	1.00	0.98	0.00	0.20	0.73

the predicted cache line candidates into the fully-associative SRAM buffer by allowing for the probability values from *Pattern Profiler*. Both *Pattern Profiler* and *Prefetcher* rely on the refresh timing information available in *Refresh Manager*. Ranks sharing the same refresh circuit (e.g., ranks in the same memory channel) in a memory system take turns leveraging the *SRAM Buffer* to hold prefetched cache lines. *Rank-aware Mapping*, which is in spirit similar to bank partitioning [22], tries to minimize interleaved access patterns from concurrently running applications. It avoids interference by assigning different ranks to applications such that memory accesses to the same rank may exhibit predictable patterns.

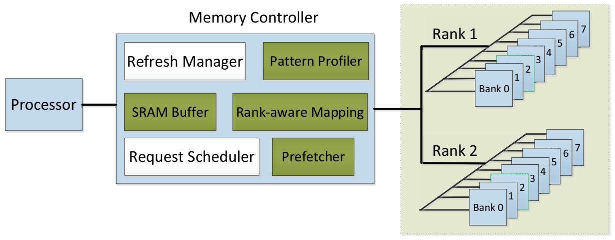


Fig. 5: A simplified architectural view of *ROP*. The *Prefetcher* prefetches cache lines into an *SRAM Buffer* before refresh operations lock up a rank. Requests can possibly be satisfied even when the target rank is being refreshed.

B. Profiling Access Patterns

The success of *ROP* depends on the usefulness of prefetched data in the *SRAM Buffer*. Therefore, it is important to prefetch the correct data into the buffer. To improve prediction accuracy, *ROP* dynamically divides the whole application execution time into different phases, each of which consists of two steps, i.e., *profiling* and *predicting*. The purpose of profiling is to observe access dynamics in a configurable length of training time to ascertain the most recent features so that the accuracy of predictions can be dynamically optimal. During the profiling period, the SRAM buffer is turned off to save leakage power and the *Pattern Profiler* collects statistics about the amount of memory accesses occurring to the target rank before and after each refresh operation. At the end of a training time, it tries to probabilistically correlate the patterns observed before and after refreshes by calculating several event probability values which are then used in the next *predicting* step to guide cache line prefetching. More specifically, it categorizes refreshes into four categories based on the number of read and write requests happening before a refresh and the number of read requests happening after a refresh in a configured time window (called *Observational Window*). Let B and A denote the amount of requests observed in the observational windows preceding and following a refresh, respectively. A refresh falls into one of the following four categories: (1) $B > 0$ && $A > 0$, (2) $B > 0$ && $A = 0$, (3) $B = 0$ && $A > 0$, and (4) $B = 0$ && $A = 0$. Using collected statistics, *Pattern Profiler* counts the occurrences of each category. Based on the occurrences, it outputs the following two conditional probability values λ and β, defined as Equation 1 and Equation 2, respectively.

$$\lambda = P\{A > 0/B > 0\} = \frac{P\{B > 0 \,\&\&\, A > 0\}}{P\{B > 0\}} = \frac{P\{B > 0 \,\&\&\, A > 0\}}{P\{B > 0 \,\&\&\, A = 0\} + P\{B > 0 \,\&\&\, A > 0\}} \tag{1}$$

$$\beta = P\{A = 0/B = 0\} = \frac{P\{B = 0 \,\&\&\, A = 0\}}{P\{B = 0\}} = \frac{P\{B = 0 \,\&\&\, A = 0\}}{P\{B = 0 \,\&\&\, A = 0\} + P\{B = 0 \,\&\&\, A > 0\}} \tag{2}$$

The value of λ represents the probability of there being read requests arriving during the next refresh period under the condition that there have been observed requests during the observational window preceding the refresh. The value of β represents the probability of there being *no* read requests arriving during the next refresh period under the condition that there have been *no* observed requests during the observational window preceding the refresh. These two numbers are then used by the prefetcher (discussed in Section IV-C) to throttle cache line prefetching. For example, suppose $\lambda = 0.8$ and $\beta = 0.7$, the prefetcher performs cache line prefetching with a confidence level of 0.8 if it has seen requests in the observational window and does not perform cache line prefetching with a confidence level of 0.7 if there was no requests. As Table I shows, most benchmarks have pretty high values of both λ and β, implying that the decision of whether to prefetch or not based on the number of requests observed in the observational window ahead of the refresh operation most of the time is accurate.

C. Probabilistic Prefetch Model

In the previous subsection, we have discussed how the *Pattern Profiler* characterizes memory access behaviors during the training time and calculates two conditional probabilities which act as throttling knobs to control whether prefetching should be performed or not in the prefetching phase. It is the *Prefetcher* 's responsibility to determine which cache lines are most likely needed during refresh period. We employ a variation of the recently proposed *Variable Length Delta*

Prefetcher (VLDP) [23] prefetching algorithm to predict the cache lines for each refresh. VLDP has been demonstrated to be more effective than other stat-of-the-art prefetchers due to the commonality of multi-delta access patterns observed at the memory controller [23]. The original VLDP algorithm employs a Delta History Buffer (DHB) table to remember historical patterns of accessed pages and up to four Delta Predication Tables (DPT) to infer the next cache line that would be accessed. The entries in the DHB are used to lookup DPTs and predict future memory requests. The salient feature of VLDP algorithm is that it can detect multi-delta patterns. To accommodate our situation, we make necessary changes to the VLDP algorithm. First, since our algorithm tries to predict cache line accesses at the memory rank scope rather than cache line accesses in the same page, we need to maintain access history for individual ranks. Second, we use only one prediction table to record memory access history. Each table entry records the observed access patterns to a bank in the rank during an observational window. The number of entries in the prediction table is equal to the number of banks in a rank. Many applications exhibit bank locality [22] and thus such table organization can improve prediction accuracy. Each table entry contains the following fields: *BankID, LastAddr, Delta1, f1, Delta2, f2, Delta3, f3*, as it is shown in Figure 6. *BankID* refers to the bank on which access patterns are observed. *LastAddr* is the address (i.e., cache line offset within the bank) of the last access in the bank. A delta is defined as the address difference between two consecutive cache line accesses. The three delta fields and their associated frequencies are used to remember three different patterns (1 delta, 2 deltas, and 3 deltas, respectively) and how many times the patterns have appeared. During the observational window, when a cache line is accessed, the *Prefetcher* calculates the address difference (denoted as D) between the new address and the stored *LastAddr*. If D equals to *Delta1*, then *f1* is incremented by 1; otherwise, *Delta1* is replaced with D and *f1* is reset to zero to indicate a new pattern. Every two accesses generate a tuple of two deltas. The *Prefetcher* compares the new two-delta with the previously remembered *Delta2*. If they are the same, meaning a repeated two-delta pattern has been observed, then *f2* is incremented by 1; otherwise, *Delta2* is replaced with the new two-delta and *f2* is reset to zero to indicate a new two-delta pattern. *Delta3* and *f3* are updated in a similar manner, but in every three accesses. When any of the three frequencies overflows (which never happens in our evaluation), all of them are reduced to a half. Finally, it updates the *LastAddr* to the new address. As it is shown, each table entry requires 204 bits. Therefore, for a rank comprising 8 banks, the prefetching algorithm only requires 204B storage overhead.

BankID	LastAddr	Delta1	f1	Delta2	f2	Delta3	f3
4 bits	26 bits	26 bits	6 bits	52 bits	6 bits	78 bits	6 bits

Fig. 6: Data fields of prediction table entry.

When the target rank is about to be refreshed, the *Prefetcher* fetches cache lines into the *SRAM Buffer* according to the identified patterns in the prediction table. Assume the capacity of the *SRAM Buffer* is C (cache lines) and the three frequency numbers in bank i are $f1_i$, $f2_i$, and $f3_i$, respectively. The amount of cache lines (B_i) prefetched from bank i is given by Equation 3 (N is the number of banks in a rank). Within bank i, the amount of cache lines following the three identified patterns relative to *LastAddr* are $\frac{f1_i * B_i}{f1_i+f2_i+f3_i}$, $\frac{f2_i * B_i}{f1_i+f2_i+f3_i}$, and $\frac{f3_i * B_i}{f1_i+f2_i+f3_i}$, respectively.

$$B_i = \frac{f1_i + f2_i + f3_i}{\sum_{j=1}^{N}(f1_j + f2_j + f3_j)} \times C \tag{3}$$

To summarize, at any given time, the memory can be in one of three states, namely *Training*, *Observing*, and *Prefetching*. In *Training* state, the *Pattern Profiler* analyzes the memory requests appearing before and after each refresh operation and outputs the values of λ and β when the training period ends and the memory transitions to *Observing* state. The duration of *Training* state can be specified as the number of refresh operations that should be included in the training period. In *Observing* state, the *Prefetcher* observes the memory requests occurring in an observational window ahead of each refresh operation and then based on the observations and values of λ and β it decides whether to perform prefetching or not. Specifically, if there are observed requests, it performs prefetching at a probability of λ; if there are no observed requests, it does not perform prefetching at a probability of β. If prefetching is decided, the memory transitions to *Prefetching* state and the predicted cache lines are fetched into the *SRAM Buffer* before a refresh starts; otherwise it continues to remain in *Observing* state. Finally, if the hit rate on the SRAM buffer falls below a threshold, the memory transitions back to *Training* state.

D. Prefetch Request

After the memory has decided to prefetch predicted cache lines into the buffer, it generates memory requests to fetch those cache lines. However, blindly issuing prefetch requests may cause performance degradation due to interference with demand requests. To minimize conflicts, we make the following two optimizations. First, memory requests addressing the rank which is to be refreshed next are drained before the rank is locked for refresh, as proposed in [7]. As a refresh takes a non-trivial period of time, draining memory requests can avoid request housekeeping resources being occupied for an entire refresh period. Second, prefetch requests are put in a dedicated queue and then opportunistically issued together with drained requests if they access the same row. In doing so, prefetching overhead is minimized since row buffer hit accesses can be accomplished much faster. The remaining requests are issued before the refresh starts. Overall, the presence of prefetching may delay individual refreshes. However, we believe that is not be a big concern, as JEDEC standard allows a refresh command to be delayed by up to 8 refresh cycles, as long as the average refresh rate is one per refresh cycle [8].

It should be noted that the *ROP* prefetcher is different from conventional prefetchers in the processor side. First, *ROP* is located in the memory controller and it aims to alleviate refresh overheads, while other prefetches are located in the processor to hide the latency of accessing underlying memory hierarchy (e.g., L2, LLC or DRAM). Second, due to limited visibility, conventional prefetchers cannot eliminate refresh overheads well as they do not know when refreshes happen. It is even possible that those prefetchers issue prefetch requests which are unfortunately blocked by refreshes in the memory system, wasting memory bandwidth between processor and off-chip memory. Third, conventional prefetchers perform prefetching constantly, while *ROP* only prefetches for brief periods of time preceding refreshes. Finally, *ROP* can be complementary to those prefetchers to further improve memory performance by mitigating refresh overheads.

V. Evaluation

A. Experimental Methodology

We implement *ROP* in the popular DRAMSim2 [24] simulator. We use the Zsim [25] simulator to run benchmarks. The processor simulator is coupled with the Pin [26] tool as the front-end. We use benchmarks from the SPEC CPU2006 suite for evaluations. Table II lists the benchmarks. The benchmarks include both memory intensive and non-intensive benchmarks. The observational window is set a refresh period and the training time is configured to include 50 refreshes. The hit rate threshold is conservatively set to 0.6. We carry out both single-core and 4-core evaluations. For single-core evaluations, only one benchmark is run at a time. For 4-core evaluations, 4 benchmarks are run concurrently on a 4-rank memory with rank-partitioning to assign each benchmark to a different rank. As shown in Table II, we use six benchmark combinations, which represent a diverse mixing of the memory intensive and non-intensive benchmarks. The main memory parameters are given in Table III. The SRAM buffer parameters are obtained using the CACTI 5.3 tool. Micro power calculator [27] is used to estimate memory power consumption. Unless otherwise noted, the SRAM capacity is set to 64 cache lines and the LLC is set to 2MB and 4MB for singe-core and multi-program experiments, respectively. For each benchmark, we fast-forward 1 billion instructions for warm-up and run 1 billion instructions for comparisons. We compare *ROP* with an auto-refresh baseline memory and an idealized no-refresh memory to evaluate its benefits. However, comparisons with other refresh schemes can be extrapolated as other refresh papers compare with the same baseline and idealized no-refresh memory system.

B. Single-core Experimental Results

1) Performance Improvement: Figure 7 compares performance using instruction per cycle (IPC) in single core configurations. Two conclusions are in order from this figure. First, the performance of *ROP* is consistently closer to the performance of No Refresh for the benchmarks with different SRAM buffer sizes, demonstrating that *ROP* effectively alleviates refresh overheads. ROP achieves up to 9.2% performance improvement relative to the baseline memory. Second, it is interesting to observe that *ROP* even slightly outperforms an idealized memory for many benchmarks due to the better performance of SRAM than DRAM.

TABLE II: Benchmarks for Evaluation

Name	Intensive	WL1	WL2	WL3	WL4	WL5	WL6
GemsFDTD	Y	√					
lbm	Y	√					
bwaves	Y	√	√		√		
gcc	Y	√	√		√		
libquantum	Y		√	√			
cactusADM	Y		√	√			
wrf				√		√	
bzip2				√		√	
perlbench					√	√	√
astar					√	√	√
omnetpp							√
gobmk							√

TABLE III: Memory Parameters

Processor	Single core/4 cores (out-of-order)
Memory Controller	64/64-entry read/write request queue FR-FCFS, writes are scheduled in batches
DRAM Configurations	DDR4-1600, 1 channel, 1 rank for single core, 4 ranks for 4 cores.

Refresh Parameters	tREFI=7.8μs,tRFC=350ns
	for 8Gb DRAM chips with 1x refresh mode

SRAM Paremeters	SRAM read/write energy=0.0132nJ/ 0.0135nJ/0.0137nJ/0.0152nJ access latency=3 cycles for 16/32/64/128 SRAM slots

2) Energy Consumption: Figure 8 shows the memory energy comparisons. As it is shown, *ROP* reduces the total amount of memory energy consumption relative to the baseline memory, even though it does not reduce the number of refresh operations and the introduction of the SRAM slightly increase memory power. The main reason is because *ROP* shortens the total execution time, which causes reduced dynamic energy. For example, *libquantum* has the most performance improvement, while it saves the most energy savings.

3) SRAM Buffer Hit Rate: Unlike using prediction coverage and accuracy to characterize conventional prefetchers, we use buffer hit rate as the metric to evaluate the efficacy of the SRAM buffer, as the *ROP* is quite different from other conventional prefetchers as discussed in Section IV-D. Hit rate is defined as the ratio of "the number of requests serviced by the SRAM buffer" to "the total number of requests arriving during a refresh period". This metric reflects how often and to what degree ROP eliminates memory stalls and is a good indicator of the effects of ROP in reducing the blocked requests. Figure 9 shows the average prefetch buffer hit rates in single-core experiments. As can be seen, the SRAM buffer constantly delivers a hit rate above 0.6. It implies that our prefetcher does prefetch the right cache lines which are then accessed during refresh periods. In addition, as SRAM buffer capacity increases from 16 cache lines to 128 cache lines, the hit rates exhibit an increasing trend.

C. Multi-program Experimental Results

In this section, we present the experimental results of multi-programmed benchmarks. We use weighted speedup (defined as Equation 4) as the metric to compare their performance. IPC_i^{shared} and IPC_i^{alone} represent the performance of the

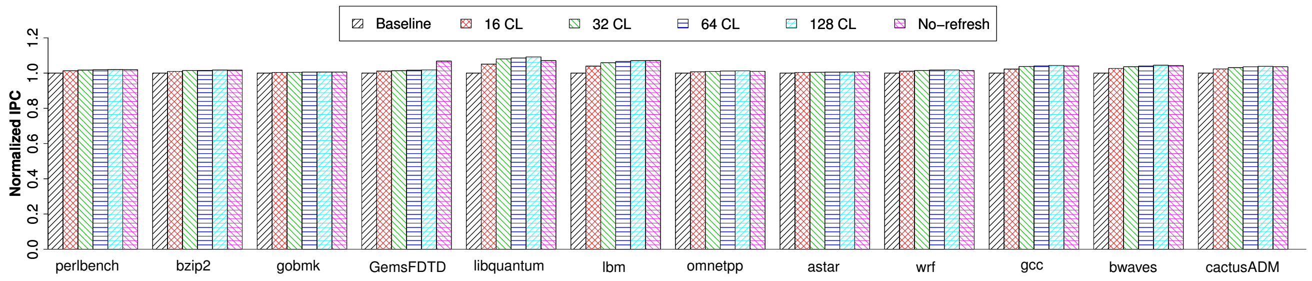


Fig. 7: Performance comparison in singe-core experiments. Values are normalized to the baseline. *ROP* improves performance by up to 9.2% and 3.3% on average.

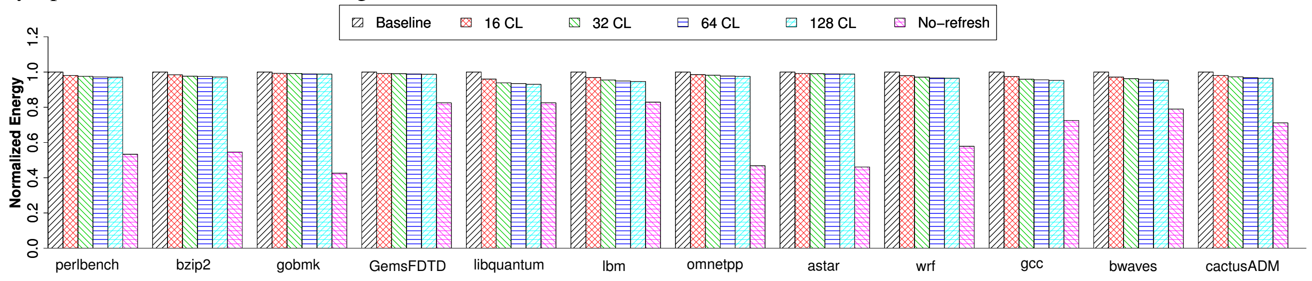


Fig. 8: Memory energy comparison in singe-core experiments. Values are normalized to the baseline. *ROP* consumes less energy than the baseline memory, saving up to 6.7% and 3.6% on average.

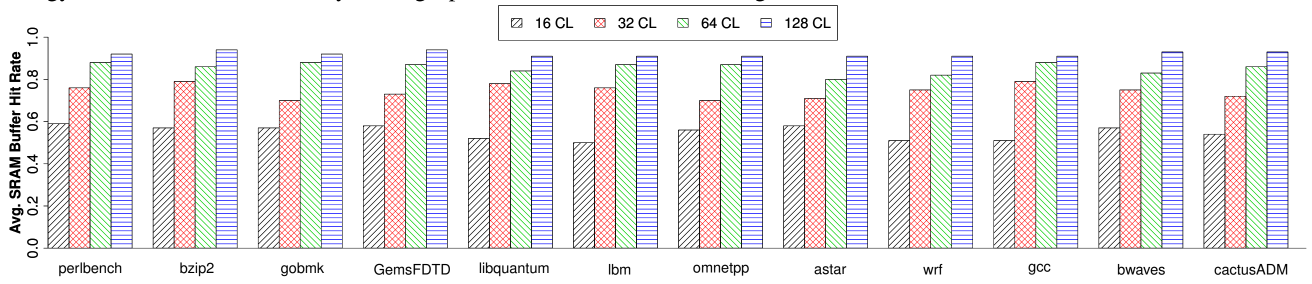


Fig. 9: The average SRAM buffer hit rates in single-core experiments.

i^{th} benchmark when it is running concurrently with other benchmarks and running alone, respectively. We compare three systems, a baseline system with auto-refresh denoted as *Baseline*, a variation of baseline system with rank partitioning denoted as *Baseline-RP*, and our proposed system *ROP*. We run the benchmarks on these three memory systems and calculate their speedups. *Baseline-RP* is for reference and its performance improvement relative to the *Baseline* shows the advantages of employing rank partitioning to avoid interference in the rank.

$$Weighted\ Speedup = \sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}} \tag{4}$$

1) Weighted Speedup: Figure 10 compares the normalized weighted speedups of the three systems. As it is shown, *ROP* is able to improve the speedups for all the three workloads. Comparing with *Baseline*, it improves speedups by a maximum of 1.8X, with a geometric mean of 1.29X. Comparing with *Baseline-RP*, *ROP* achieves a maximum speedup improvement of 18.8% for *WL1*, with a geometric mean of 6.5% across all workloads. Particularly, the more memory intensive benchmarks a workload contains (e.g., the *WL1*), the larger improvement it brings about. The reason is because memory intensive benchmarks suffer more from refresh overheads and

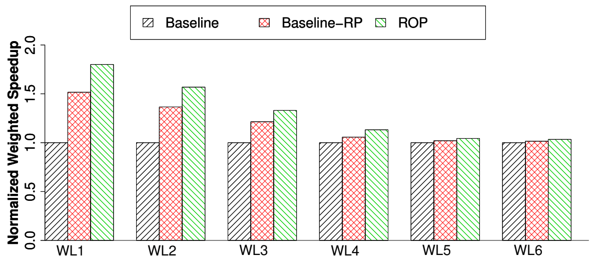


Fig. 10: Normalized weighted speedups of multi-programmed workloads. *ROP* achieves an average of 1.29X speedup improvement relative to the baseline memory.

can get greater alleviation from *ROP*.

2) Energy Consumption: Figure 11 compares the normalized energy consumption of the three memory systems. Again, *ROP* also reduces energy consumption relative to both *Baseline* and *Baseline-RP*. Comparing with *Baseline*, *ROP* reduces energy consumption by up to 40% (*WL2*), with a geometric mean of 22.6% across the workloads. Generally speaking, the more memory intensive benchmarks a workload contains, the more energy savings are achieved. This can be attributed to the similar reason as explained in Section V-B2, i.e., the execution

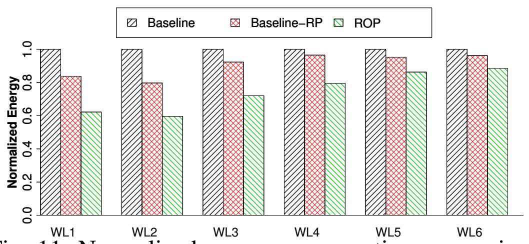


Fig. 11: Normalized energy consumption comparison.

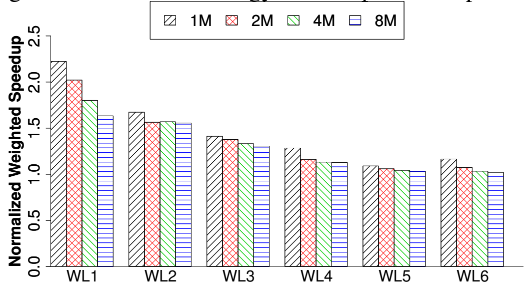


Fig. 12: Normalized weighted speedup with different LLC sizes relative to the *Baseline*.

time has been reduced more significantly and therefore less total energy has been drawn.

3) Sensitive Study on LLC Size: The last level cache in processors can filter out many memory requests and affect the access patterns presented at the memory level and therefore affect the refresh overheads that applications suffer from. To investigate the impacts that LLC may impose, we perform a sensitivity study on the LLC sizes. Figure 12, Figure 13, and Figure 14 show the performance, energy consumption, and SRAM buffer hit rates results with different LLC sizes, respectively. Three main conclusions are in order from the figures. First, *ROP* can improve performance (by up to 2.22X for *WL1* with 1M LLC, with a geometric mean of 1.32X across all workloads in all LLC sizes), reduce energy consumption (by up to 48.8% for *WL1* with 2M LLC, with a geometric mean of 24.4% across all workloads in all LLC sizes) relative to the baseline in all examined LLC sizes. It implies using a bigger LLC cannot eliminate refresh overheads and prefetching employed in *ROP* can be effective in all LLC sizes. Second, the SRAM buffer hit rate maintains an impressive high level across all different LLC sizes, which verifies that the access patterns can be accurately predicted. Third, as the LLC size increases, the obtained weighted speedup decreases. That is attributed to two reasons. One is that the increase of LLC results in less memory requests and thus *ROP* has fewer chances to alleviate refresh overheads. The other one is that a larger LLC size can hold more cache lines and results in better baseline performance, and therefore decreases the performance gap between the baseline and *ROP*. Overall, our system has advantages in various LLC sizes.

VI. Related Work

Venkatesan et al. [17] suggest a software solution called RAPID to exploit the different retention times to guide data placement decision at the software layer. The approach takes

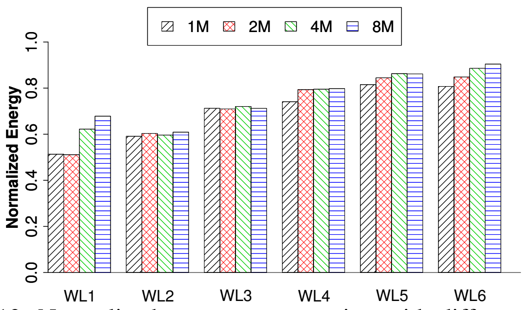


Fig. 13: Normalized energy consumption with different LLC sizes relative to the *Baseline*.

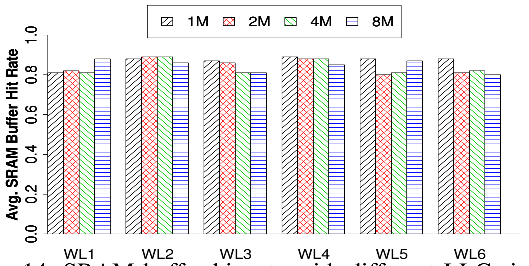


Fig. 14: SRAM buffer hit rate with different LLC sizes.

the differences in retention time into account when allocating DRAM pages. Exploiting the fact that every read or write implicitly performs a refresh, Ghosh et al. [15] propose *Smart Refresh* to reduce refresh energy. In their idea, every row is associated with a counter to remember its remaining retention time and the counter is reset to an initial value whenever it is accessed. Liu et al. [16] suggest another intelligent retention-aware refresh management approach called RAIDR. In RAIDR, DRAM rows are classified into different bins according to their retention times, and row bins are refreshed at differentiated rates. The challenge with retention-aware approaches is how to accurately characterize DRAM retention time, which is hard if not impossible due to *Variable Retention Time (VRT)* [5], [6]. In addition, it requires non-trivial space overhead to record retention time information in high-density DRAM memory. Refresh overheads mitigation could also be realized via optimizing refresh scheduling policy and leveraging internally concurrent architectures. Elastic Refresh [8] modifies the traditional work-conserving scheduling policy to postpone up to eight refreshes if there are predicted read requests arriving in the near future. More recently, Liu et al. [28] leverages the data compression technique to compress cache lines so to avoid refreshing invalid memory space resulting from compression. Mukundan et al. [7] evaluate the efficacy of the fine-grained refresh (FGR) technique defined in the new JEDEC DDR4 standard [21]. Their evaluations have shown that there is no one-size-fits-all solution. Therefore, they suggest an *"Adaptive Refresh"* scheme to dynamically employ different FGR refresh modes according to application characteristics and their phase behaviors. Refresh Pausing [9] is a refresh scheduling mechanism aiming to alleviate refresh overheads. The central idea is to allow a refresh to be interruptible, i.e., it can be paused to service pending requests and resumed after finishing servicing requests. Zhang et al. [20] and Chang [19] propose concurrent refresh-aware memory systems. The main idea of concurrent architectures is to allow requests and refresh to proceed simultaneously in separate and independent modules. Essentially, our proposed scheme *ROP* also benefits from parallelized accesses and refresh, which is however enabled by prefetching. *ROP* distinguishes from those concurrent architectures in the following respects. First, it does not rely on the existence of internally concurrent architectural organizations. Second, no matter how fine-grained the independent granularity is, it is hard for concurrent refresh-aware approaches to avoid access and refresh conflicts [20]. By contrast, the prefetching employed by *ROP* provides possible ways to eliminate refresh blocking effects to a minimal level.

VII. Conclusion and Future Work

DRAM refresh penalty is becoming more and more critical as memory density increases. In this paper, we propose a new refresh mitigation approach named *ROP* which effectively alleviates refresh overheads. ROP enables concurrent access and refresh by deploying a small SRAM buffer in the memory controller to prefetch cache lines before a refresh starts. The prefether dynamically predicts the likely-accessed cache lines in the to-be-refreshed rank and prefetches them into the buffer so that the cache lines are available even during rank frozen cycles. Our extensive evaluations have demonstrated that *ROP* can successfully alleviate refresh overheads.

We plan to extend this work in two directions. First, we intend to implement our idea in DRAM systems which perform refreshes in finer-granularities, e.g., subarray, banks, etc., as these new memory models appear to become popular. We anticipate similar efficacy in those memory systems as well, since using fine-grained refresh granularities cannot completely avoid access conflicts. Second, we want to explore other more informed prefetching algorithms to further alleviate refresh overheads. For example, leveraging processor-side information [29] could provide more accurate information regarding cache lines which are more suitable for prefetching.

Acknowledgement

We are grateful to the anonymous reviewers for their constructive feedbacks. This work is supported by the the U.S. National Science Foundation (NSF) under Grant Nos. CCF-1547804, CNS-1218960, and CNS-1320349, and the Natural Science Foundation of China (NSFC) under Grant 61502189 and 61232004, the Education Ministry of Hubei Province of China under No. B2015054.

References

[1] S. M. Rumble, A. Kejriwa, and J. Ousterhout, "Log-structured Memory for DRAM-based Storage," in Proceedings of FAST'14, 2014.

[2] S. O, Y. H. Son, N. S. Kim, and J. H. Ahn, "Row-Buffer Decoupling: A Case for Low-Latency DRAM Microarchitecture," in Proceedings of ISCA'14, 2014.

[3] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in Proceedings of HPCA'13, 2013.

[4] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-Pages: Increasing DRAM Efficiency with Locality-Aware Data Placement," in Proceedings of ASPLOS'10, 2010.

[5] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in Proceedings of ISCA'13, 2013.

[6] 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 Proceedings of ACM SIGMETRICS'14, 2014.

[7] J. Mukundan, H. Hunter, K. hyoun Kim, J. Stuecheli, and J. F. Mart´ınez, "Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM Systems," in Proceedings of ISCA'13, 2013.

[8] J. Stuechelix, D. Kaseridis, H. C. Hunter, and L. K. John, "Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory," in Proceedings of MICRO'10, 2010.

[9] P. Nair, C.-C. Chou, and M. K. Qureshi, "A Case for Refresh Pausing in DRAM Memory Systems," in Proceedings of HPCA'13, 2013.

[10] M. K. Qureshi, D. H. Kim, S. Khan, P. Nair, and O. Mutlu, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in Proceedings of DSN'15, 2015.

[11] I. Bhati, Z. Chishti, S.-L. Lu, and B. Jacob, "Flexible Auto-Refresh: Enabling Scalable and Energy-Efficient DRAM Refresh Reductions," in Proceedings of ISCA'15, 2015.

[12] Y. H. Son, S. O, Y. Ro, J. W. Lee, and J. H. Ahn, "Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations," in Proceedings of ISCA'13, 2013.

[13] J. Sim, G. H. Loh, V. Sridharan, and M. O'Connor, "Resilient Die-stacked DRAM Caches," in Proceedings of ISCA'13, 2013.

[14] D. Jevdjic, S. Volos, and B. Falsafi, "Die-Stacked DRAM Caches for Servers Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," in Proceedings of ISCA'13, 2013.

[15] M. Ghosh and H.-H. S. Lee, "Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs," in Proceedings of MICRO'07, 2007.

[16] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in Proceedings of ISCA'12, 2012.

[17] R. K. Venkatesan, S. Herr, and E. Rotenberg, "Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM," in Proceedings of HPCA'06, 2006.

[18] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM Refresh-power through Critical Data Partitioning," in Proceedings of ASPLOS'11, 2011.

[19] K. Chang, D. Lee, Z. Chishti, A. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in Proceedings of HPCA'14, 2014.

[20] T. Zhang, M. Poremba, C. Xu, G. Sun, and Y. Xie, "CREAM: a Concurrent-Refresh-Aware DRAM Memory Architecture," in Proceedings of HPCA'14, 2014.

[21] "DDR4 SDRAM STANDARD," http://www.jedec.org/standards-documents/results/jesd79-4%20ddr4, September 2012.

[22] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, "Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems," in Proceedings of HPCA'12, 2012.

[23] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. Pugsley, and Z. Chishti, "Efficiently Prefetching Complex Address Patterns," in Proceedings of MICRO'15, 2015.

[24] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," IEEE Computer Architecture Letters, vol. 10, no. 1, pp. 16–19, January 2011.

[25] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in Proceedings of ISCA'13, 2013.

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in Proceedings of PLDI'05, 2005.

[27] "Micron System Power Calculator," http://www.micron.com/support/power-calc.

[28] W. Liu, P. Huang, K. Tang, K. Zhou, and X. He., "CAR: A Compression-Aware Refresh Approach to Improve Memory Performance and Energy Efficiency," in Proceedings of the poster paper of the 2016 joint ACM SIGMETRICS/IFIP Performance conference (Sigmetrics'2016), 2016.

[29] S. Ghose, H. Lee, and J. F. Martínez, "Improving Memory Scheduling via Processor-Side Load Criticality Information," in Proceedings of ISCA'13, 2013.