EFGR: An Enhanced Fine Granularity Refresh Feature for High-Performance DDR4 DRAM Devices

VENKATA KALYAN TAVVA, RAVI KASHA, and MADHU MUTYAM,
Indian Institute of Technology - Madras

High-density DRAM devices spend significant time refreshing the DRAM cells, leading to performance drop. The JEDEC DDR4 standard provides a Fine Granularity Refresh (FGR) feature to tackle refresh. Motivated by the observation that in FGR mode, only a few banks are involved, we propose an Enhanced FGR (EFGR) feature that introduces three optimizations to the basic FGR feature and exposes the bank-level parallelism within the rank even during the refresh. The first optimization decouples the nonrefreshing banks. The second and third optimizations determine the maximum number of nonrefreshing banks that can be active during refresh and selectively precharge the banks before refresh, respectively. Our simulation results show that the EFGR feature is able to recover almost 56.6% of the performance loss incurred due to refresh operations.

Categories and Subject Descriptors: B.3.1 [**Semiconductor Memories**]: Dynamic Memory (DRAM); B.3.2 [**Semiconductor Memories**]: Design Styles—*Primary memory*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: DRAM architecture, refresh, DDR4, Fine-Granularity Refresh, activation energy, precharge, selective precharge, partial rank refresh

ACM Reference Format:

Venkata Kalyan Tavva, Ravi Kasha, and Madhu Mutyam. 2014. EFGR: An enhanced fine granularity refresh feature for high-performance DDR4 DRAM devices. *ACM Trans. Architect. Code Optim.* 11, 3, Article 31 (October 2014), 26 pages.
DOI: <http://dx.doi.org/10.1145/2656340>

1. INTRODUCTION

A dynamic random access memory (DRAM) cell holds data as charge in a capacitor, and this capacitor has the tendency to leak the charge gradually over time. Retention time of a DRAM cell is defined as the time the cell can retain enough charge so that a read operation can correctly detect the data stored in it. To maintain data integrity, DRAM devices periodically refresh the storage cells conforming to this retention time.

The retention time of a DRAM device is conservatively decided considering the retention time of its weakest cell. The standard retention time specified for commodity DDR-style DRAMs is 64ms under normal temperature ($<85^{\circ}\text{C}$). When the ambient temperature is more than 85°C , due to the increased leakage in the DRAM cells, the retention time is reduced to 32ms. With technology scaling, because of shrinking cell

The work is partly supported by the project grant (No. SB/S3/EECE/0160/2013) from the Science and Engineering Research Board, Department of Science and Technology (DST), India.

Authors' addresses: V. Kalyan T., R. Kasha, and M. Mutyam, Programming Languages, Architecture and Compilers Education (PACE) Laboratory, Department of Computer Science and Technology, Indian Institute of Technology, Madras, Chennai - 600 036, Tamilnadu, India; email: {kalyantv, kasha, madhu}@cse.iitm.ac.in. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2014 ACM 1544-3566/2014/10-ART31 \$15.00

DOI: <http://dx.doi.org/10.1145/2656340>

Table I. Refresh Cycles Times for Different DDR4 DRAM Devices

Device Density	$t_{RFC1x}(ns)$	$t_{RFC2x}(ns)$	$t_{RFC4x}(ns)$
4GB	260	160	110
8GB	350	260	160
16GB	480	350	260
t_{REFI}	$7.8\mu s$	$3.9\mu s$	$1.95\mu s$

sizes and increased influence of process variation, achieving this retention time is extremely difficult [Liu et al. 2013]. Studies [Restle et al. 1992; Lee and Park 2010] have also identified a dynamic nature in a cell's behavior, namely, variable retention time (VRT) and data pattern dependency (DPD). VRT, first identified by Yaney et. al [1987] and studied thoroughly by Restle et. al [1992], is a phenomenon wherein a DRAM cell's retention time varies from time to time. On the other hand, the DPD [Lee and Park 2010] of retention time states that the DRAM cell's retention time is heavily dependent on the data stored in the adjacent cells. Recent studies [Liu et al. 2013; Khan et al. 2014] reaffirm the existence of these dynamic cell behaviors. In Liu et al. [2013], the effects of VRT and DPD are shown to be increasing in modern DRAM devices. Further, considering VRT and DPD, Khan et al. [2014] points to the limitations of existing error (caused by DRAM cell retention time failures) correction and mitigation techniques. These phenomena are observed to be aggravated with technology scaling [Liu et al. 2013]. It is clear that with technology scaling, the retention time is increasingly influenced by process variations, VRT, and DPD. This in turn leads to an increased refresh rate of the DRAM devices to ensure data integrity.

When a DRAM device is being refreshed, it is unavailable for the service of memory requests. In the past, because of small DRAM sizes, the time spent to refresh these cells was small, if not significant. Owing to the integration of multiple cores on a single-processor chip, the demand for large main memory has increased. To meet the demand for increased memory capacity, with the help of advances in fabrication technology, high-density commodity DRAM devices are realized. JEDEC has announced the DDR4 standard [DDR4 2012] with device densities as high as 32Gb. For such high-density devices, the time spent in refreshing the device is significant.

In order to distribute the refresh time uniformly, the memory controller issues 8K refresh commands to the memory, one for every interval, denoted as t_{REFI} . So, at high temperatures, the refresh interval is half that of normal temperatures. For each refresh sent, the time spent in refreshing is denoted as t_{RFC} . With increases in device size, the number of bits refreshed for each refresh command is increasing, leading to an increase in t_{RFC} . Table I provides the values of t_{RFC} for the latest high-density DDR4 devices. Traditionally, the refresh command has been viewed at rank granularity. One major downside of the refresh is that during a refresh operation, an entire DRAM rank is not available for the regular memory requests, which has a significant impact on the overall performance. For example, a rank with 16Gb DRAM devices remains inaccessible for almost 12.3% of the time (refer to Table I), delaying the memory requests by orders of magnitude. An increase in the refresh time along with an increased refresh rate has an adverse impact on both performance and energy, and hence, refresh operations can no longer be overlooked.

Addressing the increased impact of refresh on performance, JEDEC has proposed various refresh modes as part of the DDR4 standard. Fine Granularity Refresh (FGR) is one of them, providing an option to vary both t_{REFI} and t_{RFC} of the device. Motivated by an alternative implementation of the FGR feature, wherein during a finer granularity refresh only a few banks of a rank are involved, in this work, we propose an Enhanced FGR (EFGR) feature.

The EFGR feature tries to expose the banks that are not involved in the refresh (non-refreshing) to the memory controller so that they can be used to service the requests. That is, with the EFGR feature, a refresh command can be viewed as an operation blocking only a few banks in a rank rather than an entire rank. In a way, EFGR increases the availability of DRAM banks to the memory controller. We propose the EFGR feature as three optimizations to the FGR feature.

To follow are the contributions made by this article:

- Optimization 1: Motivated by an implementation of the FGR feature in which only a few banks of the rank are involved in refresh, we first propose modifications to the peripheral circuitry of a DDR4 DRAM device so as to expose the nonrefreshing banks to the memory controller.
- Optimization 2: Given that a refresh operation is power hungry, any refresh design needs to address the device's peak power consumption during refresh. We intuitively show that because of the new FGR feature, for finer granularity refreshes, the peak power consumed is much less than the basic 1x mode. This allows us to actually access a few nonrefreshing banks to service memory requests in parallel with an ongoing refresh, without violating the peak power constraint.
- Optimization 3: We proceed further and question the conventional pre-refresh condition that all the banks in the rank have to be in precharged state to execute a refresh command. In addition, we show that this condition can be relaxed to the set of those few banks to be involved in the refresh.
- Our experimental results show that the proposed EFGR feature achieves almost a 6.8% improvement over the baseline distributed refresh. We conduct extensive sensitivity analyses to understand the impact of optimizations 2 and 3 on performance.
- We also show that EFGR performs better than one of the state-of-the-art refresh handling techniques. On top of that, EFGR, being orthogonal to the existing techniques, is shown to be complementary to them.

The remainder of the article is organized as follows: Section 2 provides details of the refresh operation in high-density DRAM devices. In Section 3, we explain the FGR feature and provide its implementation details. We then explain how our proposed EFGR feature has three optimizations over the FGR feature in Section 4. Section 5 provides details of our experimental setup. Results and analysis are provided in Section 6. We briefly discuss other refresh handling works in Section 7 and finally conclude in Section 8.

2. REFRESH OPERATION IN HIGH-DENSITY DRAM DEVICES

A refresh operation can basically be viewed as an activation of a predefined number of DRAM memory cells (say, a page), followed by their precharge. Data in the cells are restored as part of activation of the cells. The number of pages (N) in a device that are to be refreshed for every refresh command can be determined using the device density (D), page size (P), and refresh interval t_{REFI} , as $N = (D/P) \times (t_{REFI}/64ms)$. In high-density DRAM devices, the value of N is increasing, hence the increase in t_{RFC} .

Exact implementation of a refresh command is highly manufacturer dependent. Given that the amount of sustainable current by the power circuitry of the device is limited, activating a large number of pages simultaneously is not possible. Also, the peak noise current increases with an increase in the number of cells refreshed in a single cycle. Keeping in view these issues, in general, the internal refresh is divided and performed as a sequence of subrefresh operations [Kilmer et al. 2012].

During each subrefresh operation, the refresh controller can engage all the banks or only a few of them. Given that the activation and precharge of the pages during a refresh are done by the refresh controller, parallelism within a bank (subarrays) can

be used. Since these parallel operations share some of the circuits within the bank, the total current drawn need not scale linearly with the amount of parallelism used. To limit the peak noise, two subrefresh operations are not issued back to back but are staggered with some time delay (t_d). So the total time needed per refresh command, t_{RFC} , can be given as $(num_of_subrefreshes - 1) \times t_d + time_per_subrefresh$. Considering these design choices, we can see that the notion of t_{RFC} being directly proportional to the number of pages to refresh is no longer valid [Itoh 2001; Keeth et al. 2007].

With the FGR feature in DDR4 (explained in detail in Section 3), the memory controller is provided with some predefined set of choices that vary the number of pages to be refreshed (N) per refresh, which in turn determines the *num_of_subrefreshes* per refresh command. Because of the staggering of the refresh operations, the time taken by each mode of refresh (t_{RFC1x} , t_{RFC2x} , and t_{RFC4x}) does not scale linearly with N.

Since it is widely employed, we consider the refresh operation with staggered deployment of subrefreshes. Furthermore, within each subrefresh, only a few banks are engaged. Within each bank, subarray-level parallelism is exploited to reduce the effective refresh time per row [Venkata Kalyan et al. 2014].

3. FINE GRANULARITY REFRESH FEATURE IN DDR4

To handle refresh operations, apart from the Auto-Refresh and Self-Refresh modes (continued from the previous DDR generations), the JEDEC standard equips DDR4 with two new features, namely, Temperature Controlled Refresh (TCR) and Fine Granularity Refresh (FGR) features. The TCR feature provides an option to select the refresh interval based on the ambient temperature. The FGR feature, on the other hand, provides an opportunity to vary both the refresh cycle time and the refresh interval. In this work, we are interested in the FGR feature and its implementation, and henceforth do not consider other refresh modes.

FGR 1x mode is equivalent to the conventional distributed refresh provided in previous DDRx devices, wherein within the retention time (64ms), the memory controller needs to send 8K refresh commands, one for every t_{REFI} interval. On the other hand, FGR 2x and 4x modes require refresh commands to be sent at an increased rate of two and four times, respectively. So, the memory controller issues 16K and 32K refresh commands in 2x and 4x modes, respectively. Each command in FGR 2x (4x) mode refreshes half (one-fourth) of the number of bits refreshed during one 1x mode refresh operation. Accordingly, the refresh cycle time (t_{RFC}) differs for each of these modes and is represented as t_{RFC1x} , t_{RFC2x} , and t_{RFC4x} for 1x, 2x, and 4x modes, respectively. For varying device densities, the values of t_{REFI} and t_{RFC} for each of these modes are provided in Table I.

The refresh operation in a DRAM device is nothing but activation and precharge of some predefined number of rows. The number of rows refreshed during a refresh operation depends on the density of the DRAM device, row size, and rate at which refresh commands are sent (FGR mode). For example, for a 16Gb x16 device with eight banks and row size of 2KB, 128 rows are refreshed in 1x mode. Similarly, in 2x and 4x modes, for each refresh command issued, 64 and 32 rows are refreshed, respectively. Even though the number of rows to be refreshed is specified, the distribution of these rows across the device is left to the manufacturer's choice. With the FGR mode, a decision can be taken to involve either *all* the banks or only a *few* banks during the refresh.

Considering the previous example of a 16Gb x16 device, during a refresh in 1x mode, 16 rows in each bank are refreshed. Now, during FGR 2x mode, either eight rows in each of the eight banks can be refreshed or 16 rows can be refreshed from half of the banks. In the same way, an FGR 4x mode can involve all the eight banks and refresh four rows in each of them, or it can involve only two banks and refresh 16 rows in each of them. The implementation of FGR where all the banks are involved in every mode has an

Fig. 1. Block diagram of conventional 16Gb x16 DRAM device.

added advantage that the refresh noise is distributed across the entire device. Though the second method of implementation does not have this advantage, we can still select the banks that are physically far. In this article, we show that with the increasing influence of refresh on performance, it is worth opting for. FGR implementation of 2x and 4x modes wherein only fewer banks are involved in the refresh operation opens up the possibility for the remaining (nonrefreshing) banks to be available for the memory requests. In this work, we investigate the feasibility of this design and study its impact on performance. In the following subsection, we present the implementation details of FGR and provide an insight on exposing the nonrefreshing banks to the memory controller.

3.1. FGR Implementation

Figure 1 shows the block diagram of the peripheral circuitry of a 16Gb x16 DRAM device, containing the refresh controller (the set of blocks within the dotted area) along with the *external bank address decode* block, bank select, and row address muxes. The *timing and mode controller* block distinguishes the normal and refresh operations in the device upon receiving an REF command from the memory controller. This block generates the InternalREF signal, which enables the *refresh mode decode and bank select* block. InternalREF also acts as a select line for the bank select (BSmux0 - BSmux7) and row address (RAMux) muxes. The bank select signals, one per bank, enable the *row address latch and decode* (RAL&D) block of their corresponding banks. The RAL&D block latches the address supplied by the RAMux and decodes it, enabling the access to that bank.

Upon receiving an REF command, the *timing and mode controller* block maintains the InternalREF signal high for a period of t_{RFC} (depending on the FGR mode selected). So, during the entire refresh period, the bank select and row address muxes will select the refresh bank select and refresh row address, respectively. The address of the row to be refreshed in the current refresh operation is supplied by the internal address counter (IAC), maintained by the refresh controller.

The FGR feature in DDR4 DRAM devices allows the memory controller to choose between 1x, 2x, and 4x refresh modes. Also, on-the-fly 1x-to-2x and 1x-to-4x mode shifts are possible. The desired mode is specified by the memory controller using address lines $A_8A_7A_6$ and stored in mode register 3 (MR3) on the device. Table II shows the MR3 definition with the entries concerning the FGR mode. The *refresh mode decode and bank select* block, enabled by the InternalREF signal, decodes the refresh mode (from

Table II. Mode Register 3 (MR3) Definition for FGR Mode

A ₈	A ₇	A ₆	Fine Granularity Refresh mode
0	0	0	Normal mode (Fixed 1x)
0	0	1	Fixed 2x
0	1	0	Fixed 4x
1	0	1	Enable on the fly 2x
1	1	0	Enable on the fly 4x

MR3) and generates the required refresh mode bank select signals (RMBS0–RMBS7), one per each bank. We next explain the working of the *refresh mode decode and bank select* block; the actual implementation of the block can vary with the manufacturer. US Patent 2012/0151131 [Kilmer et al. 2012] presents one possible implementation.

When enabled, the *refresh mode decode and bank select* block selectively raises signals RMBS0–RMBS7. The selection is based on the FGR mode. In 1x mode, all the signals are raised. In 2x mode, signals of four banks are raised, and similarly, in 4x mode, bank select signals of only two banks are raised. The decision on which banks to enable during a 2x or 4x mode refresh is more of a design choice. Without loss of generality, we assume that during each of the refreshes in 2x mode, banks with either even indices (Bank0, Bank2, Bank4, and Bank6) or odd indices (Bank1, Bank3, Bank5, and Bank7) will be selected for refresh. Similarly, we assume that in 4x mode, during each refresh, any one of Bank0–Bank4, Bank1–Bank5, Bank2–Bank6, or Bank3–Bank7 will be chosen. Now, when in 2x mode, the *refresh mode decode and bank select* block will choose either of the bank sets based on the number of REF commands received so far. A 1-bit flag can be maintained to select either of the sets, and the flag is set/reset every time an REF command is received. Likewise, in 4x mode, a 2-bit flag can be used to select the set of banks for refresh.

In 4x refresh mode, among the eight banks in the device, two banks will undergo refresh for each REF command from the memory controller. Let us assume that during the current refresh, Bank1 and Bank5 are selected for refresh. So, the *refresh mode and bank select* block will raise RMBS1 and RMBS5 high and maintain all the other signals low. Following this, the bank select muxes BSmux1 and BSmux5 will select RMBS1 and RMBS5, respectively, and raise the final bank select signals BS1 and BS5. One major disadvantage of this design is that all the other nonrefreshing banks will not be accessible (because their RAL&D blocks are disabled) and remain idle during the entire refresh period. In our proposed enhancement to FGR, we address this aspect of underutilization of banks during FGR 2x/4x mode operations and modify the peripheral circuitry of the DRAM device so as to expose the idle banks to the memory controller. With this modification, the memory controller can get requests serviced by the nonrefreshing banks and lessen the impact of the ongoing refresh operation on performance.

4. ENHANCED FINE GRANULARITY REFRESH (EFGR)

In this section, we present the details of our enhancement to the FGR feature. Our primary motivation for EFGR is the alternative implementation of the basic FGR feature, wherein only a few banks are involved during the finer (2x/4x) granularity refresh. When all the banks of a rank are involved in refresh, we observe that the basic FGR 2x and 4x modes are not effective for the workloads considered (complete details of our experimental setup are provided in Section 5). Figure 2 shows the potential speedup (over FGR 1x) achievable if the memory needs no refresh at all (*No Refresh*). The figure also shows the speedup obtained by FGR 2x and 4x modes when compared to the 1x mode. It can be clearly seen that both 2x and 4x modes degrade performance significantly. Only *MT-c* benchmark experiences a 1% improvement in performance.

Fig. 2. Performance of FGR 2x, FGR 4x, and No Refresh over FGR 1x mode.

Fig. 3. Average number of requests waiting for refreshing and nonrefreshing banks during refresh in FGR 2x and FGR 4x modes.

Clearly, for the workloads chosen, FGR 2x and 4x modes do not provide any extra advantage when the entire rank is involved in refresh.

The number of memory requests delayed because of an ongoing refresh operation provides us with the second motivation for EFGR. Figure 3 presents the average number of requests waiting for service from refreshing and nonrefreshing banks during an ongoing refresh of a rank. In FGR 2x mode, an equal number of banks are in refreshing and nonrefreshing modes; hence, the average number of requests waiting is almost equal in both sets of banks. It can be observed that on average, more than five and six requests are waiting for the nonrefreshing banks in FGR 2x and 4x modes, respectively. Given that during a fine granularity refresh there are a significant number of requests waiting for the nonrefreshing banks, which in the basic FGR remain idle and underutilized, we intend to expose them to the memory controller. It can be noted that in the process of exposing the nonrefreshing banks to the memory controller, we are actually overruling the traditional conception that an entire rank is unavailable during a refresh operation. With EFGR, only those requests going to the refreshing banks experience delayed service, whereas the requests to the nonrefreshing banks can proceed unaffected. In a way, EFGR by revealing the nonrefreshing banks to the memory controller achieves the concurrent operation of both refresh and memory request service.

The following three subsections provide the implementation details of EFGR as three optimizations to the conventional FGR. As the first optimization (in Section 4.1), we introduce modifications to the peripheral circuitry of a DRAM device so as to decouple the nonrefreshing banks from the ongoing refresh operation. The second optimization tries to quantify the number of nonrefreshing banks that can be accessed in parallel with the refresh by considering the peak power consumption of the device. Given

Fig. 4. Block diagram of the 16Gb x16 DRAM device with the proposed changes.

that the refresh operation is one of the most power-hungry operations, in Section 4.2 we intuitively show that with fine granularity refreshes, simultaneous accesses to the nonrefreshing banks are possible without actually violating the peak power profile. The final optimization (Section 4.3) questions the traditional pre-refresh condition that *all* the banks need to be in the *precharged* state in order to execute a refresh operation. We show that this pre-refresh condition for FGR is pessimistic and can be relaxed to provide performance benefits.

4.1. Exposing the Nonrefreshing Banks to the Memory Controller

The main drawback of the basic FGR implementation (Figure 1) is that even during a finer granularity refresh operation, wherein only a few banks are involved, the non-refreshing banks cannot be used and remain idle for the period of refresh cycle time. In order to exploit these idle nonrefreshing banks for servicing memory requests, we introduce few minor changes to the peripheral circuitry of the DRAM device.

In Figure 4, the needed modifications are highlighted in the dark shade. The single row address mux $RAMux$ (in Figure 1) is replaced by per bank row address muxes ($RAMux0$ - $RAMux7$), with $RMBS0$ - $RMBS7$ as their corresponding select lines. Also, the select lines of the bank select muxes ($BSmux0$ - $BSmux7$) are modified. For each of these muxes (say, $BSmux0$), the modified logic selects the output ($RMBS0$) from the *refresh mode decode and bank select* block only when $InternalREF$ and $RMBS0$ are both high. During a finer granularity refresh operation, if a bank is not selected for the current refresh, the corresponding external bank select signal $EBS0$ - $EBS7$ will be selected (if needed) and $RAL\&D$ is enabled. With these minor changes, we can decouple the nonrefreshing banks from an ongoing refresh operation.

For example, consider 4x refresh mode. Say that during the current refresh operation Bank2 and Bank6 are undergoing refresh. Now, when a request arrives for a bank (say, Bank7), the external bank select $EBS7$ is held high, and because the refresh mode bank select $RMBS7$ remains low, the row address mux $RAMux7$ selects the external row address supplied by the memory controller. In this way, the memory controller can service requests to the nonrefreshing banks within the rank during a finer granularity refresh. When in 1x mode, the modified design works similarly to the basic FGR, involving all the banks, and hence provides no extra advantage.

Table III. I_{dd} Values of Various Operations for a 16Gb 1,600Mbps DDR4 DRAM

Component	Value (in mA)	Component	Value (in mA)
IDD0	24	IDD1	32
IDD3P	7.2	IDD2P	6.4
IDD2N	10.1	IDD3N	16.6
IDD4R	60	IDD4W	58
IDD5B	102	IDD6	6.7
IDD7	107		

The additional hardware does not lengthen the critical path for two reasons. First, the AND gates that drive the select lines of each of the bank select muxes (BSmux0-BSmux7) remain low during the normal operation of the device as the InternalREF signal is low. So, BSmux0-BSmux7 select the external bank select signals EBS0-EBS7 by default. During the refresh operation, once the RMBS0-RMBS7 and the InternalREF are determined, the corresponding AND gates will continue to remain low or rise high depending on the current refreshing banks, resulting in no extra delay in banks' accesses. Second, the additional row address muxes RAMux0-RAmux7 too do not extend the critical path because the select lines for these muxes, RMBS0-RMBS7, remain low for the normal operation of the device and few of them rise high at the beginning of the refresh operation for the refreshing banks.

Different DRAM manufacturers can opt for different peripheral circuit designs. There might be a case that individual bank select signals are not issued by the *refresh mode decode and bank select* (RMBS0-RMBS7) and the *external bank decode* (EBS0-EBS7) blocks. US Patent 2012/0151131 [Kilmer et al. 2012] presents one such implementation, wherein both the blocks are merged and a single bank select signal is driven out, selecting one bank at a time. Even with such a design, EFGR's additional hardware will be similar to the proposed changes, and all the modifications are limited to the merged (common) bank decode block.

4.2. Scope for Parallel Access to Banks During FGR—Power Perspective

The refresh operation is one of the most power-hungry operations in DRAM devices and poses the main challenge in the design of EFGR. To support our attempt to access the nonrefreshing banks during a finer granularity refresh operation, we need to study the peak power profile sustainable by the DRAM device. Because the exact peak power profiles are usually not disclosed by the manufacturers, accurate peak power analysis is beyond the scope of this article. We use the average power consumed during an operation as an indication for its peak power. DRAM power calculators like Micron's model [Inc. 2007] or the improved model by Chandrasekar et al. [2011] or even the detailed power model by Vogelsang [2010] do not provide any insights regarding peak profiles. The average current drawn can be used as a fair index to understand the peak current (power) drawn during each operation.

Table III provides the average I_{dd} values for various operations of a DRAM device (taken from [Mukundan et al. 2013]). The ones of our interest are operating one bank active-precharge current ($IDD0$), precharge standby current ($IDD2N$), active standby current ($IDD3N$), operating burst read current ($IDD4R$), operating burst write current ($IDD4W$), and burst refresh current in 1x mode ($IDD5B$). It can be clearly seen that refresh current $IDD5B$ that is drawn during a 1x mode refresh is very high. JEDEC [DDR4 2012] specifies the burst refresh current drawn during 2x and 4x modes as $IDD5F2$ and $IDD5F4$, respectively.

It can be observed that during a 4x mode, within the t_{RFC4x} time, one-fourth of the work (refreshing the rows) is done when compared to that of 1x mode in t_{RFC1x} time. For 16Gb devices, t_{RFC1x} and t_{RFC4x} are 480ns and 260ns, respectively (Table I).

Considering these values, the current drawn during a 4x mode can be obtained by equating the product of supply voltage V_{dd} , current drawn $IDD5F4$, and refresh cycle time t_{RFC4x} to one-fourth of the product of V_{dd} , $IDD5B$, and t_{RFC1x} . We obtain $IDD5F4$ as $47mA$.

DRAM devices can actually sustain $IDD5B$ current during a 1x refresh operation, so during a 4x mode refresh, we can see that there is a scope of using the other nonrefreshing banks as long as the total current does not exceed it. So, a read/write operation and an activation and/or precharge operation can be done alongside an ongoing 4x mode refresh. Considering the background current $IDD3N$ along with these operations, we estimate that two nonrefreshing banks can be in active mode during a 4x mode refresh. These nonrefreshing banks are exposed to the memory controller with hardware changes discussed in Section 4.1.

The second optimization in EFGR is actually the constraint on the number of banks that can be accessed simultaneously with an ongoing fine-grained refresh operation. From the previous discussion, we see that two banks can be active and accessed. Note that even though two banks are in the active state, we cannot overlap two read or write operations. Even during the normal operation, two read/write operations can be issued to two banks with a delay equal to the maximum of t_{BURST} and t_{CCD} , to avoid conflict for the common resource (data bus). Similarly, a new timing constraint, $t_{CASToCASdelay_REF}$, applicable only during a refresh, can be introduced to delay a read/write operation further when one read/write operation is already in progress. Since t_{CL} is 10 cycles (Table V), we consider this new delay as 10 cycles to ensure that there is absolutely no overlap of two read/write operations. It implies that any two consecutive CAS commands (targeted to a same bank or different banks) will have to meet the new timing constraint during refresh, enforced by the memory controller.

Two timing parameters, for safety reasons, limit the peak current drawn by a device. Delays—row activate-to-activate (t_{RRD}) and four-bank-activation-window (t_{FAW})—ensure that frequent activations (that draw high current) do not take place. In any t_{FAW} window, not more than four activations should happen, spaced at least t_{RRD} apart. These timing constraints limit the DRAM performance. Given that a refresh can be viewed as a pair of activation and precharge operations, the device has to obey t_{RRD} and t_{FAW} constraints during a refresh too. In EFGR 4x, given that two banks are under refresh, only two other activations are possible in a given t_{FAW} window. It can be noted that during a refresh operation, the activations by the refresh controller are of high priority and are issued the moment their respective timing constraints are met, without further delay.

We hope that when the EFGR feature is endorsed, the manufacturers can specify the maximum number of banks that can be accessed during a refresh and $t_{CASToCASdelay_REF}$, both of which can be determined during the stress tests. Due to the stringent peak power constraint, we do not consider EFGR in 2x mode for the remaining part of the article and restrict EFGR to 4x mode alone.

4.3. Selective Precharges

The third optimization in EFGR is to minimize disruption to the nonrefreshing banks at the beginning of the refresh operation. For a refresh command to be executed, a pre-refresh condition that all the banks of a rank are to be in *precharged* state has to be ensured. This pre-refresh condition is required to ensure data integrity in the active banks that have a page open.

For the conventional refresh, because *all* the banks in a rank are involved in the refresh operation, the memory controller has to check the status of *all* the banks and precharge the active banks. During the time from when the memory controller decides

Fig. 5. Average number of precharges sent by the memory controller in FGR 4x mode to satisfy the pre-refresh condition that *all banks are to be in precharged state*.

to refresh a rank till the actual time when the REF command is sent, the rank remains underutilized (apart from servicing the PRE from the memory controller).

With FGR mode, given that during a finer granularity refresh only a few banks of the rank are actually involved in the refresh, this prerequisite is conservative and needs to be revisited. The pre-refresh condition needs to be met by the refreshing banks alone and not by the other banks because a bank that is not involved in refresh need not be precharged forcibly. This forceful precharge disrupts the service of the requests to that particular bank and can lead to an increase in the request service time and loss in performance.

Figure 5 shows the average number of precharges (PRE) sent by the memory controller to meet the pre-refresh condition during an FGR 4x mode refresh. For each benchmark, the number of precharges sent to both the refreshing and the nonrefreshing banks to satisfy the pre-refresh condition is shown. On average, per refresh, 1.72 precharges are sent, of which 1.29 are intended to forcibly close a nonrefreshing bank. Figure 5 shows that with this conservative approach, for almost 75% of the time both underutilization of the rank and disruption of the activity in the nonrefreshing banks happen.

We name the third optimization in EFGR *selective precharge of banks*. With selective precharge optimization enabled, precharges are sent to meet the prerefresh condition restricted to refreshing banks. Ideally, the prerefresh condition can be modified to ensure that only those banks to be involved in refresh are in the *precharged* state. But because of the constraint imposed by the second optimization, the memory controller can keep only two nonrefreshing banks active even at the beginning of the refresh. So, the memory controller issues precharges not only to those banks that are to be involved in refresh but also to a few other active banks if the number of active banks is more than the limit indicated by the second optimization. For example, for the 16Gb x16 device considered, suppose in 4x mode Bank0 and Bank4 are to be refreshed next and presently Bank0, Bank1, Bank4, Bank5, Bank6, and Bank7 are active. The memory controller will precharge Bank0, Bank4, and any two among Bank1, Bank5, Bank6, and Bank7. Say if Bank5 and Bank6 are chosen for precharge, the other banks (Bank1 and Bank7) continue to be in their respective states.

Given that few of the nonrefreshing banks are already in active mode, closure of a few of them might seem conservative and unnecessary. But because an active bank draws more current when compared to an idle bank, to limit the maximum instantaneous current drawn by the device, some of these nonrefreshing active banks are precharged. If the number of nonrefreshing banks that are active is greater than the limit specified by the second optimization, a decision on choice of banks to precharge has to be taken by the memory controller. One logical choice is to close the banks that have a fewer number

of requests waiting. From our experiments, we observe that even random selection of banks is sufficient and there is only incremental improvement in performance when compared to closure of banks based on the number of requests waiting.

For the memory controller to exploit the EFGR feature, it should have information about the pattern in which banks of a rank are refreshed during a fine granularity refresh and the number of accessible nonrefreshing banks. This information can be stored in the *Serial Presence Detect* (SPD) present in the DRAM module. SPD is read during the BIOS, so this information can be communicated to the memory controller at the system boot time without any extra overhead.

To implement EFGR, a few minor modifications are needed to the memory controller. First, at the memory controller, instead of a single per-rank refresh status, the status of groups of banks has to be maintained. Second, when a rank is under refresh, the additional constraints such as the number of accessible banks and $t_{CAS\text{to}CAS\text{delay_REF}}$ have to be imposed. Last, before the start of a refresh, the memory controller needs to close the open banks selectively.

The three optimizations when included together in the conventional FGR form the proposed EFGR. The first optimization that proposes changes to the hardware is the basic feature in EFGR. In our experiments, we explore the impact of varying the constraint imposed in the second optimization so as to study its potential impact on performance. To exactly quantify the effect of selective precharging, we study performance of EFGR with and without selective precharging. As pointed out by Chandrasekar et al. [2011], using their improved power modeling for DRAM, the precharges sent to meet the conventional pre-refresh condition can contribute to almost 30% of the total refresh power. Hence, with selective precharging in EFGR, the power contributed by these precharges will reduce. Since in this work we target performance, we study the impact of EFGR mostly from the performance point of view.

5. EXPERIMENTAL METHODOLOGY

To evaluate the proposed technique, we use the memory system simulator USIMM [Chatterjee et al. 2012] from the Memory Scheduling Championship (MSC) [MSC 2012]. The USIMM simulator models a DRAM-based memory system in detail, with all the required timing constraints and transaction queues (read and write). We aggressively modify the tool for our study and introduce command queues (CQs) and model different refresh handling techniques. We implement FGR 2x and 4x modes, which can be enabled with any of the other refresh handling techniques.

Based on the vacancy in the CQ, in each DRAM cycle, a request from the read or write queue (when in write mode) is selected (based on age, FCFS). The selected request is split into multiple simple DRAM understandable commands (ACT, CAS, and PRE) and enqueued into the CQ. Also, in every cycle, the CQ is looked up for any command satisfying the timing constraints and the ready command is issued to the DRAM. The CQ structure can be either per channel or per rank or even per bank [Jacob et al. 2007]. We choose the simple per-channel implementation because our experiments with per-rank or per-bank queues did not yield significant improvement over it.

A refresh operation can contend along with the regular read and write requests in every cycle. Our baseline refresh handling technique is *Demand Refresh*, which is equivalent to the traditional distributed refresh provided in the DDRx standard. After every t_{REFI} , the memory controller issues the refresh command at the earliest time possible and blocks an entire rank for a period of t_{RFC} . When in either FGR 2x or 4x mode, corresponding t_{REFI} and t_{RFC} are considered (refer to Table I). We also conduct experiments for a case where memory does not need a refresh (*No Refresh*).

For mobile platforms, JEDEC provides a Low-Power DDR (LPDDR) standard. Starting with the LPDDR2 standard [LPDDR2 2009; LPDDR3 2012], concerning refresh,

Table IV. Base System Configuration

Processor Configuration	
Number of cores	4
Processor frequency	3.2GHz
Pipeline depth	10
Fetch/retire width	4/4
ROBsize	160
Last-level cache (private)	512KB per core
Cache block size	64B
Memory Controller Configuration	
Read transaction queue	128 entries
Write transaction queue	64 entries
Write queue high/low water mark	40/20
Address mapping	Row:Col:Rank:Bank:Ch
Command scheduling policy	FR-FCFS
Row management policy	Opportunistic close-page
Command queue	32 entries
Number of channels	1
DRAM Configuration - DDR4 16Gb x16 devices	
Memory clock frequency	800MHz
Number of ranks	4 per channel
Number of banks	8 per rank
Number of rows	128K rows per bank
Row size	2KB
Burst length	8

JEDEC provided two extreme choices of *All-Bank* refresh and *Per-Bank* refresh. As the names suggest, in *All-Bank* refresh, an entire rank is engaged in the refresh (for t_{RFCab} time). On the other hand, in *Per-Bank* refresh, *only* one bank is refreshed per command (for t_{RFCpb} time). If there are eight banks in the rank, for *Per-Bank* refresh, the memory controller issues an REF command for every $t_{REFI}/8$ cycles. The *No Refresh* case, though, gives an idea on the maximum possible performance improvement, it is impractical; hence, we consider *Per-Bank* refresh in our evaluation and compare EFGR against it. Though the ratio of t_{RFCab} to t_{RFCpb} is around 2.3 in the LPDDR2 standard [LPDDR2 2011], in our experiments we consider the ratio to be 3. Considering Table I, it can be noted that with this ratio, t_{RFCpb} is equal to 160ns (209ns if the ratio is 2.3), suggesting that in our experimental evaluation *Per-Bank* refresh is given extra advantage.

In case JEDEC supports an 8x refresh mode (as suggested by Kilmer et al. [2012]), EFGR 8x can be similar to *Per-Bank* refresh. Unlike *Per-Bank* refresh in the LPDDR standard, an additional advantage with EFGR in the DDR4 standard is the availability of both 4x and 8x modes for the memory controller, wherein the memory controller can choose the refresh mode based on demand on the memory (such as Adaptive Refresh [Mukundan et al. 2013]).

Complete details of the base system configuration are provided in Table IV. We model a quad-core system with a single memory channel. We model the memory subsystem based on the configurations used in the Intel Xeon E7-88xx series [Xeon 2014] and IBM Power7 systems 710 [Sinharoy et al. 2011], wherein the ratio of threads to memory controllers is at least four. By default, read requests are prioritized over write requests. Write requests are scheduled only when the occupancy of the requests in the write queue crosses the high water mark and are drained till the occupancy falls below the low water mark. As mentioned earlier, FR-FCFS [Rixner et al. 2000] is the command

Table V. Timing Parameters for a 16Gb 1,600Mbps DDR4 DRAM Device

Parameter	DRAM Cycles	Parameter	DRAM Cycles
t_{RCD}	10	t_{RAS}	28
$t_{CAS} (t_{CL})$	10	t_{RP}	10
t_{RC}	38	t_{RRD}	6
$t_{CCD.S}$	4	$t_{CCD.L}$	5
t_{WTR}	6	t_{WR}	15
t_{FAW}	28	t_{RTP}	6
t_{RTRS}	2	t_{BURST}	4
$t_{CAS}t_{CAS}delay_REF$	10		

Table VI. MSC Suite Workload Characteristics

Suite	Workload	MPKI	Instructions Per Cycle (IPC)
COMMERCIAL	comm1	6.9	1.67
	comm2	8.4	1.31
	comm3	3.5	2.17
	comm4	2.4	2.54
	comm5	1.5	2.84
PARSEC	black	3.2	2.26
	face	6.1	1.63
	ferret	6.7	1.61
	fluid	3.0	2.29
	freq	3.0	2.4
	stream	3.7	2.17
	swapt	3.5	2.24
	MT-c	14.7	0.91
SPEC2006	MT-f	7.6	1.53
	leslie	6.4	1.16
BIOBENCH	libquantum (lib)	14.0	1.02
	mummer	20.3	0.82
	tigr	27.3	1.02

scheduling policy, and we opt for close-page policy with opportunistic page hits for row buffer management since it is better suited for multiprogrammed workloads. We consider 16Gb 10-10-10 DDR4 DRAM devices operating at 800MHz with t_{RFC} and t_{REFI} chosen according to the FGR mode. Timing parameters of the DRAM device are provided in Table V. For EFGR 4x, we conservatively consider the t_{RFC} value to be same as that of FGR 4x. This is a conservative estimate because the ratio of t_{RFC1x} to t_{RFC4x} according to the work by Kilmer et al. [2012] is around 3.5, whereas the ratio considering the JEDEC standard (Table I) is 1.8.

To evaluate our technique for a wide range of workloads, we use the workload set from MSC [MSC 2012]. The MSC suite contains 18 applications in total, composed of five commercial applications (*comm1-comm5*), nine benchmarks from the PARSEC suite, and two benchmarks each from the SPEC2006 and BIOBENCH suites. Among the nine benchmarks of PARSEC, two include multithreaded versions of applications: *fluid* (*MT-f*) and *cannal* (*MT-c*). Table VI gives the key characteristics of these benchmarks. For each simulation, each of these benchmarks is run in rate mode on the quad-core processor till the end of the slowest one. We report speedup calculated using a commonly used performance metric for multiprogrammed environments, weighted speedup [Snively and Tullsen 2000]. Energy per memory access is reported to help

Fig. 6. Performance improvement obtained by FGR 4x, EFGR 4x, Per-Bank refresh, and No Refresh, normalized over FGR 1x.

Fig. 7. Normalized (over FGR 1x) average read latency for FGR 4x, EFGR 4x, Per-Bank refresh, and No Refresh. The lower the better.

understand the DRAM memory system power, calculated using the improved energy calculation model proposed by Chandrasekar et al. [2011].

6. RESULTS AND ANALYSIS

Figure 6 shows the performance improvement obtained by *Demand Refresh* 4x (FGR 4x), EFGR 4x, *Per-Bank* refresh, and *No Refresh* scenarios, normalized over Demand Refresh in 1x (FGR 1x) mode. The following observations are in order: First, *No Refresh* (impractical) shows that refresh can degrade performance of the considered workloads significantly (12% on an average). Second, the *Per-Bank* refresh when employed achieves up to 10% improvement in performance. Third, only *MT-c* is benefited by 4x fine granularity refresh mode. On average, FGR 4x experiences a performance drop of 5.8%. Henceforth, we do not consider FGR 4x for our study. Finally, the graph shows that EFGR 4x shows significant performance improvement for almost all the benchmarks. It improves performance by 6.8% on average. The benchmarks that have a higher scope of improvement for *No Refresh* and *Per-Bank* refresh are also the ones benefited most by EFGR.

Figure 7 shows the normalized average read latency for each workload for FGR 4x, EFGR 4x, *Per-Bank* refresh, and *No Refresh* cases. It clearly gives an indication that FGR 4x lengthens the average request time when compared to FGR 1x. Also, the figure shows that *Per-Bank* refresh and EFGR significantly reduce the read latency and are comparable to *No Refresh*, hinting that exposing the nonrefreshing banks is actually very effective. The read latency of EFGR 4x is around 10 processor cycles more than that of *Per-Bank* refresh.

Figure 8 helps us to understand the variation in performance improvement obtained across the benchmarks. It presents for each benchmark the average number of read requests per 1K cycles (say, frequency of requests) as the primary y-axis and the average number of requests waiting for service during refresh as the secondary

Fig. 8. Average number of read requests per 1K cycles and average number of requests waiting for service during an ongoing refresh operation, considering FGR 1x.

y-axis. It is intuitive that for a benchmark that has a low frequency of requests and fewer number of requests waiting, the impact of refresh on performance is less. This is indeed the case for *comm5* (3.8%) and *MT-f* (0.8%) benchmarks. On the other hand, for benchmarks like *comm1*, *lib*, *mummer*, and *tigr*, which have both a high frequency of requests (>25) and a high number of requests waiting (>10), *No Refresh* improves performance by more than 12%. The improvement in performance by EFGR 4x too varies similarly (except for *lib*, explained next).

Benchmark *lib* is the only benchmark that does not experience significant improvement in performance with EFGR 4x (even when *Per-Bank* refresh is considered, the performance gain achieved (7.3%) is much less than the overall gain). The reason for this behavior is found from our observation that DRAM address mapping has a significant role in *lib* accesses. Because the chosen address mapping (Row:Col:Rank:Bank:Ch, say AM1) exploits the available parallelism in DRAM to the maximum by distributing the requests across all the banks and ranks, servicing the requests of the nonrefreshing banks and delaying (not attending) the requests of the refreshing banks are not actually improving performance for *lib*. This is because for *lib*, the processor is stalled on the requests to the refreshing banks. In order to prove this observation, we opt for an address mapping (Row:Col:Bank:Rank:Ch, say AM2) that skews the requests to a fewer number of banks at any given moment of time. We find that when AM2 is employed, for *lib*, EFGR 4x achieves 11.2% improvement in performance, hinting that when the requests target a fewer number of banks (i.e., exhibit low bank-level parallelism), freeing up those banks is very effective. Similarly, for *Per-Bank* refresh, with AM2, performance gain is around 11.6%. We do not adopt AM2 for our study because when FGR 1x is considered, it loses almost 9.3% (for *lib*) when compared to FGR 1x with AM1.

6.1. Impact of Number of Accessible Nonrefreshing Banks

To understand the impact of the constraints imposed in the second optimization of EFGR, we conduct experiments wherein the number of nonrefreshing banks that can be accessed in parallel with an ongoing refresh is varied. We limit the number of accessible nonrefreshing banks to one and two and observe the performance. Also, in order to understand the theoretical limit of performance improvement with EFGR, we run the EFGR 4x-unconstrained configuration, wherein there is no restriction on the maximum number of nonrefreshing banks accessed by the memory controller. This configuration actually gives us an idea of the ability of the workloads to exploit the nonrefreshing banks. Figure 9 shows performance improvement obtained by varying the number of accessible nonrefreshing banks over FGR 1x mode.

Even when only one nonrefreshing bank is made available to the memory controller during refresh, EFGR achieves 4.6% improvement in performance, on average. As

Fig. 9. Performance improvement obtained from EFGR when the maximum active banks constraint is varied.

previously observed, the improvement increases to 6.7% with two banks. Benchmark *stream* experiences a drop in performance gain for the EFGR 4x-unconstrained configuration from the EFGR 4x-two banks active configuration. To understand this behavior, on closer look, we observe that for this benchmark, the number of rank-to-rank switches increases significantly for the unconstrained configuration. Coupled with low bank-level parallelism, for *stream*, the effective improvement in performance lessens. On the other hand, for address mapping AM2, we observe no anomalous behavior, and the gain in performance increases with an increase in the number of accessible nonrefreshing banks.

EFGR without any constraints can achieve up to 7.8% improvement. The gain in performance of EFGR when the number of available banks is increased from two to unconstrained is less than the improvement obtained when banks are increased from one to two because of the increased delay between any two CAS commands ($t_{CAS to CAS delay_REF}$) during refresh. When compared to *No Refresh*, the remaining 4.2% performance loss can be because of nonavailability of the two banks under refresh and/or because of the serialized service of the waiting requests due to low bank-level parallelism. Even with the peak power constraint, EFGR is able to recover almost 56% of loss incurred by refresh operations. On a realistic note, when compared to the *Per-Bank* refresh, EFGR-unconstrained loses only 2.2% in performance. This makes a strong case to include EFGR as a feature in future DDR4 devices.

6.2. Impact of Selective Precharging

To quantify the effect of selective precharging (optimization 3) of EFGR, we study performance when it is disabled and compare it with performance obtained by enabling it. Figure 10 shows the improvement in performance with and without selective precharging in EFGR. Performance of EFGR without selective precharging drops to 5.9% from 6.7%.

Figure 11 presents the percentage of time at least one PRE command was issued by the memory controller before issuing an REF command. It is an indicator of the underutilization of rank to satisfy the pre-refresh condition. EFGR 4x without selective precharging, for 69% of the time, waits to precharge all the banks before refresh. This is almost equal to FGR 1x mode (72.7%). The percentage drops significantly to 39% for EFGR 4x with selective precharging, hinting at the proportional decrease in pre-refresh wait time. Performance improvement is also because of the reduced disruption to the nonrefreshing banks by avoiding unnecessary precharges. Though the improvement in performance is not significant when precharges are issued selectively, there is a potential decrease in the average refresh power [Chandrasekar et al. 2011].

Fig. 10. Performance improvement obtained from EFGR with and without application of selective precharges during the pre-refresh period.

Fig. 11. Percentage of the time the memory controller has issued at least one PRE to satisfy the pre-refresh condition, considering FGR 1x, EFGR 4x-without selective precharging, and EFGR 4x. The lower the better.

Fig. 12. Energy consumed per memory request serviced.

6.3. Note on Energy Consumption

Though EFGR mainly targets performance, because of reduction in the overall static energy, there are some energy savings when compared to FGR 1x mode. Figure 12 presents the energy-per-access values for FGR 1x, EFGR, *Per-Bank*, and *No Refresh* techniques. On average, there is a reduction of $1.1nJ$ of energy when compared to FGR 1x mode.

6.4. Effectiveness with Multiprogrammed Workloads

For our basic study, rate mode was chosen to understand the behavior of individual benchmarks for different refresh techniques. We now present results obtained by considering various multiprogrammed workloads. We consider 16 applications (excluding MT-c and MT-f) and form 22 mixes. Figure 13 presents the improvement in performance over FGR 1x mode for EFGR, *Per-Bank* refresh, and *No Refresh* cases. *mix9*, *mix12*,

Fig. 13. Performance improvement obtained by EFGR, Per-Bank, and No Refresh over FGR 1x mode for various multiprogrammed workloads.

Fig. 14. Performance improvement obtained from EFGR with only one bank and two banks active, and No Refresh over FGR 1x mode in extended temperature mode.

and *mix17* experience less than 5% improvement because *lib* is one of the benchmarks in each of these three workloads and it has very little improvement with EFGR (as explained in Figure 6). On average, EFGR achieves a 6% performance improvement over FGR x1. *Per-Bank* and *No Refresh* cases show 10.1% and 13.1% improvement, respectively. Clearly, EFGR is as effective for multiprogrammed workloads as for rate mode workloads. Hence, we continue our further study with rate mode alone.

6.5. Effectiveness at High Temperatures

At extended temperatures (85°C–95°C), the refresh rate is doubled to tackle the increased DRAM cell leakage and ensure data integrity. Figure 14 shows the performance improvement obtained at the extended temperature range. When we consider the EFGR feature in the extended temperature range, it is able to achieve almost a 19.8% performance improvement over FGR 1x. Around 29.3% improvement is theoretically achievable (*No Refresh*). On a realistic front, *Per-Bank* refresh achieves up to 25.1% improvement. This shows that exposing the nonrefreshing banks during the refresh is indeed very effective in hiding the refresh penalty. We observe that even when only one nonrefreshing bank is allowed to be accessed, the EFGR feature is able to achieve a 17% improvement in performance. This hints that even with a conservative approach in optimization 2, EFGR still has good potential to be adopted in future systems.

6.6. Effectiveness with Memory Density

To understand the effectiveness of EFGR for different DRAM device densities, we consider devices of 8Gb, 16Gb, and 32Gb capacity. We optimistically predict that t_{RFC} for 32Gb would be around 800ns. Figure 15 presents the average performance achieved by EFGR 4x, *Per-Bank*, and *No Refresh* cases. We can clearly see that the effectiveness of EFGR in reducing the impact of the refresh operation on performance increases with device density. This is because, for higher-density devices, the availability of the

Fig. 15. Average performance improvement achieved by EFGR, Per-Bank, and No Refresh schemes over FGR 1x mode for various DRAM device densities.

rank for memory requests further decreases, and hence, making nonrefreshing banks available to the memory controller would be much more beneficial.

6.7. Comparison with Adaptive Refresh, Preemptive Command Drain, and Delayed Command Expansion Techniques

Mukundan et al. [2013] recently proposed three mechanisms to reduce refresh penalty on performance. The first mechanism proposed was *Adaptive Refresh* (AR), which uses an on-the-fly 1x-to-2x (1x-to-4x) mode shift feature in FGR to dynamically choose between 1x and 2x (4x) modes. Tracking the data bus usage in each epoch, AR chooses the best mode for each application and for each phase within the application. During the training period, for two consecutive intervals of size n each, the data bus utilization in 1x and 2x (or 4x) modes is observed. A mode is chosen if it has more data bus utilization over the other. During the testing period that follows this training period, for m intervals ($m \gg n$), the chosen mode is employed. The process repeats at the end of every testing period.

The other two mechanisms proposed in Mukundan et al. [2013] address the increased contention of the common command queue among ranks during the refresh of one of the ranks. The phenomenon of increased contention is termed as *command queue seizure*. Preemptive Command Drain (PCD) and Delayed Command Expansion (DCE) reduce the contention at the command queue. PCD addresses *command queue seizure* by giving priority to the requests targeted to the rank to be refreshed next, during the time before issuing the refresh. On the other hand, DCE lessens the *command queue seizure* further, by delaying the requests (to the refreshing rank) to enter the command queue from the transaction queue for the entire period of refresh. PCD and DCE together give an additive effect and improve performance further.

We compare the proposed EFGR technique with AR, DCE, and PCD mechanisms. For the implementation of AR, we choose n and m as 8 and 100, respectively. The length of each interval is equal to t_{REFI} in 1x mode. Figure 16 shows performance improvement obtained by AR 1x-2x, AR 1x-4x, DCE 1x, PCD 1x, and DCE + PCD 1x over FGR 1x mode refresh. It can be observed that AR 1x-2x and AR 1x-4x have a negative impact on performance, losing almost 1.5% and 2.3%, respectively, on average. Given that none of the benchmarks are benefited by FGR 2x/4x (Figure 6), the training period in these modes leads to a decrease in performance. Actually, by dynamically choosing between the refresh modes, AR 1x-2x and AR 1x-4x improve performance of FGR 2x and 4x modes by 1.1% and 3.6%, respectively.

For the workloads considered, DCE and PCD improve performance by 1% and 1.6%, respectively, on average. When employed together, because of their complementary nature, performance improves to 2.1%. Figures 17 and 18 give us the reason for the limited effectiveness of DCE and PCD. In FGR 1x mode, when a rank is under refresh, Figure 17 shows the number of commands at the beginning and at the end of the refresh

Fig. 16. Performance improvement of AR, PCD, DCE, PCD+DCE, EFGR, PCD+DCE+EFGR, and No Refresh techniques over FGR 1x mode.

Fig. 17. Number of commands waiting at the beginning and at the end of a refresh operation in FGR 1x mode. The requests waiting for the refreshing rank and the other ranks are shown separately.

Fig. 18. Percentage of the total number of refreshes when the command queue occupancy is below 50%, between 50% and 80%, and above 80% in FGR 1x mode, per benchmark.

for both refreshing rank and other ranks. Figure 18 provides the percentage of total refreshes during which the average occupancy of the Command Queue (CQ) is less than 50%, between 50% and 80%, and above 80%, respectively. Clearly, from the two figures we can see that the CQ is not the point of contention for many benchmarks. Hence, in the implementation of PCD, we consider 350 DRAM cycles before the beginning of a refresh operation as the mark to start the preemption. Also, instead of imposing DCE till the end of the ongoing refresh, we relax it 50 DRAM cycles ahead of the refresh end time so that available space in CQ can be filled up by the requests to the refreshing rank too and save some cycles. These numbers (350 and 50) for PCD and DCE are chosen after an extensive sensitivity analysis in our setup. The effectiveness of PCD + DCE actually improved over the basic PCD + DCE by 0.5%, from 1.5% overall.

Only benchmarks like *face* (5.8%), *ferret* (3.7%), *lib* (3.7%), *freq* (2.8%), *stream* (2.3%), and *swapt* (2.3%) get a benefit of DCE and PCD noticeably because for these benchmarks, the CQ occupancy is more than 80% for a considerable amount of refreshes (greater than 50%). Even though the techniques are simple to implement, the minimal improvement in performance shows that working around refresh is not an effective

option for the workloads with a high memory request rate and that the major impact of refresh on performance is due to the increased delay in servicing the requests by the refreshing rank.

Given that DCE-PCD mechanisms and EFGR are orthogonal to each other, they can be employed together. When applied with EFGR, DCE-PCD mechanisms are modified to do preemption or delay of requests' expansion, keeping in view only those banks to be involved in the refresh and not the entire rank. The combined performance improvement observed is 7.4%. The meager increase in performance over EFGR is due to the fact that even during the EFGR refresh, the CQ is not blocked by the requests of an entire rank but only by a few set of banks, reducing the scope of DCE-PCD.

6.8. Interaction with Other Techniques

EFGR, being a refresh handling technique, does not negatively impact the DRAM row-buffer/page management techniques (such as Minimalist Page Open [Kaseridis et al. 2011], application-aware page policy [Xie et al. 2013], conservative row activation [Fang and Zhu 2013], thread row buffers [Herrero et al. 2013]). Also, EFGR is orthogonal to the request handling and scheduling techniques (such as Eager writeback [Lee et al. 2000], virtual write queue [Stuecheli et al. 2010a], FR-FCFS [Rixner et al. 2000], staged memory scheduling [Ausavarungnirun et al. 2012], parallelism-aware batch scheduling [Mutlu and Moscibroda 2008], thread cluster memory scheduling [Kim et al. 2010]) and with simple modifications can be employed along with them. For example, PABS [Mutlu and Moscibroda 2008] can be modified to batch the requests of nonrefreshing and refreshing banks separately during an ongoing refresh and schedule the batches of nonrefreshing banks' requests prior to other batches (to refreshing banks).

7. OTHER REFRESH HANDLING WORKS

Concurrent refresh (CR) [Kiriata et al. 2005] was proposed for embedded DRAMs, wherein an attempt was made to hide the refresh latency completely. In CR, the refresh controller on the embedded DRAM device refreshes one or multiple rows *every* cycle. Complex circuitry consisting of up-and-down counters is used to identify a single bank or a set of two banks for refresh. In *each cycle*, the refresh controller checks for any bank conflict between refresh and memory requests. If no conflict is found, the banks will undergo refresh. An external REF command is required only when the refresh controller fails to refresh a bank due to continuous conflicts. The refresh controller bears the sole responsibility of ensuring that all the rows in the bank are refreshed periodically. This technique was not adopted for commodity DDR-style DRAMs for two reasons. First, a complex refresh controller circuitry is needed to conduct and track refresh operations, and second, the memory will be consuming peak power for almost all the time because refresh is likely to happen every cycle. Though our proposed EFGR too tries to overlap the refresh operations with memory request service, it does not include any disruptive changes to the existing refresh controller and/or the peripheral circuitry. Also, EFGR can be tuned to limit the peak power utilization.

Recently, two works have been proposed that aggressively try to overlap the memory requests and refresh in an attempt to eliminate the refresh overhead. Chang et al. [2014] consider the per-bank refresh feature of LPDDR memories and extend it to commodity DRAMs. The per-bank refresh feature is modified to consider idle banks for issuing refreshes over the traditional round-robin issue of refreshes. Further, the authors propose to exploit the subarray-level parallelism within a bank by refreshing a subarray and accessing another subarray within the same bank simultaneously. The Concurrent-REfresh-Aware DRAM Memory System (CREAM) [Zhang et al. 2014] proposes to reduce the refresh overhead by subrank-level refresh (SRLR) and subarray-level refresh (SALR) techniques. In SRLR, per refresh, the authors propose to refresh

a single subrank instead of the entire rank. The power savings are used to access the other subranks in the rank for memory accesses, hence achieving concurrency in refreshes and memory request service. On the other hand, SALR reduces the refresh impact further by accessing subarrays (within a subrank under refresh) that are not involved in the current refresh operation. SALR too tries to exploit the subarray-level parallelism within a bank to refresh a subarray and access another subarray for memory requests.

Refresh Pausing [Nair et al. 2013] is another refresh handling technique that targets reducing the impact of refresh on performance. Noticing that in every refresh operation multiple rows are refreshed one after another, the authors identify and define certain pause points (termed as Refresh Pause Points, RPPs) where the ongoing refresh operation can be paused and resumed later. At every RPP, the refresh operation is paused if one or more read requests is pending. Once the request is serviced, the refresh operation is resumed. In this way, Refresh Pausing improves performance by reducing the waiting time of memory requests. In high-density DRAM devices, identification of RPPs is not trivial because within a single refresh operation, pages to be refreshed are divided into subgroups and each subgroup's refresh operation is staggered with some delay to avoid peak power consumption. In such a scenario, finding an RPP is questionable.

Smart Refresh [Ghosh and Lee 2007] proposes to save energy by avoiding refreshes to recently accessed rows. In Coordinated Refresh [Bhati et al. 2013], the authors propose scheduling refreshes when the DRAM device is in low-power mode and target achieving both energy savings and performance improvement. Exploiting the variation in data retention times of DRAM cells, RAPID [Venkatesan et al. 2006] proposes a software method to place data in pages with high retention times and avoid unnecessary refreshes. RAIDR [Liu et al. 2012], on the other hand, groups the DRAM pages into bins based on their retention times. The bins with high/low retention time are refreshed less/more frequently and in the process save both power and improve performance. Flicker [Liu et al. 2011] too proposes a software technique to reduce the refresh power consumption by reducing the refresh rate of the pages containing noncritical data. Baek et al. [2013] propose two software-oriented techniques, namely, Refresh Incessantly but Occasionally (RIO) and Placement-Aware Refresh in Situ (PARIS). RIO removes (logically) the page frames that have weak DRAM cells (with a lesser retention period) and hence reduces the number of times memory is refreshed. PARIS selectively refreshes only those DRAM rows that are currently in use, effectively reducing the refresh activity when DRAM capacity is not fully utilized. Our proposed EFGR feature is orthogonal to these techniques and can be employed along with them.

In Elastic Refresh [Stuecheli et al. 2010b], the authors use prediction to avoid conflict of refresh operations with memory requests. By exploiting the *postponing of refresh commands* feature provided by the JEDEC Standard, Elastic Refresh schedules refreshes at predicted opportune times. It should be noted that Elastic Refresh and EFGR complement each other.

8. CONCLUSION

For high-density DRAM devices, the refresh operation degrades performance significantly by blocking an entire rank for t_{RFC} time. Realizing that during a Fine Granularity Refresh (FGR), only few banks of a rank are used for refreshing, in this work we proposed an Enhanced FGR (EFGR) feature for DDR4 devices. The EFGR feature, obtained by including three optimizations in the basic FGR feature, aims to expose the bank-level parallelism even when a rank is under refresh, that is, to service requests and refresh different banks simultaneously. Through the first optimization, we show that, with simple modifications to the peripheral circuitry of the DRAM, the

nonrefreshing banks can be made accessible to the memory controller. The second optimization, when applied, demonstrates that even when only one bank is allowed to be active in parallel with an ongoing refresh, a gain of 4.6% in performance is obtained. The third optimization, termed *selective precharging*, improves performance of EFGR further by 0.9% by avoiding unnecessary precharges and relaxing the pre-refresh condition. Put together, the three optimizations of EFGR help in recovering almost 56% of performance lost due to refresh operations.

Comparison with a state-of-the-art refresh handling technique shows that EFGR clearly outperforms it. On a closer study of the two techniques, we realize that they are complementary in nature and when employed together improve performance further. A simple and effective design and the complementary nature of our proposed EFGR feature make it a compelling choice for future commercial DDR-style DRAM devices.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their thorough and in-depth reviews, which immensely helped in improving the draft.

REFERENCES

- Rachata Ausavarungrun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. 2012. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *International Symposium on Computer Architecture (ISCA)*. 416–427.
- Seungjae Baek, Sangyeun Cho, and Rami Melhem. 2013. Refresh Now and Then. In *IEEE Transactions on Computers* 99 (2013), 1–14.
- Ishwar Bhati, Zeshan Chishti, and Bruce Jacob. 2013. Coordinated Refresh: Energy Efficient Techniques for DRAM Refresh Scheduling. In *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED'13)*.
- Karthik Chandrasekar, Benny Akesson, and Kees Goossens. 2011. Improved Power Modeling of DDR SDRAMs. In *Euromicro Conference on Digital System Design (DSD'11)*. 99–108.
- Kevin Kai-Wei Chang, Donghyuk Lee, Zeshan Chishti, Alas R. Alameldeen, Chris Wilkerson, Yoongu Kim, and Onur Mutlu. 2014. Improving DRAM Performance by Parallelizing Refreshes with Accesses. In *IEEE International Symposium on High Performance Computer Architecture (HPCA'14)*.
- Niladrish Chatterjee, Rajeev Balasubramanian, Manjunath Shevgoor, Seth H. Pugsley, Aniruddha N. Udipi, Ali Shafiei, Manu Awasthi, Kshitij Sudan, and Zeshan Chishti. 2012. USIMM: The Utah Simulated Memory Module a Simulation Infrastructure for the JWAC Memory Scheduling Championship. Technical Report.
- JEDEC DDR4. 2012. JEDEC DDR4 SDRAM Standard. Retrieved from <http://www.jedec.org/standards-documents/docs/jesd79-4>.
- Kun Fang and Zhichun Zhu. 2013. Conservative Row Activation to Improve Memory Power Efficiency. In *International ACM Conference on International Conference on Supercomputing (ICS'13)*. 81–90.
- Mrinmoy Ghosh and Hsien-Hsin S. Lee. 2007. Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs. In *IEEE/ACM International Symposium on Microarchitecture (MICRO'07)*. 134–145.
- E. Herrero, J. Gonzalez, R. Canal, and D. Tullsen. 2013. Thread Row Buffers: Improving Memory Performance Isolation and Throughput in Multiprogrammed Environments. In *IEEE Transactions on Computers* 62, 9 (2013), 1879–1892.
- Micron Technology Inc. 2007. TN-41-01: Calculating Memory System Power for DDR3. Technical Report.
- Kiyoo Itoh. 2001. *VLSI Memory Chip Design*. Springer.
- Bruce Jacob, Spencer Ng, and David Wang. 2007. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann.
- Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. 2011. Minimalist Open-Page: A DRAM Page-mode Scheduling Policy for the Many-Core Era. In *IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*. 24–35.
- Brent Keeth, B. Jacob Baker, Brian Johnson, and Feng Lin. 2007. *DRAM Circuit Design: Fundamental and High-Speed Topics* (2nd ed.). Wiley-IEEE Press.
- Samira Khan, Donghyuk Lee, Yoongu Kim, Alaa R. Alameldeen, Chris Wilkerson, and Onur Mutlu. 2014. The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative

- Experimental Study. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'14)*. 519–532.
- Charles A. Kilmer, Kyu hyoun Kim, Warren E. Maule, and Vipin Patel. 2012. Memory System with a Programmable Refresh Cycle. United States Patent Application 0151131 A1.
- Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*. 65–76.
- T. Kirihata, P. Parries, D. R. Hanson, H. Kim, J. Golz, G. Fredeman, R. Rajeevakumar, J. Griesemer, N. Robson, A. Cestero, B. A. Khan, Geng Wang, M. Wordeman, and S. S. Iyer. 2005. An 800-MHz Embedded DRAM with a Concurrent Refresh Mode. In *IEEE Journal of Solid-State Circuits* 40, 6 (2005), 1377–1387.
- Hsien-Hsin S. Lee, Gary S. Tyson, and Matthew K. Farrens. 2000. Eager Writeback—A Technique for Improving Bandwidth Utilization. In *ACM/IEEE International Symposium on Microarchitecture (MICRO'00)*. 11–21.
- Myoung Jin Lee and Kun-Woo Park. 2010. A Mechanism for Dependence of Refresh Time on Data Pattern in DRAM. *IEEE Electron Device Letters* 31, 2 (2010), 168–170.
- Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu. 2013. An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms. In *International Symposium on Computer Architecture (ISCA'13)*. 60–71.
- Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. 2012. RAIDR: Retention-Aware Intelligent DRAM Refresh. In *International Symposium on Computer Architecture (ISCA'12)*. 1–12.
- Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2011. Flicker: Saving DRAM Refresh-Power through critical data partitioning. *SIGARCH Computer Architecture News* 39, 1 (2011), 213–224.
- JEDEC LPDDR2. 2009. JEDEC LPDDR2 SDRAM Standard. Retrieved from <http://www.jedec.org/standards-documents/docs/jesd209-2f>.
- Micron Technology LPDDR2. 2011. 2Gb: x16, x32 Mobile SDRAM S4. (2011).
- JEDEC LPDDR3. 2012. JEDEC LPDDR3 SDRAM Standard. Retrieved from <http://www.jedec.org/standards-documents/results/jesd209-3>.
- MSC. 2012. Memory Scheduling Championship (MSC). Retrieved from <http://www.cs.utah.edu/~rajeev/jwac12>.
- Janani Mukundan, Hillery Hunter, Kyu-hyoun Kim, Jeffrey Stuecheli, and José F. Martínez. 2013. Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM Systems. *SIGARCH Computer Architecture News* 41, 3 (2013), 48–59.
- Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *International Symposium on Computer Architecture (ISCA'08)*. 63–74.
- Prashant Nair, Chia-Chen Chou, and Moinuddin K. Qureshi. 2013. A Case for Refresh Pausing in DRAM Memory Systems. In *International Symposium on High Performance Computer Architecture (HPCA'13)*. 627–638.
- P. J. Restle, J. W. Park, and B. F. Lloyd. 1992. DRAM Variable Retention Time. In *International Electron Devices Meeting (IEDM'92)*. 807–810.
- Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. 2000. Memory Access Scheduling. *SIGARCH Computer Architecture News* 28, 2 (2000), 128–138.
- B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. 2011. IBM POWER7 Multicore Server Processor. *IBM Journal of Research and Development* 55, 3 (2011), 191–219.
- Allan Snaveley and Dean M. Tullsen. 2000. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. *SIGARCH Computer Architecture News* 28, 5 (2000), 234–244.
- Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. 2010a. The Virtual Write Queue: Coordinating DRAM and Last-level Cache Policies. In *International Symposium on Computer Architecture (ISCA'10)*. 72–82.
- Jeffrey Stuecheli, Dimitris Kaseridis, Hillery C. Hunter, and Lizy K. John. 2010b. Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory. In *IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*. 375–384.
- T. Venkata Kalyan, K. Ravi, and M. Madhu. 2014. Scattered Refresh: An Alternative Refresh Mechanism to Reduce Refresh Cycle Time. In *Asia and South Pacific Design Automation Conference (ASP-DAC'14)*. 598–603.

- R. K. Venkatesan, S. Herr, and E. Rotenberg. 2006. Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-non-volatile DRAM. In *International Symposium on High-Performance Computer Architecture (HPCA'06)*. 155–165.
- T. Vogelsang. 2010. Understanding the Energy Consumption of Dynamic Random Access Memories. In *IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*. 363–374.
- Intel Xeon. 2014. Intel Xeon E7 88xx Series Processors. Retrieved from <http://ark.intel.com/products/family/78584>.
- Mingli Xie, Dong Tong, Yi Feng, Kan Huang, and Xu Cheng. 2013. Page Policy Control with Memory Partitioning for DRAM Performance and Power Efficiency. In *IEEE International Symposium on Low Power Electronics and Design (ISLPED'13)*. 298–303.
- D. S. Yaney, C. Y. Lu, R. A. Kohler, M. J. Kelly, and J. T. Nelson. 1987. A Meta-Stable Leakage Phenomenon in DRAM Charge Storage—Variable Hold Time. In *International Electron Devices Meeting (IEDM'87)*. 336–339.
- Tao Zhang, Matt Poremba, Cong Xu, Guangyu Sun, and Yuan Xie. 2014. CREAM: A Concurrent-Refresh-Aware DRAM Memory System. In *IEEE International Symposium on High Performance Computer Architecture (HPCA'14)*. 368–379.

Received January 2014; revised July 2014; accepted July 2014