CLARA: Circular Linked-List Auto and Self Refresh Architecture

Conference Paper · January 2016

DOI: 10.1145/2989081.2989084

6 authors, including:

Mike O'Connor
NVIDIA
44 PUBLICATIONS **833** CITATIONS
SEE PROFILE

Evgeny Bolotin
Technion - Israel Institute of Technology
21 PUBLICATIONS **795** CITATIONS
SEE PROFILE

Joel S. Emer
NVIDIA
124 PUBLICATIONS **5,975** CITATIONS
SEE PROFILE

CLARA: Circular Linked-List Auto and Self Refresh Architecture

Aditya Agrawal
adityaa@nvidia.com

Mike O'Connor
moconnor@nvidia.com

Evgeny Bolotin
ebolotin@nvidia.com

Niladrish Chatterjee
nchatterjee@nvidia.com

Joel Emer
jemer@nvidia.com

Stephen Keckler
skeckler@nvidia.com

ABSTRACT

With increasing DRAM densities, the performance and energy overheads of refresh operations are increasingly significant. When the system is active, refresh commands render DRAM banks unavailable for increasing periods of time. These refresh operations can interfere with regular memory operations and hurt performance. In addition, when the system is idle, DRAM self-refresh is the dominant source of energy consumption, and it directly impacts battery life and standby time. Prior refresh reduction techniques seek to reduce active-mode auto-refresh energy, reduce self-refresh energy, improve performance, or some combination thereof. In this paper, we present *CLARA*, a circular linked-list based refresh architecture which meets all three goals with very low overheads and without sacrificing DRAM capacity. This approach exploits the variation in retention time at a chip granularity as opposed to a DIMM-wide, rank granularity in prior work. CLARA reduces auto- and self-refresh by 86.2%, independent of workload. Auto refresh reduction improves average CPU performance by 3.1% and 6.5% in the normal and extended temperature range, respectively. GPU performance improves by 2.1% on average in the extended temperature range. DRAM idle power during self-refresh is reduced by 44%. The area overhead of CLARA in the DRAM is about 0.085% and negligible in the memory controller.

CCS Concepts

•**Hardware** → **Dynamic memory;** •**Computer systems organization** → *Architectures;*

Keywords

DRAM, Auto refresh, Self refresh

1. INTRODUCTION

DRAM is a dynamic memory technology which requires periodic refresh operations to maintain data integrity. These refresh operations incur energy and performance costs, however. When the system is active, *auto-refresh* commands render one or more DRAM banks unavailable for some time, delaying processor requests requiring those banks and reducing performance. When the system is idle, the DRAM is responsible for refreshing itself. These *self-refresh* operations are a significant fraction of the energy of an idle system, and this energy consumption directly impacts battery life and standby time in mobile devices. As DRAM densities continue to increase, more refresh operations have to be performed in a given time period. As a result, the impact of refresh on system performance and power are also increasing. DRAMs are increasingly operating at higher temperatures due to close proximity to high-power processors in 2.5D/3D integrated systems as well. Operating in the extended temperature range ($85-95\,^{\circ}C$) doubles the required refresh rate, further increasing the performance and energy costs of refresh operations.

It is well-known that different DRAM cells/rows have different charge retention times and, hence, different refresh requirements [24, 31]. This property has been effectively exploited to reduce the cost of refreshes and improve yields in a number of earlier proposals [47, 34, 40, 6, 16, 10, 48]. These techniques typically profile the retention time of each DRAM row, store that information in main memory, memory controller, external EPROM device, or OS page table, and then refresh each row at its required rate. The information can be stored in a variety of data structures such as a bloom filter, array, linked list, fat tree, or binary tree. For example, a bloom filter [34] can be used to test the presence of a DRAM row in a set before sending a refresh request. A linked-list [48] can be used to create an arbitrary refresh sequence of DRAM rows. These techniques aim to reduce auto-refresh energy, reduce self-refresh energy, improve performance, or some combination of these.

We propose a linked-list based refresh scheme, *CLARA*, which also takes advantage of retention time variation to reduce the number of refresh operations required and improve performance. It achieves these goals with very low overheads and without sacrificing DRAM capacity. Our approach exploits the fact that a refresh operation implicitly accesses the data in the row being refreshed in order to both store the refresh characterization in area-efficient DRAM cells, and to access that information on a refresh to indicate the next row to be refreshed. This approach enables the following key benefits:

- Reduces the number of active mode auto refresh commands that must be sent to the DRAM, improving energy efficiency and performance.
- Reduces the number of self-refresh operations in the low-power standby mode, saving energy.
- Exploits fine-grained retention time variation within a device.
- Very low hardware overheads (<0.1% area).

The CLARA architecture reduces both auto- and self- refresh by 86.2%, independent of the workload. The auto-refresh reduction

MEMSYS '16, October 03-06, 2016, Alexandria, VA, USA

DOI: http://dx.doi.org/10.1145/2989081.2989084

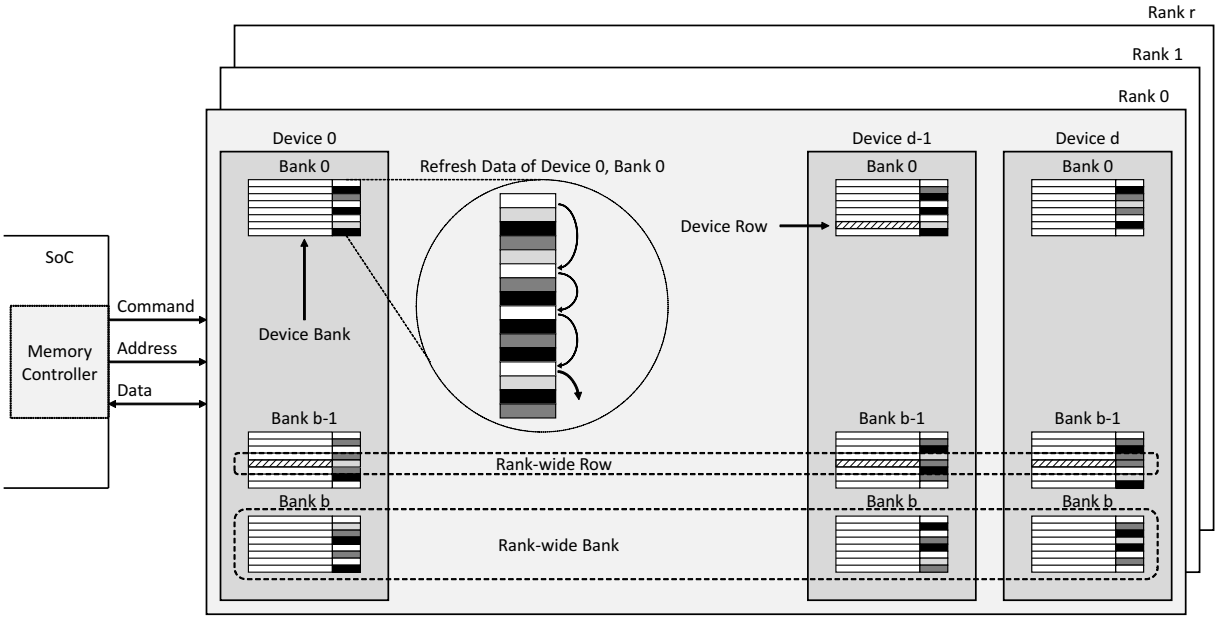


Figure 1: A DRAM channel organization. The circular inset shows the CLARA refresh data as described in Sec. 3.

improves CPU performance by 3.1% and 6.5% in the normal and extended temperature range, respectively. GPU performance improves by 2.1% in the extended temperature range. CLARA saves 44% of DRAM idle power due to the reduction of self-refresh operations.

2. BACKGROUND

2.1 DRAM Organization

The DRAM main memory is organized into channels. As shown in Fig. 1, each channel has independent command, address and data signals. Each channel contains one or more independent ranks. Each rank typically consists of multiple DRAM chips, often 4, 8 or 16. A DRAM chip is also referred to as a DRAM device. Each DRAM device, in turn, has one or more independent banks, henceforth referred to as ***device banks***. Each device bank typically has thousands of rows, henceforth referred to as ***device rows***.

The DRAM devices in a rank work in unison and appear as a single entity to the memory controller. As shown in Fig. 1, multiple device banks together constitute a ***Rank-wide bank*** and multiple device rows together constitute a ***Rank-wide row***.

2.2 Refresh Mechanism

According to DDR standards [17, 18, 20, 36] each row should be refreshed once every 64 *ms* in the normal temperature range ($< 85\ ^\circ$C) and once every 32 *ms* in the extended temperature range ($85-95\ ^\circ$C). For the following discussion, we will assume the normal temperature range of operation. Let N, be the number of rows in a bank. In the past, the time interval between refresh requests to a bank, t_{REFI}, was 64 ms/N, and each request refreshed 1 row. With increasing DRAM densities the number of rows per bank (N) has also increased. The value of t_{REFI} has been held constant, however, at 64 $ms/8192 \approx 7800\ \mu s$, requiring multiple rows, M, to be refreshed with every request.

Each refresh request lasts for a refresh cycle time, t_{RFC}, which includes the time to refresh M rows, precharge the bank and recover the charge pump. Refreshing multiple rows allows the DRAM vendor to optimize the process and amortize precharging and recovering the charge pump over several row-refresh operations.

Serial and Parallel Refresh. While DRAM vendors do not disclose their specific implementation of multi-row refresh, timing parameters suggest that the mechanism is serial when $M \leq 4$ and parallel when $M > 4$. Table 1 shows the refresh cycle time, t_{RFC} when different number of rows, M are refreshed using the DDR4 Fine Granularity Refresh (FGR) [20] mode. Let t_{RC}[1], be the time to refresh a single row and t_{rec}, be the recovery time. When $M \leq 4$, $t_{RFC} = M \times t_{RC} + t_{rec}$. This suggests that the rows are refreshed sequentially. However, when $M > 4$, $t_{RFC} < M \times t_{RC} + t_{rec}$. This suggests that multiple rows in different subarrays, are refreshed in parallel.

Capacity	N	FGR Mode	M	$M.t_{RC}+t_{rec}$	t_{RFC}
4 Gb	32 K	4x	1	110 ns	110 ns
4 Gb	32 K	2x	2	160 ns	160 ns
4 Gb	32 K	1x	4	260 ns	260 ns
8 Gb	64 K	1x	8	460 ns	350 ns
16 Gb	128 K	1x	16	860 ns	480 ns

Table 1: Serial and parallel multi-row refresh.[2]

Auto and Self Refresh Schemes. To maintain data, DRAM rows must be refreshed in both active (auto-refresh) and idle (self-refresh) modes. Different DDR standards use different auto-refresh schemes such as *all-bank*, *per-bank* and *single-bank* refresh. In the *all-bank* scheme [18], a refresh command refreshes multiple rows in every bank of the rank. The entire rank is unavailable for t_{RFC}. In the *per-bank* scheme [36], a refresh command refreshes multiple rows in a single DRAM bank. However, the banks are refreshed in

[1] t_{RC} is the row cycle time and is equal to the row activate time, t_{RAS} plus the row precharge time, t_{RP}.

[2] Number of rows, N is for a DDR4 x8 device. $t_{RC} = 50\ ns$ for the slowest DDR4 part i.e. DDR4-1600 (12-12-12). t_{rec} was estimated to be 60 *ns*. t_{RFC} values are from DDR4 standard [20].

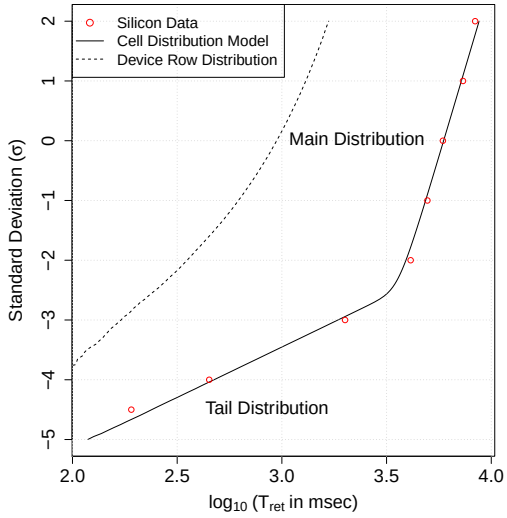


Figure 2: DRAM cell [24] and device row T_{ret} distribution.

a strict round robin order. In the *single-bank* scheme [25], the banks can be refreshed in any order. This allows the memory controller to schedule refresh operations to otherwise idle-banks, optimizing performance. DDR4's Fine Granularity Refresh (FGR) [20] is similar to all-bank refresh except that it allows refreshing one-half and one-quarter number of rows on each refresh command, reducing the period for which the rank is unavailable.

In all the above schemes the refresh row address is maintained internally by the DRAM devices and is not sent by the memory controller. Earlier DRAMs provided a *RAS-only* refresh scheme which allow the memory controller to specify a row address for the refresh operation. This refresh mode in no longer supported in the most recent DRAM families. A regular DRAM 'activate' operation followed by a 'precharge' can still allow the memory controller to effectively refresh a given row, however. These targeted-row refresh operations are used by the refresh reduction approaches in [23, 34].

2.3 Retention Time Variation

It is well known that different DRAM cells have different charge retention times. Kim and Lee [31] obtained the distribution for 100, 60 and 50 *nm* and have projections for 10 *nm*. A model for the cell retention time distribution was first proposed by Hamamoto *et al.* [24]. The retention time of a cell, T_{ret} can be expressed as $T_{ret} = A \times e^{(E_a/kT)}$ where, E_a is the trap activation energy, k is the Boltzmann constant, T is the absolute temperature, and A is the constant of proportionality. A normal distribution in the trap activation energy produces a log-normal distribution of cell T_{ret}.

As shown in Fig. 2, the cumulative distribution function (CDF) of the cell retention time consists of a *Main Distribution* and a *Tail Distribution*. Hamamoto *et al.* determined that $\mu(E_a) = 0.68\ eV$ and $\sigma(E_a) = 0.008\ eV$ for the main distribution; and $\mu(E_a) = 0.77\ eV$ and $\sigma(E_a) = 0.039\ eV$ for the tail distribution.

The retention time of a device row can be obtained by determining the minimum T_{ret} of all cells constituting the row. Fig. 2 also shows the device row T_{ret} distribution. As shown in the plot and tabulated below, the percentage of device rows with $64\ ms <= T_{ret} < 128\ ms$ is only about 0.03%. More than 90% of device rows have a $T_{ret} >= 512\ ms$. However, current DRAM devices refresh each row every 64 *ms*. This is very pessimistic and results in performance and power loss. We exploit this variation to reduce both auto and self-refresh, thereby improving performance and power.

3. CLARA ARCHITECTURE

T_{ret}	Percentage
64 *ms*	∼0.03
128 *ms*	∼0.60
256 *ms*	∼7.5
512 *ms*	∼91

Table 2: DRAM device row T_{ret} distribution.

CLARA exploits the variation in retention time at a DRAM device level to reduce both auto and self refresh. In order to take advantage of this variation, the retention characteristics of each device row must first be profiled. We discuss this profiling and associated guard-banding in detail in Sec. 4.1. For the purposes of this section, we assume the device rows have been profiled. We describe how the device rows are categorized according to their retention time, and then we describe the linked-list architecture that enables only the necessary rows to be refreshed at the required rates.

3.1 Categorizing the Rows

In order to simplify the hardware, we group rows with similar retention times into groups that will be refreshed at the same rate. Depending on the T_{ret}, we bin the device row into one of 4 categories viz. 64 *ms*, 128 *ms*, 256 *ms* or 512 *ms*. We call the number of rows in a device bank with T_{ret} of 64 *ms*, 128 *ms* and 256 *ms* as α, β, and γ respectively. If the total number of rows in a device bank is N, the number of device rows with T_{ret} of 512 $ms = N - \alpha - \beta - \gamma$.

We call an interval of 64 *ms* an *epoch*. Thus, a device row with T_{ret} = 64 *ms* will require a refresh every epoch, a device row with T_{ret} = 128 *ms* will require a refresh every alternate epoch, and so on, as summarized in Table 3.

From the table, we observe that in epochs 0, 2, 4 and 6, only rows with T_{ret} = 64 *ms*, *i.e.* α rows, require refresh; in epochs 1 and 5, rows with T_{ret} = 64 *ms* and 128 *ms*, *i.e.* $\alpha+\beta$ rows, require refresh; and in epoch 3, rows with T_{ret} = 64 *ms*, 128 *ms* and 256 *ms*, *i.e.* $\alpha+\beta+\gamma$ rows, require refresh. Finally, in epoch 7, all rows, *i.e.* N rows, require refresh, and is the same as the conventional refresh scheme. The different epochs have different refresh requirements. Overall, by refreshing only the required rows in each epoch, we can reduce refresh overheads. Our proposed linked list based refresh architecture achieves this goal.

Epoch (64 *ms*)	T_{ret} = 64 *ms*	T_{ret} = 128 *ms*	T_{ret} = 256 *ms*	T_{ret} = 512 *ms*	# Refresh Required
0	Y				α
1	Y	Y			$\alpha+\beta$
2	Y				α
3	Y	Y	Y		$\alpha+\beta+\gamma$
4	Y				α
5	Y	Y			$\alpha+\beta$
6	Y				α
7	Y	Y	Y	Y	N

Table 3: Variable refresh requirements.

3.2 Linked List Architecture

The key insight behind the CLARA architecture is that when a row in a DRAM device is refreshed, the data in the row is read into the sense-amplifiers in order to be restored back into the bit-cells. We take advantage of this implicit read operation in order to traverse a linked-list of rows with identical refresh requirements. In this section we describe how the linked-list is stored and organized in the device bank, and then explain how the refresh logic in the

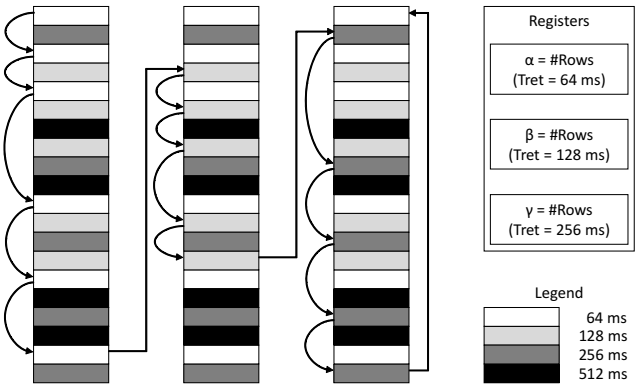


Figure 3: Circular linked-list in a device bank.

device bank uses this linked-list to refresh only the required device rows in each epoch, during both auto and self refresh.

Linked List Storage and Organization. Fig. 1 shows a device bank with device rows. We propose to add a few bits to each device row. These additional bits are called refresh-data bits. The data and refresh-data bits share the same access logic such as word lines, row decoders etc., and are activated/refreshed at the same time. We exploit the fact that a row refresh operation effectively reads the entire row (data and refresh-data). In addition to being written back to the array, the refresh-data is used internally by the device bank refresh logic to determine the next refresh row address. The refresh-data bits are not read or written by the memory controller except during configuration.

As shown in the circular inset in Fig. 1 and in greater detail in Fig. 3, the refresh-data within each bank is organized so as to form a circular linked list of rows. The refresh-data bits hold only the address offset to the next node (row). The first row of each device bank, i.e. $address = 0$, is assumed to have a T_{ret} of 64 *ms* and serves as the head of the linked list. As shown in Fig. 3, we first link all rows with T_{ret} = 64 *ms*, then link all rows with T_{ret} = 128 *ms* and then link all rows with T_{ret} = 256 *ms*. The last row with T_{ret} = 256 *ms* points to the first row. In Fig. 3 the refresh-data has been replicated for clarity. In Sec. 3.5, we describe in detail how to generate this linked list.

In addition, there are three registers which hold the count of rows with T_{ret} = 64 *ms*, 128 *ms* and 256 *ms*, *i.e.* the values α, β, and γ respectively. It follows that $\alpha+\beta+\gamma$ is equal to the length of the linked list.

From Table 3, in epoch 7, when rows with T_{ret} = 512 *ms* are refreshed, all other rows are refreshed as well. This is the same as conventional refresh and no refresh-data is used. Therefore, there is no need to include rows with T_{ret} = 512 *ms* in the linked list or provide a register to hold the count of rows with T_{ret} = 512 *ms*.

Basic Operation. The linked-list organization allows us to refresh only the required device rows in each epoch. In epochs 0, 2, 4 and 6, only rows with T_{ret} = 64 *ms* require refresh, which are the first α entries in the linked list; in epochs 1 and 5, rows with T_{ret} = 64 *ms* and 128 *ms* require refresh, which are the first $\alpha+\beta$ entries in the linked list; and in epoch 3, rows with T_{ret} = 64 *ms*, 128 *ms* and 256 *ms* require refresh, which are the first $\alpha+\beta+\gamma$ entries in the linked list. In epoch 7, all N rows are refreshed and the linked list is not used. At the beginning of each epoch we start at the head of the linked list. Also, since the organization is a linked list and is circular, it can easily handle additional requests in any epoch. A few additional requests can arrive in the case of multi-row refresh and when the system has multiple banks and devices, as explained below.

For self-refresh, the control logic in the bank uses the three register values (α, β, γ) to generate the appropriate number of refresh commands in every epoch. For auto-refresh, the memory controller maintains a copy of the same three register values, and sends the appropriate number of auto-refresh commands in every epoch. Note that the self-refresh logic and the memory controller do not need to know which device-rows require refresh, only how many require refresh in a particular epoch.

Supporting Multi-Row Refresh. In conventional gigabit devices, each request typically refreshes 2, 4, 8 or more rows. In CLARA, we refresh the same number of rows in every command. However, we obtain the row addresses from the linked list.

Assuming, each command refreshes M rows, it follows that the device bank should receive at least $\lceil \alpha/M \rceil$ requests in epochs 0, 2, 4 and 6; at least $\lceil (\alpha+\beta)/M \rceil$ requests in epochs 1 and 5; and at least $\lceil (\alpha+\beta+\gamma)/M \rceil$ requests in epoch 3. In epoch 7, it should receive $\lceil N/M \rceil$ requests.

A refresh request can arrive from either the memory controller or the internal self-refresh logic. On receiving a refresh request, the bank refreshes M rows of the linked list following the steps in Procedure 1, where N is the number of rows in a device bank, *row* is the array of rows in the bank, *refreshdata* is the associated array of refresh-data in the bank, *epoch* is the current refresh epoch, and *address* is the refresh row address. *address* is reset to zero, at the beginning of every epoch.

Procedure 1 Linked-list based device bank refresh logic.

```
count = 0
while count < M do
  Refresh row[address]
  if epoch = 7 then
    address = address + 1
  else
    address = address + 1 + refreshdata[address]
  end if
  address = address mod N
  count = count + 1
end while
```

3.3 Supporting Multiple Banks & Devices

The description above focused on a single bank within a single device. In practice, a memory channel consists of multiple ranks each with multiple devices and multiple banks. However, it is important to note that all banks in all devices have the same statistical distribution (mean and variance) of retention times. Therefore, the values of α, β and γ are similar for all device banks. For example, the number of rows with T_{ret} = 64 *ms* is similar across all device banks.

Therefore in CLARA for auto-refresh, the memory controller stores and uses only the values of $max(\alpha)$, $max(\beta)$ and $max(\gamma)$. This results in a small number of additional refresh operations in some device-banks. This is easily handled by the circular linked-list design. However, the big advantage is very small storage and control overheads in the memory controller, as explained below with the help of an example. For self-refresh, the control logic in every device bank uses the local values of α, β, and γ, as already explained.

Let's assume there are two devices P and Q in a rank, each with 2 banks (0 and 1). Let the distribution of T_{ret} i.e. values of α, β and γ, for the 4 device banks be as shown in Table 4. These values are for a bank with $N = 64$ *K* rows and follow the distribution as obtained in [24]. For these values, $max(\alpha) = A = 28$, $max(\beta) = B = 440$

and $max(\gamma) = \Gamma = 5225$. Also, assume each auto-refresh command refreshes $M = 8$ rows. Given the values of A, B, Γ, N and M, we can calculate the number of auto-refresh commands in each epoch.

Count	P0	Q0	P1	Q1
64 *ms* (α)	17	28	23	18
128 *ms* (β)	407	386	396	440
256 *ms* (γ)	5080	5179	4997	5225
512 *ms*	60032	59943	60120	59853
Total	65536	65536	65536	65536

Table 4: Example T_{ret} distribution of 4 device banks.

The memory controller sends $\lceil A/8 \rceil$ i.e. 4 requests in epochs 0, 2, 4 and 6; $\lceil (A+B)/8 \rceil$ i.e. 59 requests in epochs 1 and 5; and $\lceil (A+B+\Gamma)/8 \rceil$ i.e. 712 requests in epoch 3. In epoch 7, all rows are refreshed and the controller sends the usual 8 K requests. Overall, in 8 epochs, the CLARA memory controller sends $4\times4+2\times59+712+8192 = 9038$ requests. A conventional memory controller will send 8 K requests every epoch and 64 K overall. This is a significant reduction in the number of auto-refresh commands over the conventional scheme.

These values can be used for all refresh schemes such as *all-bank*, *per-bank*, *single-bank*, and *Fine Granularity Refresh* (FGR). In FGR, the memory controller can setup the banks to refresh one-half or one-quarter rows every auto-refresh command. In this mode, the memory controller can repeat the above calculations using $M = 4$ or $M = 2$, instead of $M = 8$; or simply double or quadruple the previously calculated values. Using per bank α, β, and γ values instead of A, B, and Γ values would further reduce the single-bank refresh by $\approx 0.27\%$ and the all-bank refresh by $\approx 0.03\%$. The storage and complexity reduction in the memory controller by using just 3 values far outweigh these additional savings.

3.4 Hardware Implementation

Fig. 4 shows the hardware implementation of the linked-list based refresh logic in the device bank. The additional blocks required by our scheme are shown in gray, and consist of a 3-bit epoch counter, an adder and a multiplexer. Every epoch i.e. 64 *ms*, the timer increments the epoch counter as well as resets the row address to zero. Depending on the epoch, the multiplexer chooses the address from either the linked list or the conventional address incrementor.

Timing Overhead. As it is in conventional devices, this address generation logic is used when the row is being written back from the sense amplifiers to the array, and it is not in the critical path of the refresh operation. The delay introduced by an adder and a multiplexer is insignificant compared to the row precharge time, t_{RP}, which is about 15 *ns* [20, 2, 19] and hence, there is no timing overhead.

Area Overhead. As mentioned in Sec. 3.2, the refresh-data bits are stored as additional bits with every row. The refresh-data storage overhead is, however, very small. Consider, for example, a state of the art DDR4, 8 Gb, x8 device [20]. Each device bank has $N = 65536$ rows, with a device row size of 2 *KB*. If the number of refresh-data bits, $L = log_2 N = 16$ bits, the storage overhead is $(16/16384) = 0.098\%$. This is an insignificant overhead. In Section 6.1, we show that we can reduce this overhead even more.

The area overheads in the memory controller are three 16-bit registers, an adder, a shifter and some control logic to send the appropriate number of auto-refresh commands in every epoch. These are very small overheads. CLARA uses the same auto-refresh command as in conventional systems and does not introduce any new refresh commands.

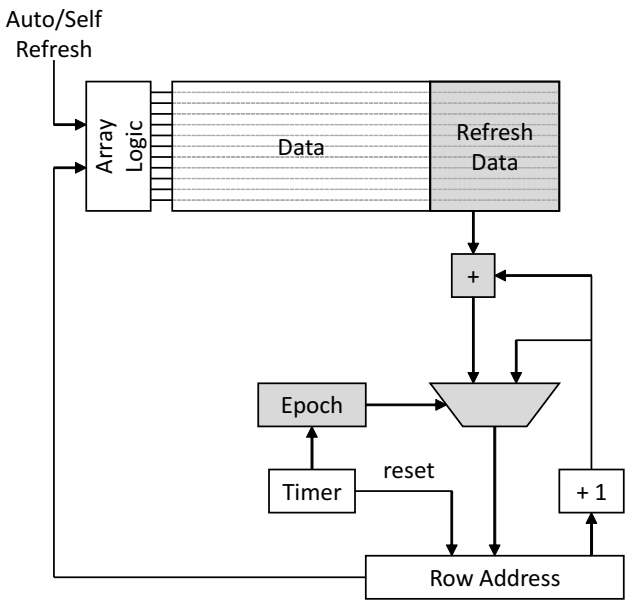


Figure 4: Linked-list based device bank refresh logic.

Power Overhead. Accessing 16 extra bits for every 16K bits has insignificant power overhead of 0.098% during activate, precharge and refresh. There is no overhead on read, write or bus power. Overall, power saved during active (2-6%) and idle (44%) far outweigh the overheads. (See evaluation).

3.5 Device Bank Linked-List Generation

As mentioned in Sec. 3.2, the refresh-data within each device bank is organized so as to form a circular linked-list of rows. The refresh-data bits hold the address offset to the next node. Let *offset* be the address offset between two nodes of the linked list. If the number of refresh-data bits is L, then the maximum *offset* distance, $D = 2^L$, (1 through 2^L). If N is the number of rows in a bank and if $L = log_2 N$, then $D = N$, i.e. the maximum offset distance is equal to the number of rows in a bank. In this case, constructing the linked list as described in Sec. 3.2 is always possible and straightforward.

When $L < log_2 N$, three cases arise. We will explain the linked list construction for each case using an example. For these examples, let $N = 20$, $L = 3$, and $D = 8$.

Case 1: *offset* $\leq D$

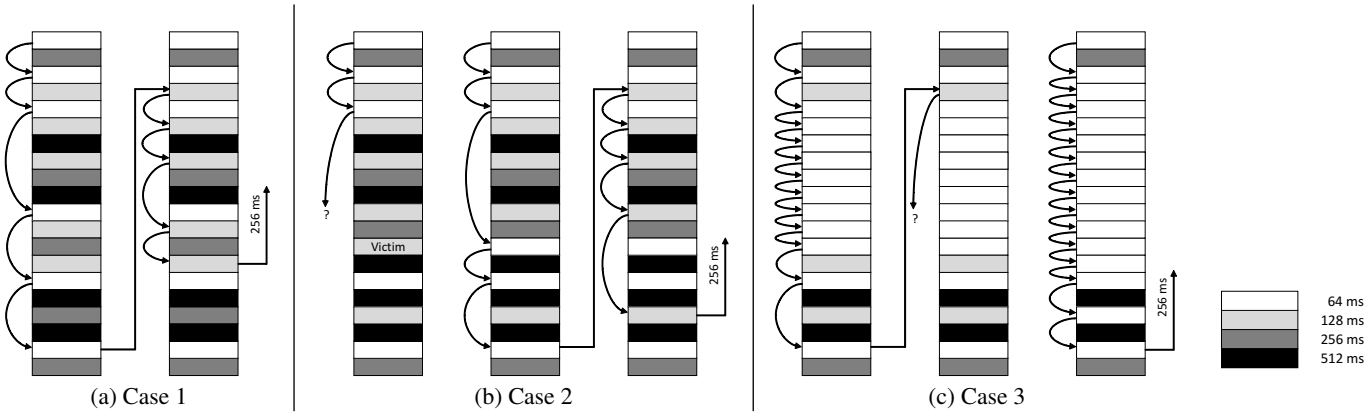
As shown in Fig. 5a, the offset between rows with T_{ret} = 64 *ms*; between the last row with T_{ret} = 64 *ms* and the first row with T_{ret} = 128 *ms*; between rows with T_{ret} = 128 *ms* and so on is less than D. In this case, a linked list can be easily formed.

Case 2: *offset* $> D$

As shown in Fig. 5b, the offset between the third and fourth rows with T_{ret} = 64 *ms* is larger than D. In this case we find a *victim row* whose *offset* $\leq D$ and T_{ret} is larger than the current T_{ret} value being linked, in this case 64 *ms*. We then demote its T_{ret} value to the current T_{ret} value being linked, in this case 64 *ms*. Effectively, we use victim rows as stepping stones to aid the linked list construction. Obviously, the number of victim rows falls with increasing D.

When it is possible to pick a victim row from among multiple eligible rows, we heuristically select the one which maximizes the *offset*, as shown in Fig. 5b. In some cases we might need to find multiple victim rows between a pair of rows. Again, we select victim rows which maximize the *offset* so that the total number of victim rows between the pair of rows is minimized.

If the offset between, say the last row with T_{ret} = 64 *ms* and the



(a) Case 1 (b) Case 2 (c) Case 3

Figure 5: Device bank linked-list generation.

first row with T_{ret} = 128 *ms* is larger than D, then we find a victim row whose *offset* $\leq D$ and T_{ret} is larger than 128 *ms*. We then demote its T_{ret} to the higher of the two endpoints, in this case 128 *ms*.

Case 3: $(offset > D)$ && $(!victim\ row)$

As shown in Fig. 5c, the offset between the first and second rows with T_{ret} = 128 *ms* is larger than D. In addition, we cannot find a victim row as all the intermediate rows have T_{ret} = 64 *ms*. In this case, we demote all rows of the current T_{ret} value being linked to the next lower T_{ret} value. In our example, we demote all rows with T_{ret} = 128 *ms* to 64 *ms*. We then have to redo the linked list for T_{ret} = 64 *ms*, as shown in Fig. 5c. If we now continue to link rows with T_{ret} = 256 *ms*, we will encounter the same issue and those rows will also need to be demoted to 64 *ms*. This case happens when the maximum offset, D is almost equal to the number of retention time bins. In our proposal, this happens when $D = 4$ i.e. $L = 2$.

In our evaluation we show the trade-off between the number of refresh-data bits and refresh reduction.

4. RELATED ISSUES

4.1 Profiling and Configuration

Initialization. At power-on, the DRAM module and memory controller are set by default to the traditional mode of refreshing all DRAM rows at the default maximum rate. The system software or BIOS is responsible for reading the refresh profile linked-list information from disk or non-volatile memory and populating the refresh linked-list held in each DRAM row via a new DRAM command (an extension of the existing MRS commands). This command writes specified refresh-data to the currently activated row.

At the time the system is first commissioned or after any DRAM module is replaced, which can be detected utilizing serial number data in the SPD data, no refresh profile data is available. The system software manages refresh characterization of the module(s) and the construction of the linked-list, as described below. Note that this characterization and linked-list construction has to be done only once.

Profiling and Linked-List Construction. The first step towards exploiting variation is to profile the T_{ret} of each device row. A variety of proposals discussing variation in DRAMs [34, 47, 31, 24] and eDRAMs [29, 32, 49, 15, 4, 5] have used and/or proposed T_{ret} profiling schemes.

Profiling and linked-list construction cannot be done on a tester as the information would be lost soon after. In-situ profiling of the refresh characteristics of the module may not represent the worst-case temperature and voltage, however, the behavior of DRAM cells under different temperature and voltage conditions is reasonably well characterized. These effects and others can be accounted for analytically and with the application of appropriate guard-bands.

Profiling under DPD and VRT. For CLARA, to protect against DPD effects, we use the test patterns in [33]. The profiling temperature is obtained from the on-chip temperature sensor and then the T_{ret} value at 85 °C is obtained using the equation in [4, 14]. We apply a guard band to protect against worst-case voltage. Finally, to protect against VRT effects, we apply a guardband of 4x. This is because experimental data in [33] shows that the maximum change in T_{ret} due to VRT is $\sim$ 4x.

Note that the memory vendor guarantees that all lines have T_{ret} >= 64 *ms* at 85 °C even under worst-case voltage and VRT. Therefore, even if we apply a very large guard band (voltage and VRT) on the measured values, all rows will have a T_{ret} of 64 *ms* but not lower.

Binning. Depending on the precision of the profiling routine, the measured T_{ret} values can take a large number of distinct values. To simplify the hardware implementation, we use only a finite set, S of T_{ret} values. We choose, S ={64 *ms*, 128 *ms*, 256 *ms*, 512 *ms*}. Using a larger set by adding a bin for 1024 *ms* offers only marginally more savings as shown in Sec. 6.1. Note that there are no rows with T_{ret} less than 64 *ms* at 85 °C.

4.2 Tester & Testing Time

A tester checks the functionality of DRAM array bits and peripheral logic at the worst case temperature and voltage. Some repairs, using spare rows and/or columns, are also performed to improve yields. CLARA's overhead in the DRAM array is only about 0.098%. The additional blocks introduced in the peripheral logic are an adder, a multiplexer and a counter. Hence, the testing overheads are minimal.

Note that profiling or linked-list construction is not done on the tester. Also chips which pass are guaranteed to have a T_{ret} >= 64 *ms* at 85 °C even under Data Pattern Dependence (DPD) and Variable Retention Time (VRT) effects.

4.3 Serial and Parallel Refresh

Let M be the number of rows refreshed every command. If the rows are refreshed sequentially (typically when $M \leq 4$) the linked list organization and the refresh logic are as described in the sections above. If rows from multiple subarrays are refreshed in parallel (likely when $M \geq 8$), multiple linked lists can be maintained and traversed. For example, assume the upper and lower half of a bank each refresh 4 lines sequentially. Also assume the upper and lower halves work in parallel to refresh $M = 8$ rows every command. In this case we maintain and traverse two smaller linked lists, one each

CPU Parameters	
Chip	4 core CMP
Core	3.2 GHz, 4 issue Out-of-Order
Instruction L1	32 KB, 2 way, Private, Write through
Data L1	32 KB, 2 way, Private, Write through
L2	256 KB, 8 way, Private, Write back
L3	4 MB, 4 banks, Shared, Write back
Coherence	Snoopy MESI Protocol at L2
Line Size	64 Bytes
Main Memory	32 GB, DDR3L-1600, 25.6 GB/s
CPU Memory Parameters	
Channels	2
Ranks per Channel	2
Device Width	8 bits
Devices per Rank	8
Device Density	8 Gb
Banks per Device	8
Rows per Bank	64 K
Columns per Bank	2 K
Device Timing Parameters	
Device	Micron MT41K1G8 [19]
t_{REFI} (< 85 °C)	7800 ns
t_{REFI} ($85-95$ °C)	3900 ns
t_{RFC}	350 ns
$t_{RCD}-t_{RP}-CL$	11-11-11 DRAM clocks

Table 5: CPU architectural parameters.

for the upper and lower half. The upper and lower halves each will have the logic as in Fig. 4.

4.4 Soft Error Detection and Recovery

According to [44] $\sim$ 50% of DRAM errors are single bit errors. We can efficiently detect these common bit errors in the linked-list values using a parity. Alternatively, stronger error detection and correction could be added to reliability conscious systems with little overhead — even the extreme of triple-redundancy for all linked-list values increases the overhead from 0.098% to 0.294%.

The linked list value is checked for errors in parallel with the next row address generation in Fig. 4. If the check fails during auto-refresh, the refresh logic can notify the MC using a dedicated pin. The memory controller can then set the bank or the entire rank (depending on 'single-bank' or 'all-bank' scheme) to operate in epoch 7 (conventional) and refresh all rows every epoch, until it reloads the correct refresh-data to rebuild the linked list. If the check fails during self-refresh, the refresh logic can switch to epoch 7 and refresh all rows every epoch. It can notify the memory controller immediately or on exit from self-refresh mode.

4.5 Temperature Adaptation

In the extended temperature range ($85-95$ °C), the refresh rate is doubled. In this range, CLARA reduces the epoch length from 64 *ms* to 32 *ms*. The MC and the self-refresh logic initiate the same number of refreshes, as obtained in Sec. 3.3 and Sec. 3.2, respectively, but in half the time. The timer in Fig. 4 increments the epoch and resets the row address every 32 *ms* instead of 64 *ms*.

5. EVALUATION SETUP

5.1 Architectural Parameters

We evaluate CLARA on a simulated 4 core chip multi-processor (CMP). Each core has a 4 issue, out-of-order execution engine running at 3.2 GHz. Each core also has a private instruction L1 cache, a data L1 cache and a unified L2 cache. The cores share a L3 cache, which is divided into 4 banks. The chip employs a snoopy MESI coherence protocol between the L2s. The L3 cache is connected to the off chip main memory.

GPU Parameters	
Number of SMs	30
Threads per SM	1024
Warp Size	32
L1/L2/Cacheline Size	32 KB / 786 KB / 128 B
Main Memory	GDDR5
GPU Memory Parameters	
Channels	6
Ranks per Channel	1
Device Width	32 bits
Devices per Rank	2
Device Density	16 Gb
Banks per Device	16 banks in 4 bank-groups
Rows per Bank	64 K
Row Size	2 KB
Device Timing Parameters	
Device	Hynix H5GQ1H24AFR [1]
t_{REFI}	3900 ns
t_{RFC}	350 ns
$t_{RCD}-t_{RP}-CL$	12-12-12 ns

Table 6: GPU architectural parameters.

The CPU main memory is organized as 2 channels of DDR3L-1600, with a peak bandwidth of 25.6 GB/s. Each channel has 2 ranks and each rank has 8 banks. The total memory capacity is 32 GB. In normal operating conditions (< 85 °C), the memory controller issues a refresh request every 7800 ns (t_{REFI}). In extended temperature range ($85-95$ °C), the memory controller issues a refresh request every 3900 ns (t_{REFI}). Higher temperatures are likely in server environments, 3D stacked organizations, automotive systems etc. The architectural parameters are summarized in Table 5. Detailed DRAM timing parameters were obtained from [19].

We evaluate the impact of CLARA on a throughput processing environment using the GPGPUSim [7] simulator. We simulate a GTX-480-like system that has 30 SMs, each with 1024 threads (warp-size of 32) and a 32 KB L1 cache. We simulate 6 memory partitions, each with a 128KB L2, and a 64-bit DRAM channel comprised of two x32 GDDR5 devices. Each channel has 16 banks organized into 4 bank groups. We assume a futuristic 16 Gb device with 64 K rows and $t_{RFC} = 350$ ns. A GDDR5 device issues a refresh request every 3900 ns (t_{REFI}). The GPU architectural parameters are summarized in Table 6. DRAM timing parameters were obtained from [1].

5.2 Applications and Tools

We evaluate CPU performance using applications from NAS Parallel Benchmarks (NPB) [39], SPEC CPU2006 [26], and HPC Challenge (HPCC) Benchmark [27]. The applications and their problem sizes in parenthesis are as follows: BT (small), CG (workstation), IS (workstation), LU (small), MG (workstation), SP (small), astar (test), bzip2 (test), gromacs (test), h264ref (test), lbm (test), libquantum (train), mcf (test), milc (test), povray (test), DGEMM (1024×1024), STREAM2 (default), PTRANS (4096×4096), and Jacobi (500×500). All applications from NAS parallel benchmarks and HPCC benchmarks (except STREAM2), and Jacobi have 4 threads; while applications from SPEC suite are single threaded.

We evaluate GPU performance using applications from the Rodinia benchmark suite [13]. The applications are as follows: Back Propagation, Breadth First Search, B+ Tree, CFD Solver, Gaussian Elimination, Heart Wall, HotSpot, Kmeans, LavaMD, LU Decomposition, Needleman-Wunsch, Particle Filter and SRAD.

We use the DRAM cell T_{ret} distribution in [24] to generate the T_{ret} values of 128 device banks, each with 1 G (2^{30}) cells. The refresh reduction reported in Sec. 6 is the average across 128 device banks. To estimate CPU system performance we use SESC [42] integrated with DRAMSim2 [43, 21]. To estimate GPU performance we use GPGPUSim [7]. We use R [41] for statistical analysis.

5.3 Comparison with Prior Work

We compare CLARA against the following schemes for auto and self refresh reduction and performance improvement:

Baseline. The conventional refresh scheme where the memory controller issues 8 *K* auto refresh commands every 64 *ms*. Each command refreshes 8 rows, thereby refreshing 64 *K* rows every 64 *ms*.

RAIDR. A recently proposed scheme [34], which bins the T_{ret} of a 'rank-wide row' into one of 3 retention categories i.e., 64 *ms*, 128 *ms* or 256 *ms*. It reduces the number of refresh commands, however, each command refreshes only 1 rank-wide row by using the deprecated 'RAS-only-refresh' mode. RAIDR logic overhead increases with number of retention categories (bins). With 4 bins, most refresh requests ($> 90\%$) require 3 bloom filter checks. Therefore, RAIDR uses 3 bins. We will, however, also compare against a RAIDR implementation which uses 4 bins i.e. 64 *ms*, 128 *ms*, 256 *ms* and 512 *ms*.

Ideal. A refresh scheme where the T_{ret} of each device row can be measured, and each device row can be refreshed, with arbitrary precision.

Among schemes which exploit variation, RAPID [47] and RIO [6] trade off DRAM capacity for refresh reduction. In addition, they require operating system modifications. DTail-R [16] uses the DRAM physical address space to store refresh-data but that refresh-data has to be accessed once every 128 refresh decisions. VRA [40] is similar to CLARA only in the part that it stores additional bits per DRAM row. However, as discussed in Sec. 7, the area and layout issues of VRA makes it unrealistic for current systems. RAIDR [34] is a hardware only scheme that does not require software modification or result in DRAM capacity loss. Hence, we compare against it. REFLEX [10] is similar to RAIDR w.r.t. auto-refresh reduction. Please see Sec. 7 for a detailed discussion. We will do the evaluation in both the normal ($< 85\ ^\circ$C) and the extended ($85 - 95\ ^\circ$C) temperature range.

6. EVALUATION

6.1 Refresh Data Length

Fig. 6 shows the number of refreshes in a bank as a function of number of refresh-data bits per row. The bank has $N = 64\ K$ rows and each row has 16384 bits. The X-axis shows the number of refresh-data bits per row, L which varies from 0 i.e. no refresh-data bits to $log_2 N = 16$ i.e. the maximum number of bits required to store an offset. The Y-axis shows the total number of refreshes in 8 epochs i.e. 512 *ms*.

$L = 0$ corresponds to the baseline and the total number of refreshes is equal to $64\ K \times 8 = 524288$. As we increase L, the number of refreshes drops very quickly. This is because the number of victim rows reduce exponentially. With just 5 bits, the number of refreshes is less than 20% of the baseline. With 10 bits, it is 13.79% of the baseline. As discussed in Sec. 3.5, when $L = 16$ the linked list can be constructed without any victim rows. In this case each row can be refreshed at its required rate and we perform the minimum number of refreshes i.e. 71956, which is 13.72% of the baseline. The ideal scheme requires 40508 refreshes, which is 7.72% of the baseline. From the data we observe that there are no victim rows even with 14 or 15 bits. Therefore, the area overhead of CLARA in the DRAM array is only $14/16384 \times 100 = 0.085\%$. The peripheral logic area overhead is only an epoch counter, an adder and a multiplexer.

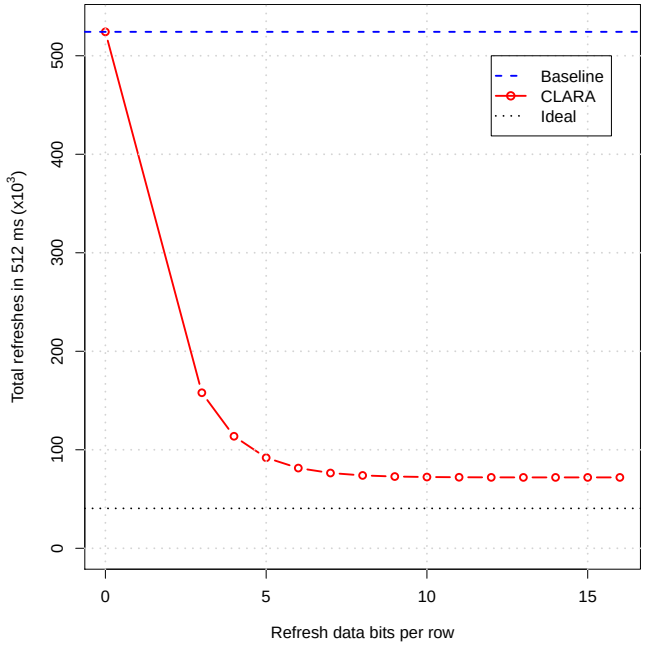


Figure 6: Total refreshes vs refresh data bits per row.

Adding a bin of 1024 *ms* reduces the number of refreshes by an additional 2.5% i.e. to 11.2% of the baseline. These are diminishing returns with additional hardware.

6.2 Auto and Self Refresh

Fig. 7 shows the auto- and self-refresh reduction using different refresh schemes. The X-axis shows two sets of bars, one for the auto-refresh mode and one for the self-refresh mode. Each mode is evaluated at the baseline, RAIDR, CLARA and the ideal refresh schemes. The Y-axis shows the reduction *w.r.t.* the baseline. All

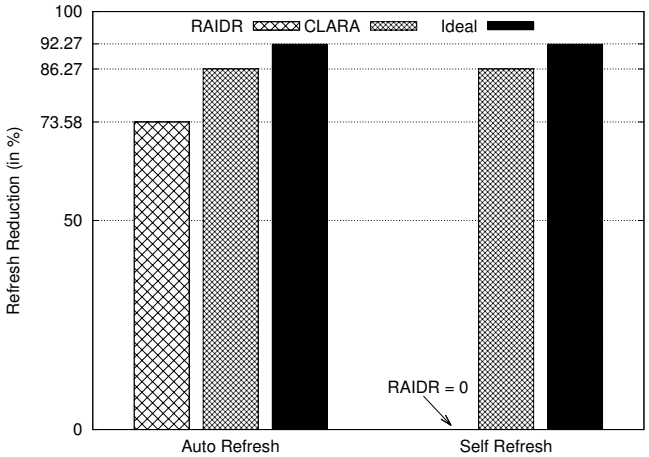


Figure 7: Auto and self refresh reduction comparison.

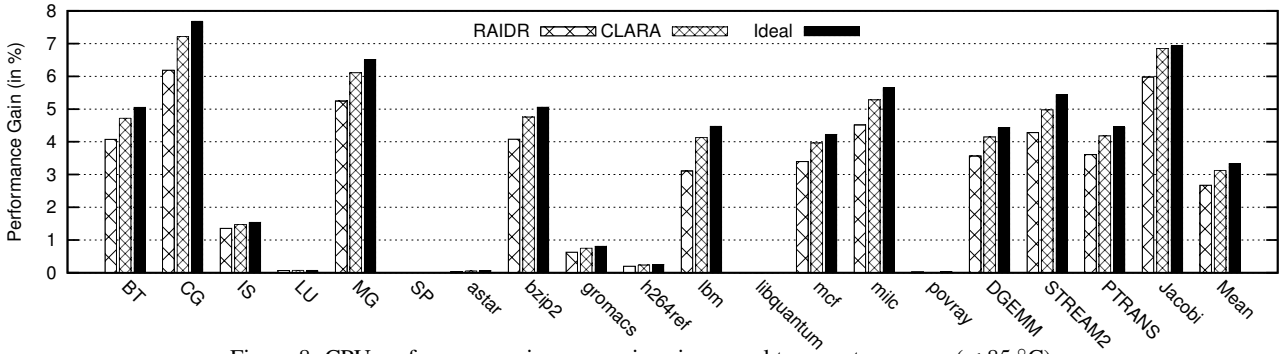


Figure 8: CPU performance gains comparison in normal temperature range (< 85 °C).

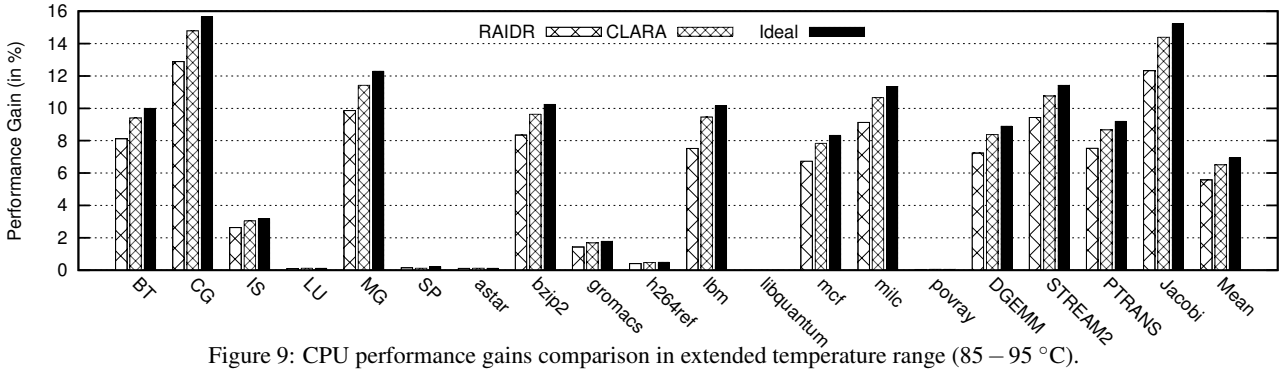


Figure 9: CPU performance gains comparison in extended temperature range ($85 - 95$ °C).

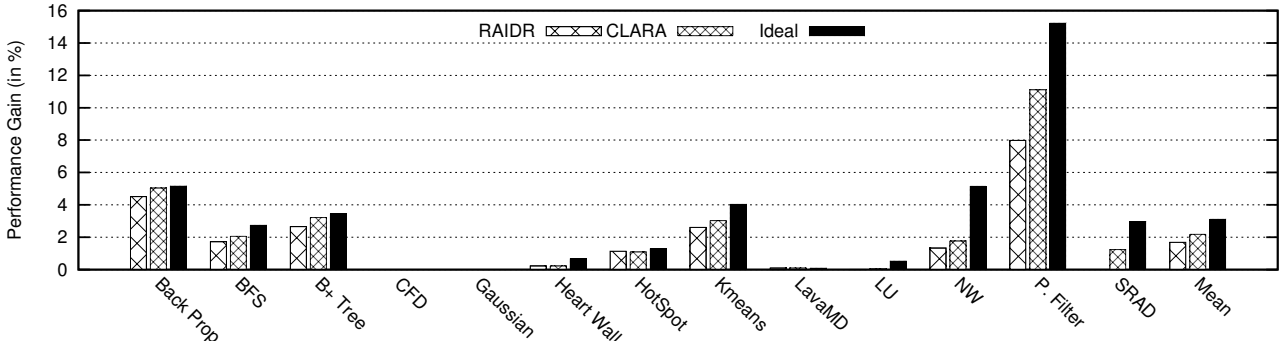


Figure 10: GPU performance gains comparison in extended temperature range ($85 - 95$ °C).

schemes are hardware based and the refresh reduction is application agnostic. In the extended temperature range, the refresh rate doubles for all the schemes and hence, the percentage reduction is the same as in the normal temperature range.

CLARA stores refresh data within the DRAM devices and hence, as explained in Sec. 3.2 and Sec. 3.3, can be used during both auto- and self-refresh modes. On the other hand, RAIDR stores refresh data in the memory controller which can be used only in auto-refresh mode. CLARA's simple hardware allows us to use 4 retention bins as opposed to 3 in RAIDR. In addition, CLARA exploits variation at a finer 'device-row' granularity as opposed to the coarser 'Rank-wide-row' granularity used in RAIDR.

Overall, CLARA achieves a reduction of 86.2% in both auto- and self-refresh while RAIDR reduces auto-refresh by 73.5%. RAIDR with 4 bins reduces auto-refresh by 79.0%. RAIDR does not reduce self-refresh at all. Auto-refresh reduction leads to performance improvements and is quantified in the following section. A reduction of 86.2% in self-refresh reduces DRAM idle power by 44%.

VRT Guard Band. Even with a VRT guard band of 2x and 4x, CLARA achieves a refresh reduction of 77.7% and 56.1% respectively, during both auto- and self-refresh. Using the same guard bands, RAIDR only reduces auto-refresh by 59.7% and 24.7% respectively.

6.3 Performance

Figures 8 and 9 show the CPU performance gain of applications, using different refresh schemes, in the normal and extended temperature range respectively. Figure 10 shows the GPU performance gain in the extended temperature range. In all plots, the X-axis is divided into multiple sets, one for each application and one for the geometric mean. Each application is run on the baseline, RAIDR, CLARA, and the ideal refresh schemes. The Y-axis shows the performance gain *w.r.t.* the baseline. In active mode, a reduction in auto-refresh commands increases DRAM bank availability which in turn improves performance. When the DRAM is in self-refresh, the CPU/GPU is either working from its caches or is also sleeping and hence there is no performance impact.

We observe that CPU applications which are either compute intensive or whose working set fits within the last level cache are not affected by the main memory subsystem and do not show any performance gains. On the other hand, memory intensive applications can exhibit a performance improvement of up to 7.5% in the normal temperature range. As shown in Fig. 8, on average across all 19 applications, CLARA performs better than RAIDR and im-

Exploits	Scheme	Improves			Device Granularity	T_{ret}/Validity Data in Device	No Program and/or OS Modification	# Refreshes Program Independent	No DRAM Capacity Loss
		Auto Refresh	Self Refresh	Performance					
Variation	**CLARA**	✓	✓	✓	✓	✓	✓	✓	✓
	REFLEX [10]	✓	✓	✓	✗	✗	✓	✓	✓
	RAIDR [34]	✓	✗	✓	✗	✗	✓	✓	✓
	RAPID [47]	✓	✓	*	✗	✗	✗	✗	✗
	VRA [40]	✓	✓	✗	✓*	✓	✓	✓	✓
	RIO [6]	✓	✓	✓	✗	✗	✗	✓	✗
	DTail-R [16]	✓	✗	✓	✗	✗	✗	✓	✗
	Wang patent [48]	✗	✗	✗	✗	✓*	✓	✓	✗*
Access Pattern	Smart Refresh [23]	✓	✗	✓	✗	✗	✓	✗	✓
Non-critical/ Invalid Data	Flikker [35]	✓	✓	✗	✗	✓	✗	✗	✓
	SRA [40]	✓	✓	✗	✗	✓	✗	✗	✓
	ESKIMO [28]	✓	✓*	*	✗	✓	✗	✗	✓
	PARIS [6]	✓	✗	✓	✗	✗	✗	✗	✓
	DTail-V [16]	✓	✗	✓	✗	✗	✗	✗	✗
Scheduling	Ref. Pausing [38]	✗	✗	✓	n/a	n/a	✓	n/a	✓
	Elastic Ref. [46]	✗	✗	✓	n/a	n/a	✓	n/a	✓
	DARP&SARP [11]	✗	✗	✓	n/a	n/a	✓	n/a	✓
	CREAM [50]	✗	✗	✓	n/a	n/a	✓	n/a	✓
	Adaptive [37]	✗	✗	✓	n/a	n/a	✓	n/a	✓
	Coordinated [8]	✗	✗	✓	n/a	n/a	✓	n/a	✓

Legend: Yes(✓), No(✗), Not evaluated(*), Not applicable(n/a)

Table 7: CLARA compared with existing DRAM refresh management techniques.

proves performance by 3.1% in the normal temperature range.

In the extended temperature range, the refresh rate and the bank unavailability doubles and hence the performance penalty in the baseline is higher. The same reduction (86.2%) in auto-refresh commands, therefore leads to higher performance gains. As shown in Fig. 9, on average across all 19 applications, CLARA performs better than RAIDR and improves performance by 6.5%.

GPU applications benefit from reducing the refresh frequency, however the performance improvements vary depending on the application (maximum 11% over the baseline with CLARA). Due to high thread-level-parallelism, GPUs are relatively insensitive to small variations in DRAM latency [3], and so the impact of refresh interruptions are most visible in bandwidth-bound workloads such as Kmeans. In all cases however, CLARA outperforms RAIDR as expected.

Energy. In active mode, a performance improvement of 2-6.5%, proportionally reduces system energy by 2-6.5%. In idle-mode, when the CPU/GPU is powered down, we reduce DRAM energy by 44% and hence system energy by 44%.

7. RELATED WORK

A variety of techniques have been proposed for refresh management in DRAMs. These techniques either reduce auto-refresh, reduce self-refresh, improve performance or a combination thereof. For example, techniques which reduce auto-refresh often (not always) result in better performance, while refresh scheduling techniques only result in better performance. These techniques vary in their granularity, application and/or Operation System (OS) support, information storage location, capacity loss and so on.

These techniques exploit either the variation in retention time (similar to CLARA), data access patterns, data criticality/validity or scheduling flexibility. We have classified all techniques into these categories as shown in Table 7. For each technique we tabulate its ability to reduce auto-refresh, reduce self-refresh and improve performance. In addition, we catalog if the technique, works at a DRAM device granularity, stores row T_{ret}/validity data in the DRAM device, does not requires application and/or OS modifications, and does not result in DRAM capacity loss. If the technique reduces auto- and/or self-refresh we note if the reduction is application agnostic. A good overview of existing refresh schemes can also be found in [9].

An ideal refresh scheme should have all the above properties. It should reduce auto- and self-refresh and improve performance. The refresh reduction should be application agnostic. It should not require application and/or OS modification for maximum portability or result in DRAM capacity loss. For maximum benefits, the scheme should be able to exploit T_{ret} variation/validity information at a 'device-row' level instead of a 'Rank-wide row' level. For a scheme to be useful in self-refresh and to reduce storage overhead, the scheme should store T_{ret} variation/validity information in the DRAM device.

Variation. Amongst the schemes which exploit T_{ret} variation, RAPID [47] and RIO [6] trade off DRAM capacity for refresh reduction (auto and self). Both schemes operate on an OS-page granularity and remove the leakiest pages from the pool of physical pages which can be used by the OS. DTail-R [16] stores T_{ret} data in a reserved space in the DRAM physical address space. The T_{ret} data has to be accessed every 128 refresh decisions but was shown to improve auto-refresh and performance. RAIDR [34] stores T_{ret} data in bloom filters and can improve auto-refresh and performance but not self-refresh. The only similarity between CLARA and VRA [40] is that both store T_{ret} information per row in the DRAM device. VRA needs a comparator per row as well. While CLARA stores the bits within the DRAM array, the VRA design stores the additional bits in registers (flip-flops). The area overhead for VRA is more than 20% for a 8 bit register and comparator per row.

REFLEX [10] reduces auto-refresh by introducing a new 'dummy refresh' command which skips refreshes and increments the internal counter. It can also reduce self-refresh by refreshing weak rows before entering self-refresh(SR) mode. However, the technique works only if the time spent in SR mode is less than 64 *ms*. After 64 *ms* it will have to wake up the memory controller, exit SR,

refresh the appropriate rows and go back to SR. For systems with long idle times (mobile environments) the periodic wake ups is a concern. Furthermore, REFLEX works on a 'rank-wide-row' granularity and not 'device-row' granularity. Also, the storage overhead of REFLEX is 2 *KB* per rank in the memory controller as opposed to 6 bytes for the entire memory in CLARA.

In the patent[48], the inventor proposed a circular linked list design to store the refresh sequence of memory rows. Rows (or blocks containing rows) with shorter retention appear in the linked list more often, thus causing them to be refreshed more frequently. The described scheme targets improving yield in the presence of weak-retention cells, thus the total number of refresh commands issued is kept the same and hence the scheme neither reduces refresh (auto or self) nor improves performance. One could imagine applying this scheme to reducing refresh rate, however. The described scheme exploits variation at a 'Rank-wide block' granularity as opposed to CLARA's 'device row' granularity. The patent describes storing the refresh sequence in an external chip or in separate DRAM rows unlike CLARA, which stores the refresh sequence within the DRAM row which is implicitly available on a row refresh.

Access Pattern. Smart Refresh [23] exploits memory access pattern to prevent a recently accessed line from begin refreshed. It reduces auto-refresh and improves performance but does not reduce self-refresh. It does not require any SW support. However, the refresh reduction is contingent on the application footprint and access pattern.

Criticality/Validity. Techniques which skip refreshing invalid OS pages or reduce refresh rate to error-tolerant / non-critical data [35] require operating system support for conveying page allocation/deallocation events and language / runtime support for application annotation. Flikker [35] and SRA [40] reduce auto- and self-refresh, while PARIS [6] and DTail-V [16] reduce auto-refresh and improve performance. However, the benefits are highly application dependent. In the worst case when the application is either not annotated, has little or no error-tolerant data, or has a large memory footprint, these techniques offer little refresh savings or performance improvement.

Scheduling. Schemes which schedule refreshes improve performance but do not reduce auto- and self-refresh. These are part of the memory controller scheduler and do not require any software changes. These are orthogonal to our scheme and can be used in conjunction with CLARA. For example, in epoch 7, when all rows have to be refreshed, CLARA can benefit from refresh scheduling.

Embedded DRAM (eDRAM) is increasingly being adopted into mainstream processors [30, 45]. eDRAM capacity is about 3 orders of magnitude smaller than main memory. However, the T_{ret} of eDRAMs is also 3 orders of magnitude smaller than that of DRAMs [5]. Therefore, refresh operations are a concern for eDRAMs as they are for DRAMs. Various techniques for refresh reduction in eDRAMs have been proposed as well. These techniques exploit variation [4], dead line prediction [12], access pattern [5] and ECC [22, 49].

8. CONCLUSION

In this paper we presented CLARA, a technique which reduces auto-refresh and improves performance in active mode, as well as reduces self-refresh in idle mode. It does so without dicarding any rows in the DRAM. CLARA is also the first technique to exploit variation in retention time at a device(chip) granularity as opposed to rank granularity in prior work. It is a hardware only solution and does not require any software (application or OS) modifications. CLARA is extremely frugal in its hardware requirements. The area overhead of CLARA in the DRAM is <0.1%, and negligible in the memory conttroller.

CLARA reduces auto- and self-refresh by 86.2%, and is independent of the workload. Auto-refresh reduction improves average CPU performance by 3.1% and 6.5% in the normal and extended temperature range, respectively. It improves average GPU performance by 2.1% in the extended temperature range. Reduction in self-refresh improves DRAM idle power by 44%.

Acknowledgment

This research was developed, in part, with funding from the United States Department of Energy and, in part, with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions, and/or findings contained in this article/presentation are those of the author/presenter and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

9. REFERENCES

[1] 1 Gb GDDR5 SGRAM H5GQ1H24AFR. [Online]. Available: http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR(Rev1.0).pdf

[2] Micron DDR4 SDRAM. [Online]. Available: http://www.micron.com/products/dram/ddr4-sdram

[3] N. Agarwal *et al.*, "Page Placement Strategies for GPUs within Heterogeneous Memory Systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[4] A. Agrawal, A. Ansari, and J. Torrellas, "Mosaic: Exploiting the Spatial Locality of Process Variation to Reduce Refresh Energy in On-Chip eDRAM Modules," in *International Symposium on High Performance Computer Architecture*, Feb. 2014.

[5] A. Agrawal, P. Jain, A. Ansari, and J. Torrellas, "Refrint: Intelligent Refresh to Minimize Power in On-Chip Multiprocessor Cache Hierarchies," in *International Symposium on High Performance Computer Architecture*, Feb. 2013.

[6] S. Baek, S. Cho, and R. Melhem, "Refresh Now and Then," *IEEE Transactions on Computers*, Dec. 2014.

[7] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2009.

[8] I. Bhati, Z. Chishti, and B. Jacob, "Coordinated Refresh: Energy Efficient Techniques for DRAM Refresh Scheduling," in *IEEE International Symposium on Low Power Electronics and Design*, Sep. 2013.

[9] I. Bhati, M.-T. Chang, Z. Chishti, S.-L. Lu, and B. Jacob, "DRAM Refresh Mechanisms, Penalties, and Trade-Offs," in *IEEE Transactions on Computers*, 2015.

[10] I. Bhati, Z. Chishti, S.-L. Lu, and B. Jacob, "Flexible Auto-Refresh: Enabling Scalable and Energy-Efficient DRAM Refresh Reductions," in *International Symposium on Computer Architecture*, Jun. 2015.

[11] K.-W. Chang *et al.*, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *IEEE International Symposium on High Performance Computer Architecture*, Feb. 2014.

[12] M.-T. Chang, P. Rosenfeld, S.-L. Lu, and B. Jacob, "Technology Comparison for Large Last-Level Caches (L3Cs): Low-Leakage SRAM, Low Write-Energy STT-RAM, and Refresh-Optimized eDRAM," in *IEEE International Symposium on High Performance Computer Architecture*, Feb. 2013.

[13] S. Che *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous computing," in *IEEE International Symposium on Workload Characterization*, Oct. 2009.

[14] J.-H. Choi, K.-S. Noh, and Y.-H. Seo, "Methods of operating DRAM devices having adjustable internal refresh cycles that vary in response to on-chip temperature changes," Patent US 8 218 137, Jul., 2012.

[15] K. C. Chun, W. Zhang, P. Jain, and C. Kim, "A 700 MHz 2T1C Embedded DRAM Macro in a Generic Logic Process with No Boosted Supplies," in *International Solid-State Circuits Conference*, Feb. 2011.

[16] Z. Cui, S. A. McKee, Z. Zha, Y. Bao, and M. Chen, "DTail: A Flexible Approach to DRAM Refresh Management," in *ACM International Conference on Supercomputing*, Jun. 2014.

[17] (2009) DDR2 SDRAM Standard. [Online]. Available: http://www.jedec.org/standards-documents/docs/jesd-79-2e

[18] (2012) DDR3 SDRAM Standard. [Online]. Available: http://www.jedec.org/standards-documents/docs/jesd-79-3d

[19] DDR3L SDRAM MT41K1G8. [Online]. Available: http://www.micron.com/~/media/documents/products/data-sheet/dram/ddr3/8gb_ddr3l.pdf

[20] (2013) DDR4 SDRAM Standard. [Online]. Available: http://www.jedec.org/standards-documents/results/jesd79-4%20ddr4

[21] DRAMSim2. [Online]. Available: http://www.eng.umd.edu/~blj/dramsim/

[22] P. Emma, W. Reohr, and M. Meterelliyoz, "Rethinking Refresh: Increasing Availability and Reducing Power in DRAM for Cache Applications," *IEEE Micro*, Nov.-Dec. 2008.

[23] 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 *IEEE International Symposium on Microarchitecture*, Dec. 2007.

[24] T. Hamamoto, S. Sugiura, and S. Sawada, "On the Retention Time Distribution of Dynamic Random Access Memory (DRAM)," *IEEE Transactions on Electron Devices*, Jun. 1998.

[25] (2013) High Bandwidth Memory (HBM) Standard. [Online]. Available: http://www.jedec.org/standards-documents/results/jesd235

[26] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Computer Architecture News*, Sep. 2006.

[27] HPC Challenge Benchmark. [Online]. Available: http://icl.cs.utk.edu/hpcc/index.html

[28] C. Isen and L. John, "ESKIMO: Energy savings using Semantic Knowledge of Inconsequential Memory Occupancy for DRAM subsystem," in *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2009.

[29] S. S. Iyer *et al.*, "Embedded DRAM: Technology platform for the Blue Gene/L chip," *IBM Journal of Research and Development*, Mar. 2005.

[30] R. Kalla, "POWER7: IBM's Next Generation POWER Microprocessor," in *Hot Chips: A Symposium on High Performance Chips*, Aug. 2009.

[31] K. Kim and J. Lee, "A New Investigation of Data Retention Time in Truly Nanoscaled DRAMs," *IEEE Electron Device Letters*, Aug. 2009.

[32] W. Kong, P. Parries, G. Wang, and S. Iyer, "Analysis of Retention Time Distribution of Embedded DRAM - A New Method to Characterize Across-Chip Threshold Voltage Variation," in *IEEE International Test Conference*, Oct. 2008.

[33] 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 *International Symposium on Computer Architecture*, Jun. 2013.

[34] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *International Symposium on Computer Architecture*, Jun. 2012.

[35] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM Refresh-power through Critical Data Partitioning," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2011.

[36] (2013) Low Power DDR3 SDRAM Standard. [Online]. Available: http://www.jedec.org/standards-documents/results/jesd209-3

[37] J. Mukundan *et al.*, "Understanding and Mitigating Refresh Overheads in High-density DDR4 DRAM Systems," in *International Symposium on Computer Architecture*, Jun. 2013.

[38] P. Nair, C.-C. Chou, and M. Qureshi, "A Case for Refresh Pausing in DRAM Memory Systems," in *IEEE International Symposium on High Performance Computer Architecture*, Feb. 2013.

[39] NAS Parallel Benchmarks. [Online]. Available: http://www.nas.nasa.gov/publications/npb.html

[40] T. Ohsawa, K. Kai, and K. Murakami, "Optimizing the DRAM Refresh Count for Merged DRAM/Logic LSIs," in *International Symposium on Low Power Electronics and Design*, Aug. 1998.

[41] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, 2014. [Online]. Available: http://www.R-project.org/

[42] J. Renau *et al.* (2005, Jan.) SESC simulator. [Online]. Available: http://sesc.sourceforge.net

[43] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *Computer Architecture Letters*, 2011.

[44] V. Sridharan and D. Liberty, "A Study of DRAM Failures in the Field," in *International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2012.

[45] J. Stuecheli, "Next Generation POWER microprocessor," in *Hot Chips: A Symposium on High Performance Chips*, Aug. 2013.

[46] J. Stuecheli, D. Kaseridis, H. C.Hunter, and L. K. John, "Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory," in *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2010.

[47] R. K. Venkatesan, S. Herr, and E. Rotenberg, "Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM," in *International Symposium on High Performance Computer Architecture*, Feb. 2006.

[48] D. Wang, "DRAM Refresh Method and System," Patent US 8 711 647, Apr., 2014.

[49] C. Wilkerson *et al.*, "Reducing Cache Power with Low-Cost, Multi-bit Error-Correcting Codes," in *International Symposium on Computer Architecture*, Jun. 2010.

[50] T. Zhang, M. Poremba, C. Xu, G. Sun, and Y. Xie, "CREAM: A Concurrent-Refresh-Aware DRAM Memory Architecture," in *IEEE International Symposium on High Performance Computer Architecture*, Feb. 2014.