Relaxing DRAM Refresh Rate through Access Pattern Scheduling: A Case Study on Stencil-based Algorithms

Relaxing DRAM Refresh Rate through Access Pattern Scheduling: A Case Study on Stencil-based Algorithms

Konstantinos Tovletoglou
EEECS, Queen's University Belfast
Email: ktovletoglou01@qub.ac.uk

Dimitrios S. Nikolopoulos
EEECS, Queen's University Belfast
Email: d.nikolopoulos@qub.ac.uk

Georgios Karakonstantis
EEECS, Queen's University Belfast
Email: g.karakonstantis@qub.ac.uk

***Abstract*—The main memory in today's systems is based on DRAMs, which may offer low cost and high density storage for large amounts of data but it comes with a main drawback; DRAM cells need to be refreshed frequently for retaining the stored data. The refresh rate in modern DRAMs is set based on the worst-case retention time without considering access statistics, thereby resulting in very frequent refresh operations. Such high refresh rate leads eventually to large power and performance overheads, which are increasing with higher DRAM densities. However, such high refresh rates may not even required due to extremely low probability of the actual occurrence of the assumed worst-case scenarios, or due to the implicit refresh operation that occur during every memory access, a feature that has not been yet been studied in depth. In this paper, we enhance the state-of-the-art by systematically exploiting the implicit refresh of memory access for relaxing the refresh rate, while minimizing the resulting memory errors. This is achieved by modifying the algorithmic parameters that influence the access patterns such that all stored data are being touched within a target time interval that is necessary for meeting a target error rate. The proposed method is applied to stencil-based algorithms which represent a wide class of algorithms used in numerical analysis, image processing and cellular automata applications. The efficacy of the proposed method is demonstrated on an off-the-shelf server running a fully fledged Linux OS and results show that it is even possible to completely disable DRAM refresh with minor quality loss.**

I. Introduction

The ever increasing need for higher memory capacity is driving the aggressive scaling of Dynamic Random-Access Memory (DRAM), which is an essential component of all computing systems. However, aggressive DRAM scaling is hampered by the need of periodic refresh operations to retain the stored data, the frequency of which is conventionally being determined by the worst-case retention time of the most leaky cell. Such an approach might help to achieve error free storage, but its viability is in doubt due to the large waste of power and throughput that may incur reaching up-to 25-40% and 15-30% respectively, in future 32Gb-64Gb densities [1].

To address such an alarming challenge, many recent studies have shown that the retention time of DRAM cells varies a lot and most of the cells do not require as frequent refresh as the conventional paradigm dictates [2]. Based on such an observation recent approaches [1], [3] proposed to group rows into different retention bins and apply a high refresh rate only for rows of low retention times. However, such multirate-refresh techniques not only require costly and intrusive hardware modifications but also they do not take into account the fact that in practice the retention time of cells changes over time [4], thus rendering error-free storage impossible. A limited number of studies [5], [6] have also attempted to leverage the inherent error-resilience of various applications to relax DRAM refresh rates. However, these studies neglect to identify and make use of the inherent ability of applications to refresh memory by accessing their own data, a property that we call *Refresh-by-Access* (*RefA*). We set the exploitation of *RefA* as the main target of this paper.

We observe that in current DRRx technologies, the refresh operation and memory accesses are mutually exclusive to each other. This means that no row/chip within a rank are allowed to be accessed while the specific rank is being refreshed. This does not only cause a performance penalty, but also prevents the exploitation of the implicit refresh incurred with each memory read, which can in turn help the system relax the conventional *Auto-Refresh* (*AR*) process. Recent works have tried to exploit row access locality [7] or read operations [8] for relaxing refresh. Such methods have demonstrated the potential benefits of exploiting *RefA* but have been impractical and evaluated only in simulation, since they require intrusive modifications of the memory controller.

In this paper, we aim at exploiting systematically the *RefA* property for facilitating aggressive refresh relaxation. This is achieved by developing non-intrusive methods based on the intelligent selection of application parameters that lead to an adequate frequency of memory accesses while satisfying performance and quality constraints. The proposed methods are applied on popular stencil-based algorithms and are evaluated on a real server-grade system stack. Our contributions are summarized as follows:

- We introduce a non-intrusive technique that facilitates *RefA* to a degree that is adequate for touching all stored data within time intervals that are less than a target maximum cell retention time. This is achieved by selecting the appropriate application parameters while considering the performance overhead and quality loss that may be incurred.
- We realize our approach on a commodity server with a complete system stack and show for the first time the benefits of the *RefA* on a real system capturing the time-dependent system and application data interactions under relaxed DRAM refresh rates. The stack allows non-disruptive operation of the whole system under relaxed DRAM refresh rates, which enable us to evaluate the proposed method.

- We apply the proposed approach on the Pochoir stencil compiler to expose parameters and allow us to control the scheduling of memory accesses and exploit the *RefA*.
- We evaluate the efficacy of our approach on the developed experimental platform using with a variety of stencil-based algorithms that are commonly used in numerical analysis, image processing and cellular automata.

Our findings demonstrate that it is possible to extend the refresh rate of *AR* by orders of magnitude and even omit it with limited quality loss and performance overheads.

The rest of the paper is organized as follows. Section II describes the typical DRAM organization and refresh operation, while it discusses the related work and motivates the proposed work. Section III presents the proposed approach, while Section IV discusses its implementation. Section V describes the executed benchmarks and presents the evaluation results. Finally, conclusions are drawn in Section VI.

II. DRAM Background

In this section, we briefly describe the DRAM organization and the refresh operation, along with the state-of-the-art.

A. DRAM organization

A main memory system based on DRAMs is organized hierarchically into channels, ranks, banks, rows and columns, as shown in Fig. 1. Each DRAM module (referred to as DIMM) usually has two ranks consisting of a number of two-dimensional arrays of DRAM cells, the so called banks. Each DRAM cell is a storing element of DRAM and consists of a capacitor and an access transistor. Each access transistor connects its associated capacitor to a wire called a bitline and is controlled by a wire called wordline. Cells sharing a wordline form a row. Before a row can be activated, all bitlines in the bank must be precharged. The row's wordline is enabled by connecting all capacitors in that row to their respective bitlines. This causes charge to flow from the capacitor to the bitline. Finally, the sense amplifier connected to that bitline detects the voltage change and amplifies it, driving the bitline fully either to the power rail or to zero voltage.

B. Retention Time and Refresh Operation

The simple structure of the DRAM array and of each DRAM cell may allow high storage density, however is not capable of retaining the stored charge for a long period due to the inherent transistor's leakage current. Such leakage can eventually discharge the cell, manifesting a bit-flip. The duration that the cell can correctly retain its state (i.e. '0' or '1') without eventually experiencing any bit-flip is called *retention time*.

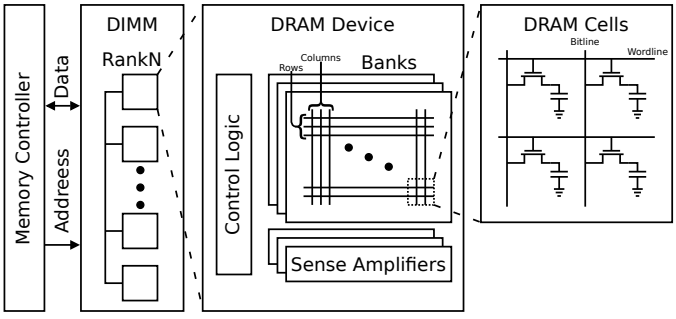


Fig. 1: DRAM memory system organization

To avoid any error induced by the limited retention time, modern day DRAMs employ an *Auto-Refresh* mechanism that periodically recharges each cell in the DIMM by simply bringing the data from a row into the sense amplifiers and restoring them back in the row. To achieve this, the memory controller issues a refresh command every t_{REFI} cycles, at which point all DRAM banks simultaneously refresh a number of rows making the rank unavailable for t_{RFC} cycles.

Currently, the refresh period T_{REFW}, i.e. the interval within which all cells of the DIMM must be refreshed, is set according to the worst case retention time of all cells. In fact all DDRx technologies adopt today a T_{REFW} of 64 ms under nominal environmental conditions or 32 ms in case of temperatures higher than 85°C.

Such a refresh period leads to considerable power and performance overheads, which are expected to worsen as the DRAM density increases [2]. In fact, the duration of the refresh operation increases linearly with each new DRAM generation, so the memory is expected to spend nearly one quarter of the time refreshing and one third of the power consumption for the refresh at 64Gb DRAM density [2].

C. State-of-the-Art

In an attempt to address the refresh related overheads, recent studies have shown that the retention time of cells varies considerably across and within a DRAM chip. Typically, only a very small number of cells needs to be refreshed once every $T_{REFW} = 64\ ms$ [9], [10], [2], [11].

The so called multirate refresh techniques exploit such non-uniformity in retention time of DRAM cells to reduce the frequency of DRAM refresh. Such schemes [1], [3], [12], [13], [14], [15], [16] group rows into different bins based on the retention time profiling and apply a higher refresh rate only for rows belonging to the lower retention time bin. However, such approaches are highly intrusive since they assume fine grain control of the refresh rate, i.e. at the level of the row and have recently been deemed impractical since they neglect the fact that the retention time of each cell can change at runtime [17], [4].

Some recent works have tried to relax the refresh rate and address the resulting errors either by correcting them through traditional error-correcting codes [4] or by allowing them to happen and exploiting the error-resiliency of application for reducing their impact on quality [5], [6]. Although such techniques are interesting they have not yet systematically exploited the inherent refresh that takes place during each memory access which is the primary target of our proposed approach. The works in [8], [7] tried to exploit the implicit refreshes through DRAM accesses. However they require substantial changes in the DRAM controller. This makes them impractical since controllers are currently providing restricted access even to basic parameters (i.e. refresh-rate). Due to the

intrusive required modifications such works have also been evaluated only on simulators and can not yet capture the real efficacy of the refresh by access mechanism. Refrint [18] is targeting eDRAM caches to optimize scheduling of writeback to memory based on the tracking of accesses, it is only feasible in caches as the size of caches is much smaller than that of main memory.

In this paper, we aim at systematically exploiting the *RefA* property, with no intrusive hardware modifications, and evaluating its efficacy on a commodity server with a complete system stack as we explain later.

III. Proposed Approach

Every DRAM access naturally opens the accessed row and consequently restores the leaked charge in the capacitor of DRAM cells, thus incurring an implicit refresh operation. *Refresh-by-Access* can be exploited to significantly relax the *AR*, while restricting the number of manifested errors.

To understand the proposed concept, let us consider a simple scenario as depicted in Fig. 2 and assume that a running application triggers N_r *Memory Accesses* (*MemA*) to the r address in memory so that:

$$MemA_r = \{MemA_{0,r}, ..., MemA_{N_r,r}\} \quad (1)$$

Clearly, the intervals between consecutive accesses to the same address can be calculated as:

$$\Delta t_{i,r} = t_{MemA_{i+1,r}} - t_{MemA_{i,r}} \quad (2)$$

If the maximum of all $\Delta t_{i,r}$ is smaller than a target retention time T_{target} of the DRAM cells, i.e.

$$\max(\Delta t_{i,r}) \leq T_{target} \quad (3)$$

then all cells would be implicitly refreshed though the memory accesses. In this case, we can relax the conventional *AR* up-to T_{target} while maintaining the actual *Bit-Error Rate* (*BER*) low, since no cell that has up to T_{target} retention time will fail. In fact, the BER will be bounded to a value lower than when we adopt *AR* of T_{target}

$$BER(max(\Delta t_{i,r})) \leq BER_{target} \quad (4)$$

To demonstrate the potential of our approach, we have considered the following proof-of-concept. We used known test patterns [19] for characterizing the BER of 8GB DIMMs by aggressively relaxing the *AR* from the conservative 64 ms to 1 sec and up to 30 sec. The resulted *Cumulative Distribution Function* (*CDF*) is depicted in Fig. 3.

For showing the efficacy of our scheme, we turn off the *AR* and test the resulting manifested errors of an artificial benchmark that exploits *RefA*. The benchmark issues memory accesses required for touching all the stored data iteratively within intervals of 10sec and 20sec. Essentially, we have ensured that $\max(\Delta t_{i,r}) = \{10,\ 20\}$ sec in each case.

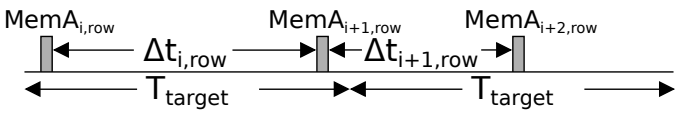


Fig. 2: Graphical representation of our approach where all time intervals between consecutive memory accesses are being kept lower than a target time interval (target retention time).

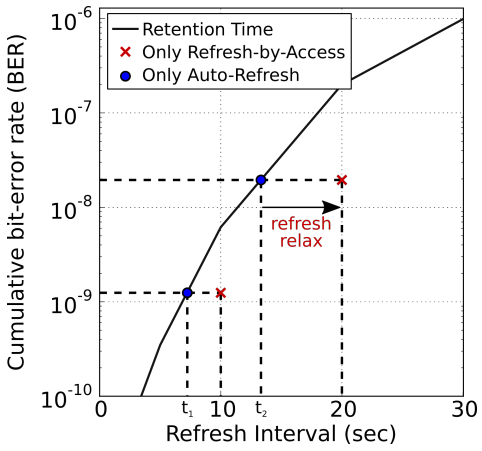


Fig. 3: Cumulative distribution function of bit-errors for relaxed refresh rate. The impact of *RefA* is also being showcased by showing the observed BER of a benchmark.

Interestingly, we can observe that by ensuring that all memory access intervals are bound to a value, then the resulting BER is equal to the one achieved by using *AR* with a more frequent refresh. In particular, the first case resulted in a BER of 10^{-9} that is equal to the one achieved with *AR* of 8.6 sec while the second case resulted in a BER of $2 * 10^{-8}$ that is equivalent to AR of 12 sec.

Our suggestion is that if we schedule the memory accesses incurred by an application in such a way that all rows are iteratively touched within a target refresh interval (that is adequate for keeping the BER at acceptable levels), then the refresh rate of AR can be aggressively relaxed, in our case be omitted. The challenge that we need to address is how could we achieve the above target, which brings us to the details of the proposed methodology that is described next.

IV. Methodology

The main attribute that we need to control for relaxation of the *AR* operations is memory accesses (*MemA*), which are application-dependent. By setting suitably such parameters, we can schedule the memory accesses in such way that we eventually meet the required conditions of Equation 3.

Fig. 4 shows our methodology to achieve this. Initially, we are selecting a set of values for the application parameters $p = \{p_1,\ p_2, ...\}$, such as data structure size, scheduling of the

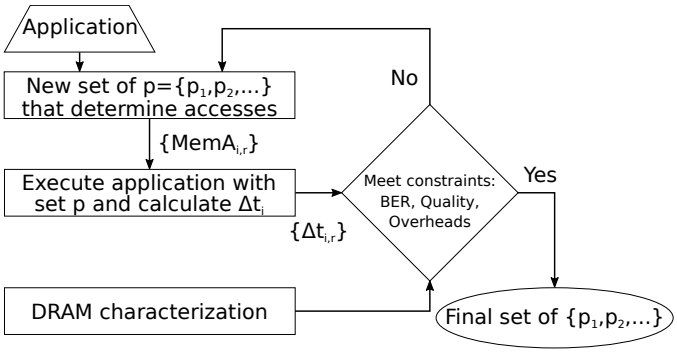


Fig. 4: Block diagram of the steps to explore the design space of an application.

accesses or other parameters that are determining the accesses of the application $MemA_{i,r}$. We are executing the application with the selected parameter set. At the same time we are calculating the $\Delta t_{i,r} : \forall i, r$ of the run, the overheads that are introduced for this set of parameters or any quality metric of interest.

Based on the DRAM characterization, we can calculate the $BER(max(\Delta t_{i,r}))$. If the achieved BER is smaller than a selected BER_{target} and all the other constraints that the designer has specified (e.g. limited performance overhead) are satisfied then we conclude that this set of parameters is acceptable. In case that one of the constraints is violated, we iterate the process with a new set of parameters until all constraints meet.

Such a methodology can be well applied to iterative algorithms, as they regularly access the same data. In this work, we are going to focus on stencil-based algorithms to demonstrate our method.

V. Case Study: Stencil-based Algorithms

Stencil-based algorithms are a class of iterative kernels. In each *sweep*, the stencil updates all the elements of a n-dimensional *grid* using neighboring elements in a fixed pattern, called *stencil*.

We are using Pochoir stencil compiler [20], which is built on top of Cilk Plus multithread extension and uses trapezoidal decompositions, which utilize the cache efficiently. The decomposition is breaking down the problem into smaller *tasks* that are responsible for a part of the *grid* and the associated *stencil* for a number of *sweeps*. It decomposes the sweep domain when the number of sweeps is greater than a threshold and recursively process the lower sub-trapezoids before the upper ones or the base domain. We are modifying the compiler so that it schedules the memory accesses to facilitate *RefA*.

For our method to work, we need to control the interval between consecutive memory accesses. For this reason, we are introducing barriers, we call those *borders*, every $border_{height}$ *sweeps*. *Borders* break down the sweep domain on the same *sweep* for all the *grid*, as shown on Fig. 5 with the dashed line. The *border* essentially does not allow to start any consequent *sweep* forcing all *tasks* in the current $border_{height}$ to complete. We can measure the time intervals between consecutive

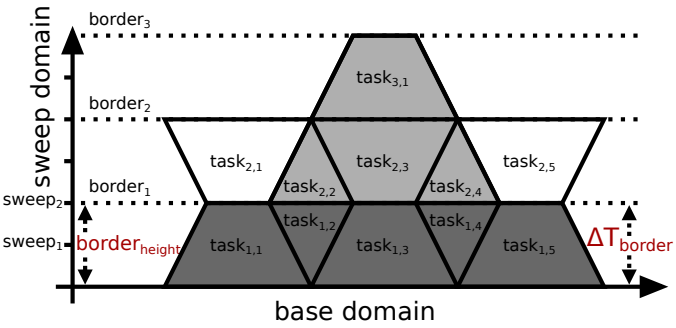


Fig. 5: Trapezoidal decomposition of 1-D problem. All the tasks below $border_1$ must be completed in order to continue to next *sweep*.

completions of *borders*, we call this interval Δt_{border}. All elements will be accessed at least once during Δt_{border}, we can conclude that each $\Delta t_{i,r}$ will be bound to be less than Δt_{border}. Fig. 5 shows also the *Original* implementation that would opportunistically progress as many *sweeps* as possible, specifically after finishing $task_{1,1}$, $task_{1,2}$ and $task_{1,3}$, it will compute all the *tasks* up to the $task_{3,1}$, highlighted in gray.

We are setting $border_{height}$ as a parameter for controlling the Δt_{border}. Lowering the $border_{height}$ will reduce the required computation of each *task* also the Δt_{border}, while increasing the number of *tasks*. The $border_{height}$ must be carefully selected so that the BER resulted from the Δt_{border} will be kept under the set threshold, BER_{target}. Furthermore, tasks should not be broken down to very small ones as there is a considerable performance overhead caused by the function calls and by breaking the cache efficiency.

VI. Evaluation

In this section, we describe our experimental setup and analyze our results.

A. Applications

To evaluate our approach, we have applied it to Pochoir compiler and used it for a range of stencil-based algorithms, that have different number of dimensions, grid size, data size of each element and complexity of the stencil. Specifically, we have experimented with the algorithms of Heat Dissipation (*Heat*) [21], Conways's Game of Life (*Life*) [21] and an artificial benchmark (*Artf*), the characteristics of which are shown in Table 1. The *Artf* is designed as a 3×3 kernel on 2-dimensional grid which in each *sweep* applies binary arithmetic and shifting between different elements so that each bit-error will persist during the execution and be manifested once at the end result.

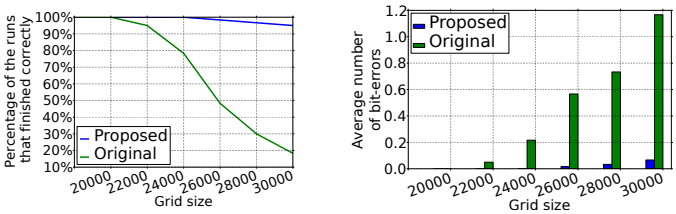
B. Experimental Setup

One of the aims of our work is to evaluate the efficacy of our method on a real system in order to capture the time dependent system and application memory access interactions, as opposed to existing works that were evaluated on simulators. To this end, we have built an experimental platform based on a dual-socket commodity server. Each socket hosts an Intel®Xeon E5-2650 (Sandy Bridge) processor featuring an integrated memory controller (iMC) to control the DRAM devices attached to the socket, specifically four 8 GB DDR3 DIMMs at 1600 MHz. The iMC exposes a set of configuration registers [22] to enable or disable refresh for the entire DRAM.

Following the fact that each iMC controls a single memory domain of the socket, DRAMs attached to different CPU

Benchmark	Grid Size		Memory (GB)		Execution Time	
	min	max	min	max	min	max
Heat-2D	18000	30000	9.3	24	38 s	96 s
Heat-3D	780	1020	7.4	16.6	56 s	263 s
Life-2D	18000	30000	1.5	4	46 s	156 s
Artf-2D	18000	30000	6.2	16	37 s	98 s

Table 1: Benchmarks



(a) Percentage of the runs that completed correctly

(b) Average number of errors per run

Fig. 6: Artf benchmark

sockets can have *AR* enabled or disabled separately. In our dual-socket system, the first memory domain is deemed reliable with the nominal *AR* applied, in which the Operating System is allocated, while the second one is configured with no refresh. In this way, we can safe guard the kernel data that cannot tolerate errors in the reliable domain for avoiding any catastrophic failures and run the application on the second memory domain.

C. Quality Metric

The algorithms, that we are experimenting with are not resilient to errors. Even one error can have catastrophic results as it may propagate to neighboring elements. So the quality metric that we measure is the percentage of runs that completed correctly. For further characterization and to be able to measure the number of bit-errors, we are using the *Artf*.

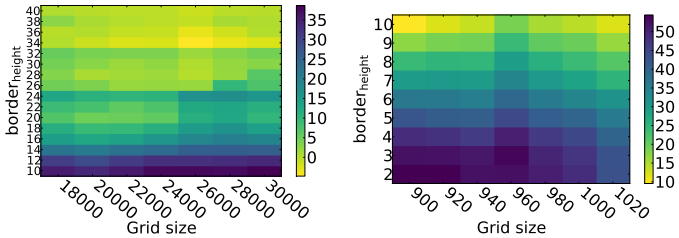
D. Experimental Results

1) Artificial Benchmark: We start with the results obtained from *Artf* as it is possible to measure manifested errors. We are comparing the benchmark compiled with the *Original* implementation of the Pochoir and our *Proposed* one.

In our experiments, we sweep the $border_{height}$ from 10 to 40 and the grid size from 18000 to 30000. The $border_{height}$ in the results of Fig. 6 is chosen to be 20 so that the following constraints are met: *i)* the maximum introduced performance overhead is lower than 15% and *ii)* the maximum Δt_{border} is below 2 seconds that corresponds to BER of 10^{-10}. The results are obtained after executing *Artf* 1340 times, until results stabilize, while varying the grid size. Fig. 6a compares the percentage of the correct runs of the two implementations and Fig. 6b shows the average number of bit-errors occurred. We can observe that given the constraints, our *Proposed* implementation outperforms retaining a high percentage, over 95%, of correct runs across the range of grid sizes, while the *Original* implementation rapidly decrease its quality.

2) Application Benchmarks: Fig. 7 and 8 present the design space exploration for the performance overhead and for the Δt_{border} of the benchmarks *Heat-2D* and *Heat-3D*. The design space is populated from the results of runs with variable $border_{height}$ and grid size. We are sweeping the $border_{height}$ for *Heat-2D* from 10 to 40 and for *Heat-3D* from 2 to 10. Respectively, we are sweeping the grid size from 18000 to 30000 and from 900 to 1020.

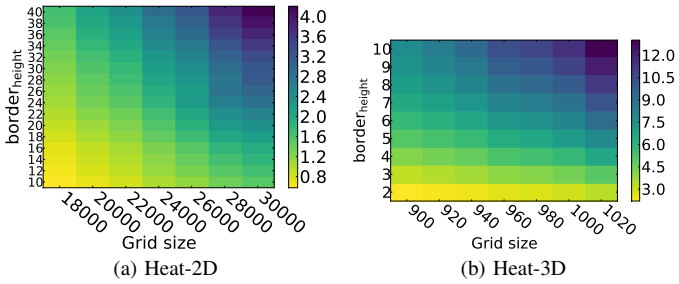
We observe that the performance overhead is mainly affected by the $border_{height}$ caused by the disruption of



(a) Heat-2D

(b) Heat-3D

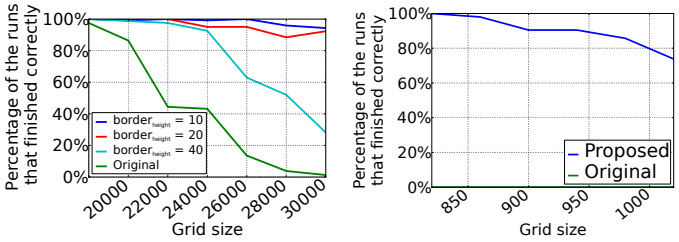
Fig. 7: Percentage of performance overhead introduced from our technique



(a) Heat-2D

(b) Heat-3D

Fig. 8: Achieved Δt_{border} in seconds



(a) Heat-2D

(b) Heat-3D

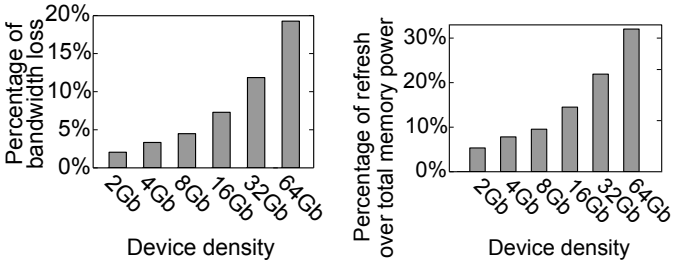
Fig. 9: Percentage of the runs that completed correctly

cache locality and by the increased number of function calls while having smaller *tasks*. Δt_{border} is affected by both the $border_{height}$ and the grid size, as the $border_{height}$ affects the number of stencil computations for each *task* and the grid size affects the number of *tasks* that belong in each $border_{height}$.

We are using the design space to select the parameter values that minimize refresh operations given different constraints such as maximum acceptable performance overhead and the maximum Δt_{border} .

Based on the performed design space exploration, we choose to analyze 3 possible scenarios for better understanding the efficacy of our approach. We are selecting parameters to optimize: *i)* for the minimum Δt_{border} and consequently the lowest BER, *ii)* for the least performance overhead and *iii)* for both those two previous criterion together. So we are exploring the error characteristics for the 3 scenarios with $border_{height} = \{10,\ 40,\ 20\}$, equivalent $max(\Delta t_{border}) = \{1.5,\ 4.5,\ 2.4\}$ seconds and $max(overhead) = \{38\%,\ 5\%,\ 12\%\}$. We can see the results of 1968 runs in Fig. 9a, which depicts the percentage of the runs that completed correctly.

We can observe that for the first and third scenario, the percentage of correct runs remain over 90% for all the grid sizes, while for $border_{height} = 40$, the percentage decreases



(a) Throughput loss due to refresh (b) Power overhead of refresh operations

Fig. 10: Refresh implications on power and performance on current and future DRAM technologies [1]

more rapidly. However in each case the results are better than the *Original* implementation.

In Fig. 9b, we explore one setting for *Heat-3D* with parameter of $border_{height} = 6$ with equivalent $max(\Delta t_{border}) = 8$ seconds and $max(overhead) = 44\%$. For the selected range of grid sizes, the original implementation has no devastating results even in the smaller grids, while the quality degrades slowly with our *Proposed* implementation.

E. Performance and Power Gains

Based on the above results it is evident that there are cases where our approach can relax or omit refresh with minor quality losses. Currently, we do not consider the cost of recovery, however exascale systems [23], [24] do already implement efficient recovery techniques. If we omit the *AR*, we obviously help to limit the *AR* overheads. As seen on Fig. 10, we can gain from 4.4% performance in current technologies and up to 19% in future technologies and decrease the power consumption of the memory system by 9.6% to 31%.

Since our technique can help relax the *AR* or even disable it by merely scheduling the accesses of the application, *RefA* can work orthogonally and complementary to other techniques that relax the refresh rate.

VII. Conclusions

In this paper, we present a non-intrusive method for relaxing the refresh-rate by systematically exploiting the implicit refresh that take place during each memory access. The proposed approach, identifies the suitable algorithmic parameters that influence the access patterns and sets them such that all stored data are being accessed within a target time interval without incurring major performance penalty. By ensuring that all accesses take place within the target time interval we guarantee than the bit-error-rate is kept within acceptable levels. The proposed approach not only helps to limit the power overhead by significantly relaxing the refresh-rate with no required hardware modification but also helps to alleviate the performance penalties incurred by the refresh, The efficacy of the proposed method is demonstrated on an off-the-shelf server running a fully fledged Linux OS and when applied to stencil-based algorithms results show that it is even possible to completely disable DRAM refresh with minor quality loss.

Acknowledgements

The presented research effort has received funding from the European Community' Horizon 2020 programme under grant no. 688540 (UniServer).

References

[1] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-aware intelligent dram refresh," in *Proceedings of the 39th ISCA '12*, pp. 1–12.

[2] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, O. Mutlu, J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An experimental study of data retention behavior in modern DRAM devices," in *Proceedings of the 40th ISCA '13*, p. 60.

[3] R. Venkatesan, S. Herr, and E. Rotenberg, "Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM," in *12th International Symposium on HPCA '06*, pp. 157–167.

[4] M. K. Qureshi, D.-H. Kim, S. Khan, P. J. Nair, and O. Mutlu, "Avatar: A variable-retention-time (vrt) aware refresh for dram systems," in *Proceedings of the 45th IEEE DSN '15*, pp. 427–437.

[5] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving dram refresh-power through critical data partitioning," in *Proceedings of the 16th ASPLOS '11*, pp. 213–224.

[6] A. Raha, H. Jayakumar, S. Sutar, and V. Raghunathan, "Quality-aware data allocation in approximate DRAM?" in *International Conference on CASES'15*, pp. 89–98.

[7] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, "Chargecache: Reducing dram latency by exploiting row access locality," *IEEE Symposium on HPCA '16*, pp. 581–593.

[8] 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 *40th Annual IEEE/ACM MICRO '07*, pp. 134–145.

[9] I. Bhati, M. T. Chang, Z. Chishti, S. L. Lu, and B. Jacob, "DRAM Refresh Mechanisms, Penalties, and Trade-Offs," *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 108–121, jan 2016.

[10] K. Kinam Kim and J. Jooyoung Lee, "A New Investigation of Data Retention Time in Truly Nanoscaled DRAMs," *IEEE Electron Device Letters*, vol. 30, 2009.

[11] M. Jung, D. M. Mathew, C. Weis, and N. Wehn, "Invited - approximate computing with partially unreliable dynamic random access memory - approximate dram," in *Proceedings of the 53rd DAC '16*, pp. 101–104.

[12] I. Bhati, Z. Chishti, S.-L. Lu, B. Jacob, I. Bhati, Z. Chishti, S.-L. Lu, and B. Jacob, "Flexible auto-refresh," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 235–246, jun 2015.

[13] Z. Cui, S. A. McKee, Z. Zha, Y. Bao, and M. Chen, "DTail," in *Proceedings of the 28th ACM ICS '14*, pp. 43–52.

[14] M. Jung, D. M. Mathew, C. Weis, and N. Wehn, "Efficient reliability management in SoCs - an approximate DRAM perspective," in *21st ASP-DAC '16*, pp. 390–394.

[15] J. Mukundan, H. Hunter, K.-h. Kim, J. Stuecheli, and J. F. Martínez, "Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems," in *40th ISCA '13*.

[16] P. J. Nair, C.-C. Chou, and M. K. Qureshi, "Refresh pausing in DRAM memory systems," *ACM TACO '14*, vol. 11, no. 1, pp. 1–26.

[17] Y. Han, Y. Wang, H. Li, and X. Li, "Data-aware dram refresh to squeeze the margin of retention time in hybrid memory cube," in *IEEE/ACM ICCAD '14*.

[18] A. Agrawal, P. Jain, A. Ansari, and J. Torrellas, "Refrint: Intelligent refresh to minimize power in on-chip multiprocessor cache hierarchies," in *19th IEEE International Symposium on HPCA '13*, pp. 400–411.

[19] C. Huzum and P. Cascaval, "March test for static neighborhood pattern-sensitive faults in random-access memories," *Elektronika ir Elektrotechnika '12*.

[20] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *23rd ACM SPAA '11*, p. 117.

[21] J. F. Epperson, *An Introduction to Numerical Methods and Analysis*. Wiley-Interscience, 2007.

[22] "Intel Xeon Processor E5-1600/2400/2600/4600 (E5-Product Family) Product Families Datasheet, Volume Two," 2012. [Online]. Available: http://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-1600-2600-vol-2-datasheet.html

[23] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar, "Exploring automatic, online failure recovery for scientific applications at extreme scales," in *Proceedings of SC '14*, pp. 895–906.

[24] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303–312, Feb. 2006.