View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Queen's University Research Portal

DARE: Data-Access Aware Refresh via spatial-temporal application resilience on commodity servers

Chalios, C., Georgakoudis, G., Tovletoglou, K., Karakonstantis, G., Vandierendonck, H., & Nikolopoulos, D. S. (2017). DARE: Data-Access Aware Refresh via spatial-temporal application resilience on commodity servers. International Journal of High Performance Computing Applications. DOI: 10.1177/1094342017718612

Published in:
International Journal of High Performance Computing Applications

Document Version:
Peer reviewed version

Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

Publisher rights
Copyright 2017 SAGE Publications
This work is made available online in accordance with the publisher’s policies. Please refer to any applicable terms of use of the publisher.

General rights
Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.

Download date:06. Nov. 2017

DARE: Data-Access Aware Refresh via Spatial-Temporal Application-Resilience on Commodity Servers

Charalampos Chalios[1], Giorgis Georgakoudis[1], Konstantinos Tovletoglou[1], George Karakonstantis[1], Hans Vandierendonck[1] and Dimitrios S. Nikolopoulos[1]

Abstract

Power consumption and reliability of memory components are two of the most important hurdles in realizing exascale systems. DRAM scaling projections indicate that with significant increase in DRAM capacity and density comes a significant performance and power penalty, due to the conventional use of pessimistic refresh periods catering for worst-case cell retention times. Recent approaches relax those pessimistic refresh rates only on "strong" cells, or build on application-specific error resilience for data placement. However, these approaches cannot reveal the full potential of a relaxed refresh paradigm shift, since they neglect additional application resilience properties related to the inherent functioning of DRAM.

In this paper, we elevate *Refresh-by-Access* as a first-class property of application resilience. We develop a complete, non-intrusive system stack, armed with low cost *Data-Access Aware Refresh (DARE)* methods, to facilitate aggressive refresh relaxation and ensure non-disruptive operation on commodity servers. Essentially, our proposed access-aware scheduling of application tasks intelligently amplifies the impact of the implicit refresh of memory accesses, extending the period during which hardware refresh remains disabled, while limiting the number of potential errors, hence their impact on an application's output quality. The stack, implemented on an off-the shelf server and running a full-fledged Linux OS, captures for the first time the intricate time-dependent system and data interactions in presence of hardware errors, in contrast to previous architectural simulations approaches of limited detail. Results demonstrate that by applying *DARE* it is possible to completely disable hardware refresh, with minor quality loss that ranges from 2% to 18%, which is far less than the recent approaches based on only spatial properties for data placement.

Keywords

1 Introduction

Power consumption and system resilience are two of the most important challenges towards exascale computing. Projections (Bergman et al. (2008); Giridhar et al. (2013)) show that the memory subsystem will contribute up to 30% of the power budget in exascale systems because of technology scaling of DRAM devices. Besides increasing power, scaling the memory capacity will increase the likelihood of errors, challenging the reliability of system operation. Clearly, solutions must address both the need for reduced power consumption and system resilient operation to be acceptable.

The ever increasing need for memory capacity is driving the aggressive scaling of Dynamic Random-Access Memory (DRAM), which will continue to play a key role by offering higher density than static RAMs (SRAM) and lower latency than Non-Volatile Memory (NVM). However, aggressive DRAM scaling is hampered by the need of periodic refresh cycles 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 guarantee error free storage, but its viability as the parametric variations are worsening and the resultant spreads in retention time increase are in doubt for future designs. In fact, it is becoming apparent that most of the cells do not require frequent refresh and designing for the worst case leads rather to a large waste of power and throughput that may reach up-to 25-50% and 30-50% respectively, in future 32Gb-64Gb densities (Liu et al. (2012)).

Recent approaches (Liu et al. (2012); Venkatesan et al. (2006)) have exploited the spatial characteristics of non-uniform retention of DRAM cells to relax DRAM refresh rates. These methods aim at enabling error-free DRAM storage by grouping rows into different retention bins and applying a high refresh rate only for rows of low retention times. However, such multirate-refresh techniques require intrusive hardware modifications which are costly, hence hinder their use. Equally important is that error-free storage by these approaches may be impossible to achieve in practice, since the retention time of cells changes over time (Qureshi et al. (2015)). Therefore, methods that relax the refresh rates of DRAM should be aware that errors are unavoidable and must be mitigated.

While it is possible to deal with errors in DRAM with error correcting schemes (Qureshi et al. (2015)), these may incur significant power consumption, negating the gains from using relaxed refresh rates. A more viable

[1]Queen's University of Belfast

Corresponding author:
Charalampos Chalios, Queen's University of Belfast
Email: cchalios01@qub.ac.uk

alternative is to mask errors, acknowledging the inherent error-resilient properties of many real world applications. Recent studies (Sampson et al. (2011); Chippa et al. (2013)) have revealed numerous error-resilience properties of applications, such as probabilistic input data processing, iterative execution patterns that resolve or mitigate errors, algorithmic smoothing of the impact of errors, or user tolerance to small deviations in output quality. These error-resilience properties provide opportunities to relax the DRAM refresh cycles without concern about the resulting faults, at least on some application data.

A limited number of studies (Liu et al. (2011); Raha et al. (2015)) have attempted to leverage application error-resilience to relax DRAM refresh rates. Invariably, these studies build on the concept of data criticality to allocate them on either of two different memory domains; one being refreshed at the conventional worst-case rate and the other at a relaxed one. Such methods, hereto referred to as *Criticality-Aware Data Allocation (CADA)*, have exhibited that application-resilience can relax refresh rates, but are still limited in three key aspects: (i) they fail 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*, (ii) they have not employed systematic ways to modulate the classification of data into criticality domains, and (iii) they were implemented and evaluated in simulation, thus ignoring the implications of the full system stack, in particular the time-dependencies that exist between data accesses and cell retention. All those limitations result in missing additional opportunities for aggressive refresh relaxation while reducing the impact of potential errors.

Interestingly, Ghosh and Lee (2007) predict that integrating DRAMs in 3D die-stacks will exacerbate the refresh-related overheads and propose a hardware solution that accounts for the implicit refreshes due to regular memory accesses. By tracking memory accesses, refresh skips those memory rows that have been recently accessed. Nevertheless, this approach requires intrusive and costly modifications to the DRAM memory controller, with space overhead increasing linearly with the size of DRAM memory.

In this paper, we overcome the limitations of CADA by elevating Refresh-by-Access as a first-class property of application-resilience to develop non-intrusive, software methods for *Data-Access Aware Refresh (DARE)*. DARE facilitates refresh relaxation on commodity servers and software stacks without intervention in existing hardware. We present an experimental prototype where we quantify for the first time the potential benefits of DARE standalone or in combination with other schemes, while comparing it to conventional approaches on a real server-grade system stack. Our contributions are summarized as follows:

- We identify and systematically exploit Refresh-by-Access to improve application-resilience and enable aggressive relaxation of DRAM refresh-rate, beyond what is achievable by existing techniques that rely on spatial access properties. Importantly, such an approach helps us reduce errors and enable applications that frequently access memory to operate with acceptable output quality, even when they allocate their data in memory domains without refresh.
- We introduce DARE, a novel and non-intrusive technique that facilitates implicit Refresh-by-Access, thus improves application resilience. Furthermore, DARE minimizes the number of errors under aggressively relaxed refresh-rates. It does so by reordering the execution of application tasks, based on their data read and write access patterns, while considering the retention characteristics of the underlying variably-reliable memory domains. Also, the task reordering if DARE improves upon existing CADA-only methods by systematically moderating and increasing the amount of data that can be stored in memory domains with relaxed refresh-rates.
- We realize a non-intrusive and complete system stack that integrates DARE and captures for the first time the time-dependent system and application data interactions in a system with relaxed DRAM refresh. We present a system that achieves non-disruptive operation of the whole system. The stack achieves non-disruptive operation of the whole system under relaxed DRAM refresh rates, enabling to compare and combine application-resilience techniques, including CADA, DARE, and application-level error mitigation, to evaluate their efficacy for the first time on a complete system.
- We evaluate the proposed method using a variety of multimedia, signal processing and graph applications on a commodity system with server-grade DDR3 DRAM chips. Our findings demonstrate that it is possible to extend refresh cycles by orders of magnitudes compared to the DDRx standard specifications of auto-refresh, or completely disable refresh, while achieving better output quality compared to existing CADA schemes. DARE achieves this without hardware modifications and with imperceptible performance effects.

The rest of the paper is organized as follows. Section 2 describes the typical DRAM organization and refresh operation, while it discusses the related work and motivates the proposed work. Section 3 presents the proposed approach, while Section 4 discusses its implementation. Section 5 describes the executed benchmarks and presents the evaluation results. Finally conclusions are drawn in Section 6.

2 Background and Motivation

2.1 DRAM Organization and Refresh Operation

Figure 1 shows the organization of a DRAM based memory subsystem. In modern architectures there is one on-chip DRAM controller per CPU, attached to one or more memory channels, which can operate independently for maintaining high bandwidth utilization. Connected to each channel can be one or more Dual Inline Memory Modules (DIMMs), which consist of one or more ranks. Each rank is a set of DRAM chips that operate in unison to complete one memory operation. Note that each DRAM chip consists of one or more banks, each of which has a dedicated control logic, so that multiple banks can process memory requests

in parallel. Finally, banks are organized in two-dimensional arrays of rows and columns of DRAM cells, which are composed of one access transistor and a capacitor that holds the information in the form of electric charge.

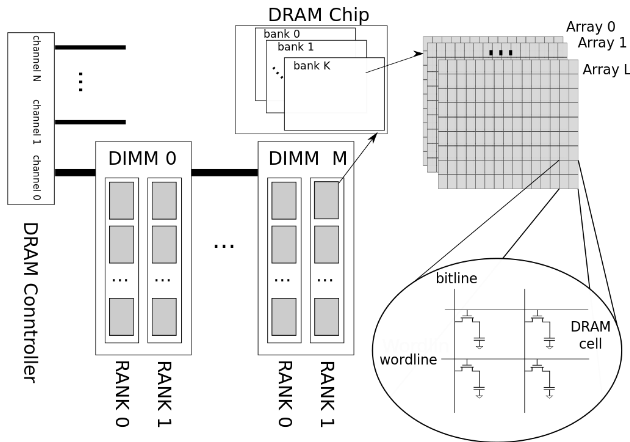
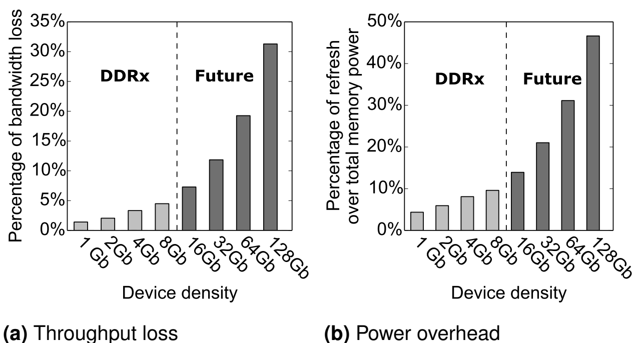


Figure 1. DRAM Organization

Due to the DRAM design, the charge of the capacitor of a cell leaks gradually and may lead to a complete loss of the stored information. The time interval during which the information stored in a DRAM cell can be retrieved correctly is called the *retention-time* T_{RET} of that cell. To maintain the integrity of the data stored cells, the charge of each capacitor must be periodically restored.



(a) Throughput loss **(b)** Power overhead

Figure 2. Power and performance overheads on current and future DRAM technologies (Liu et al. (2012))

Auto-Refresh: Modern memory architectures employ the so called auto-refresh operation, where the memory controller periodically issues a refresh command that triggers the DRAM device to refresh one or more rows per bank depending on its size. During Auto-Refresh (AF), a DRAM bank cannot serve any request and all rows in a rank remain unavailable during refresh. This causes performance loss and consumes significant power refreshing every row. According to the DDR3 specification (JEDEC (2010, 2013)), all rows of the DRAM must be refreshed at least once in a period of $T_{REFW} = 64\ ms$. For this, the DRAM controller needs to send on average one refresh command every $T_{REFI} = 7.8\ us$, which means that within a T_{REFW} period, it issues 8192 refresh commands. As the density of DRAM devices increases, the number of rows that need to be refreshed with every refresh command grows. Therefore, the duration of one refresh operation (T_{REFC}) needs to increase proportionally. As a consequence, the power consumption and the throughput loss incurred because of refreshing is expected to increase considerably in future DRAM generations, as depicted in Figure 2.

2.2 Pessimism of Auto-Refresh Operation

Recent studies on custom FPGA boards 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 $T_{REFW} = 64\ ms$ (Bhati et al. (2016); Kinam Kim and Jooyoung Lee (2009); Liu et al. (2013); Jung et al. (2016b)). To verify this observation on typical server environments, we have performed experiments on various 8GB DDR3 DIMMs of a premium vendor, using our experimental prototype and a memory tester with random and uniformly-distributed data patterns designed to detect the retention time of DRAM bit-cells. The details of the memory tester will be discussed in a following section. Figure 3a shows the spatial distribution of the retention time for one of the DIMMs, where it is evident that around 80% of the cells have a retention time of 5 sec or more, far higher than the 64 ms minimum assumed by the DDRx specification.

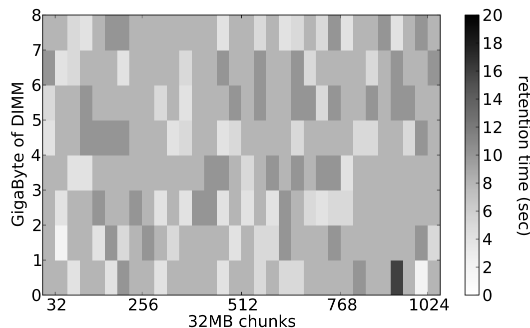
Going one step further, Figure 3b plots the CDF of bit-errors on all the 8GB DIMMs of our system, by aggressively relaxing the refresh period from the conservative 64 ms to 5 sec and up-to 60 sec. Interestingly, even though in our experiments the refresh period is relaxed from $78\times$ (5sec) to $937\times$ (60sec), the cumulative bit error rate (BER) stays low, ranging from less than 10^{-9} to about 10^{-5}. Selecting the appropriate refresh period depends on the number of errors that is considered to be acceptable for applications running on the system. Furthermore, the system needs to guarantee that those errors do not affect any critical system data and lead to catastrophic failures, an issue which we address in this work.

Commercial DRAMs target a BER ranging from 10^{-12} to 10^{-9} when operating under conservative refresh (Nair et al. (2013)). Even under this common scenario, our results indicate that within our server the refresh rate can be relaxed at least by $78\times$ (5sec) if we are prepared to adopt a BER of 10^{-9}. This indicates that insisting on a fixed refresh rate of 64 ms is extremely pessimistic and wastes power and throughput. Accepting a BER of 10^{-9}, or even higher, may allow aggressive relaxation of the refresh-rate but this highly depends on the impact that these errors have on the correctness and output quality of applications.

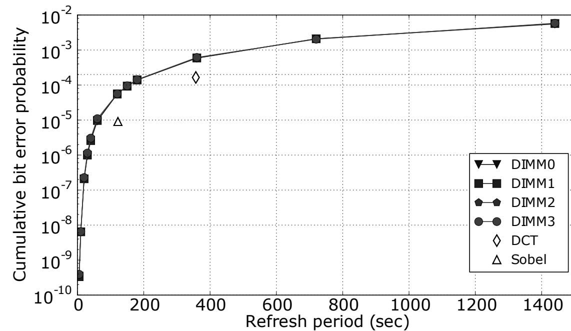
2.3 Related Work on Relaxing Refresh-Rate and Open Issues

Recent schemes have tried to exploit the non-uniform retention time of DRAM cells for relaxing the refresh rate, either by: (i) targeting very low memory BER by selectively applying retention-aware refresh on “bad” or “good” performing cells, or (ii) by allowing a higher memory BER and permitting errors to be masked by the resilient data portions of the application.

Multirate-Refresh: Techniques that preserve low BERs (Liu et al. (2012); Venkatesan et al. (2006); Bhati et al. (2015); Cui et al. (2014); Jung et al. (2016a);



(a) Spatial distribution of worst-case retention time



(b) Cumulative Distribution Function of bit-errors per DIMM with variable refresh intervals

Figure 3. Retention Time Characterization on a Commodity Server with Server-Grade DRAMs

Mukundan et al. (2013); Nair et al. (2014)), which we refer to as *multirate-refresh*, group rows into different bins based on an initial retention time profiling. This approach is similar to the one we apply in our server to select a higher refresh rate for rows belonging to the lower retention 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. Such schemes have been abandoned in modern DDRx technologies due to their high cost. Furthermore, such schemes are deemed impractical since they neglect the fact that the retention time of each cell depends on the stored data and can change at runtime (Han et al. (2014); Qureshi et al. (2015)). Therefore, their essential profiling step cannot be accurate and some cells may be refreshed at a inadequate refresh-rate. This in practice will lead to errors, which may in turn affect system data and result in catastrophic system failures. One can try to address those errors and minimize their impact on the system or the output quality through error-correcting codes (Lin et al. (2015); Qureshi et al. (2015)). However, this will essentially negate any power gains realized out of the relaxed refresh.

Refresh-by-Access: Ghosh and Lee (2007) identify the potential of implicit refreshes through memory accesses to DRAM. Their approach is hardware oriented and builds timers within the memory controller that keep track of the memory rows that have been recently accessed, in order to avoid needless refresh operations to them. However inspiring, their work requires modifications to the DRAM controller and incurs a significant space overhead that grows linearly with the size of DRAM, since they keep one 3-bit counter for every memory row in the DRAM. Moreover, they focus in designing an error-free DRAM operation, neglecting the additional savings that we can achieve if we take into account the inherent resilience of applications.

Error-Resilient Techniques: Another viable solution for addressing the potential errors is to mitigate their effect at the application layer, by exploiting inherent application-resilience properties. Recent studies (Chippa et al. (2013); Sampson et al. (2011)) revealed error-resilient properties of numerous applications, including the processing of probabilistic data, iterative structure which allows averaging of the impact of any error, or user tolerance to small deviations in output quality. A very limited number of works have utilized such a paradigm in DRAMs (Liu et al. (2011); Raha et al. (2015)). These works employ exclusively Critical-Aware Data Allocation (CADA) on different memory domains with variable reliability.

Notably, Flikker (Liu et al. (2011)) is the most representative case of CADA in the literature. The Flikker framework assumes the separation of the memory into two domains, the reliability of which is controlled by the refresh rate. CADA techniques ensure that critical portions of program data are stored on the reliable domain (using the conservative refresh rate), while the rest are stored on a less-reliable domain (with a relaxed refresh rate). They provide a way to execute an application on top of potentially unreliable memory without crashing, and obtain a result with acceptable quality using domain specific knowledge for each application.

Our approach builds on top of CADA techniques and provides a systematic way, that does not require application-specific knowledge, to reduce the number of errors even further, by amplifying implicit refresh due to memory accesses. This way, our method increases the potential power and resilience gains by both enabling more aggressive reduction on the refresh rate and reducing the impact of errors to the application.

2.4 Refresh-by-Access

Every access to the memory naturally opens accessed rows and consequently restores the stored charge in the capacitor of DRAM cells, thus incurring an implicit refresh operation. We call this property Refresh-by-Access and it significantly reduce the number of manifested errors under relaxed refresh rates, thus improve application resilience and system reliability. As we demonstrate in this paper, devising techniques to harness Refresh-by-Access yields more aggressive DRAM refresh relaxation than CADA for a wide range of applications. Studying such a property requires the use of real DRAMs and cannot take place on simulators, since, to the best of our knowledge, there are no models that jointly capture the time-dependent relation between accesses, DRAM retention time and system interactions.

The efficacy of Refresh-by-Access depends on the temporal properties of the application-specific access patterns. To better understand these properties, we execute two different applications (DCT, Sobel) on an off-the-shelf

server using server-grade DRAMs, the details of which are discussed in a later section. The results are shown on Figure 3b, contrasting the expected BER versus the one perceived at the application-level. We observe that the inherently frequent memory accesses of each application result in a much lower number of errors than the number assumed at each relaxed refresh point. The impact of this outcome is two fold: (i) it shows that in practice Refresh-by-Access help to limit the number of manifested errors that need to be tolerated by the application, and (ii) the behavior of an application on a real system will be completely different that the one observed on simulators, as the latter do not capture the impact of Refresh-by-Access.

Therefore, solutions based on CADA may demonstrate completely different efficacy than the one shown in current simulation based evaluations.

The question that remains open is how to intelligently exploit such a temporal property, in isolation or in combination with other methods that exploit spatial properties of cell retention, and how to properly evaluate and capture all the time dependent interactions on a real and complete system stack.

3 The DARE Approach

In this section, we present the DARE approach by formalizing the refresh-by-access property and developing our access-aware scheduling scheme. The objective of the proposed method is to facilitate the refresh-by-access effect, thus minimize the number of manifested errors, by reordering the execution of application tasks, hence memory access patterns. This way, DARE improves the resilience of application even under aggressively relaxed refresh rates.

3.1 Refresh-by-Access Memory Model

The probability of error of a program variable is analogous to the time data in its memory location stay non-refreshed and the type of access to that variable, a read or a write. Reading a variable consumes its data, hence obsoletes that memory location, while writing a variable updates the data in the memory location to be consumed later, either in the program or as the output. Consuming the data propagates the error to the application and this is the key distinction between a read and a write access. Reading a variable means its memory location is vulnerable to errors due to non-refresh periods *before* it is accessed. By contrast, writing a variable means it is vulnerable to errors *after* its memory location is updated, when this variable is actually consumed. We illustrate the time-dependent manifestation of errors and type of access interactions through a simple example.

Figure 4 shows timelines of memory access events of program variables. R_x denotes a *read* access to variable x and W_x denotes a *write access*. Time is quantized as an ordered set of access intervals for illustration. In this example, an application accesses two variables, namely a and b, and those variables are stored in variably-reliable memory within the time interval T_{start} to T_{end}. During this time, errors may appear due to lack of refresh. However, re-ordering memory accesses, while respecting data dependencies, reduces the probability of error. We show this by presenting two different, but both valid, schedules of memory accesses.

In Schedule A, the application accesses, hence implicitly refreshes, variables a and b, at times T_3, T_5 and T_8. The vulnerability to errors for read accesses is determined by the duration between the start of non-refreshed operation and up to the time the read operation happens. By contrast, for write accesses, the vulnerability to errors is determined by the time the write occurs and up to the time the non-refreshed operations ends. For modeling vulnerability, we calculate the error-prone time interval for each read and write memory access. For read access, this interval is equal to the time data are consumed, that is $T_{read} - T_{<start>}$, whereas for write accesses it is equal to $T_{<end>} - T_{write}$ which is the non-refresh interval after which written data may be consumed. These vulnerability intervals by memory access characterize Schedule A in terms of how likely it is for an error to propagate to the application.

However, the vulnerability to errors can be reduced by scheduling the operations of the program differently, re-ordering memory accesses at different times. Schedule B shows such a re-ordering, by swapping R_b with W_a so that reading variable b occurs earlier, whereas writing variable a is pushed later in time. Comparing the vulnerability intervals between those schedules, shows that Schedule B reduces the time data stay non-refreshed before they are consumed, thus it reduces the probability an error propagates to the application.

Notably, re-scheduling the operations in the program must observe data dependencies and program semantics. Instruction-level parallelism and task-level parallelism are well-known (Hennessy and Patterson (2003)) mechanisms for re-ordering operations. However, they have been so far used for increasing performance by improving memory locality. DARE proposes re-ordering at the level of tasks and although optimizing for memory locality may appear contradictory to re-ordering for minimizing errors, we describe further in the paper how we address this issue and demonstrate minimal performance impact.

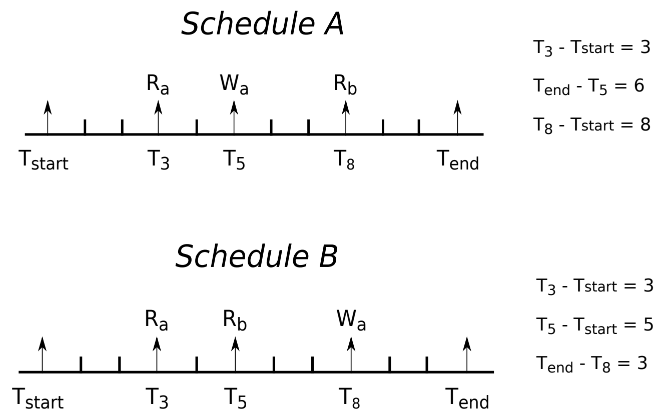


Figure 4. Example of harnessing *Refresh-by-Access* in order to reduce the Expected Number of Errors

3.2 Access-Aware Scheduling

We formulate an optimization criterion and an associated scheduling policy to minimize the time that data remain in memory without refresh. We purposely derive a scheduling

policy that has low complexity, rather than one that exhaustively covers all possible scenarios, in order to demonstrate the key contribution of this paper, namely that refresh-by-access can be tuned on a relevant set of applications.

We assume a *bag-of-tasks* model to describe task-level parallelism in applications. Note that the model is general enough to captures both, parallel DO-ALL loops and data-level parallelism. Also, an applications can potentially contain multiple consecutive regions, each one matching the bag-of-tasks model independently.

Going into more details, let an application consists of n tasks $t_1, \ldots, t_n$. Each task reads a set of memory locations denoted by R_i for task t_i. It writes to a set of memory locations denoted by W_i. Note that the bag-of-tasks model implies that there are no overlaps between any two write sets and between a pair of read and write sets. Otherwise, the tasks would incur data races. For the purpose of this work, we consider only the read and write sets stored in variably-reliable memory. Accesses to data stored in maximum-reliable memory have no impact on the scheduling problem.

Furthermore, we assume that the average retention time is higher than the execution time of a task, such that all of the read and write sets remain valid by the end of the task. Such a minimum retention time is realistic, typically even for the nominal refresh rate. We now formulate the scheduling criterion:

$$\min_{T_1,\ldots,T_n} \left(\sum_{i:R_i \neq \{\}} E(T_i - T_{start}) + \sum_{i:W_i \neq \{\}} E(T_{end} - T_i) \right) \quad (1)$$

where T_i is the time when task t_i executes, $\{\}$ is the empty set and $E(T)$ is the probability of bit errors when data is residing in variably-reliable memory for a time period T. The $E(\cdot)$ function is characterized in Section 2. For the purposes of the scheduling algorithm, it suffices to know that it is a monotonically increasing function, which implies that $E(T)$ is minimized by minimizing T.

We propose a scheduling policy that optimizes Equation 1 by distinguishing three classes of tasks, depending on whether the read or write sets are empty or non-empty. Tasks that only read data in variably-reliable memory ($W_i = \{\}$) should be executed as soon possible to minimize $T_i - T_{start}$. Likewise, tasks that only write data in variably-reliable memory ($R_i = \{\}$) should be executed as late as possible to minimize $T_{end} - T_i$. The remaining tasks either read and write variably-reliable memory, or do not access it all. These tasks are executed after the read-only tasks and before the write-only tasks.

It is common that all tasks are similar in applications matching the bag of tasks model as they correspond to iterations of the same loop. This is also the case in our applications, including the data-dependent graph analytics case. In particular, tasks have similar execution times and equally large read and write sets. Under these assumptions, interchanging tasks within each of the three groups does not further improve the optimization criterion. For example, Figure 5 shows how DARE will reorder tasks in order to minimize errors of Sobel, based on the optimization criterion. Sobel tasks transform an image in order to emphasize its edges. With Default schedule $T_i - T_{start} = 2$ for the read set of the third task, and $T_{end} - T_i = 1$ for the write set of the second task. DARE scheduling reduces both those values to zero time units, minimizing the T.

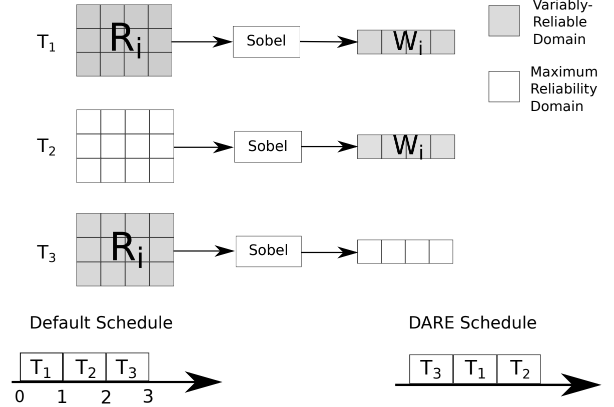


Figure 5. Default and DARE scheduling

Note that an access to a single bit in memory refreshes the whole row of the DRAM bank where that bit is located. As such, accesses to one bit, word or variable may cause others to be refreshed as well. It is possible to model these effects. All interactions between variables can be captured given precise knowledge of the memory locations of variables and knowledge of the mapping of virtual addresses to ranks, banks and rows in DRAM. However, we need not model the memory system in this level of detail, as it is irrelevant for our applications, and we believe for many others as well. The reason is that tasks typically access data that are laid out sequentially and, moreover, those data are sufficiently voluminous to span multiple DRAM rows. Under these conditions, accidental refresh of DRAM through “false sharing” has negligible potential.

The access-aware schedule can be extended to parallel execution by prioritizing the execution order of tasks following the three groups of tasks. Whether those tasks are executed sequentially, as explained above, or in parallel bears no impact. Note however that parallel execution reduces execution time. This way, it reduces the probability of errors even further.

4 Realizing DARE on a Commodity Server

As we discussed, to capture the time-dependent interactions between memory accesses and retention time, which have been neglected by existing approaches, and evaluate DARE it is essential to implement it on a real, complete system stack. This section discusses the proposed implementation of DARE on a commodity server with the required modifications on the Linux OS, for ensuring seamless operation under relaxed refresh, while enabling the allocation of data on variably-reliable memory domains. Note that the detailed modifications that we present depend on the available hardware, e.g, the granularity at which memory refresh is controlled. However, in any case, all steps and especially the changes on the software stack can be followed to realize DARE on any commercial system.

4.1 Hardware Platform

Our platform is 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 device attached to the socket. The iMC exposes a set of configuration registers (Intel (2012)) to control DRAM refresh to: (i) define the period T_{REFI} for sending refresh commands per DRAM channel, (ii) enable or disable refresh for the entire DRAM.

```
software_refresh(refresh_period)
{
  while (true) {
    enable_refresh()
    // wait 64ms to refresh once the whole
    memory
    msleep(64ms)
    disable_refresh()
    // sleep for the remaining target refresh
    period
    msleep(refresh_period - 64ms)
  }
}
```

Figure 6. Software refresh controller

The register controlling T_{REFI} has a 15-bit field for setting the period. This limits the maximum period to 336ms up from the nominal value of 64ms. Even though this is a $5.2\times$ increase, it is still conservative, considering that memory cells have retention times of several seconds. Instead, we opt for a software refresh mechanism that we devise and implement in order to set the refresh-rate at any arbitrary value. The software refresh controller, shown in pseudocode in Figure 6, uses the iMC register interface to enable or disable refresh periodically for emulating a hardware refresh. The software refresh runs as a kernel thread with high priority, independently for each memory domain in the system.

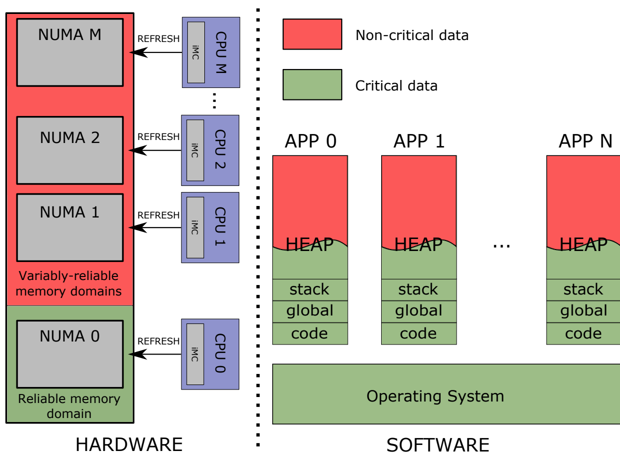


Figure 7. Hardware and software setup for variably-reliable memory

In our hardware configuration, each iMC controls a single memory domain attached to the CPU socket, thus reliability domains are aligned to sockets and NUMA (Non-Uniform Memory Access) domains (Figure 7). DRAMs attached to different CPU sockets may have different refresh periods to implement memory domains with a variable degree of reliability. In our dual-socket system, the first memory domain is deemed reliable with the nominal refresh applied, while the second one is the variably-reliable one, with a configurable refresh. We elaborate on the next section on this organization, driven by the fact that critical data need to be stored and accessed reliably.

Hardware error detection and correction codes (ECC) are a common mechanism employed in DRAMs to improve the bit error probability of the memory. In our testing platform, it is necessary to disable ECC, since detection of an uncorrectable error triggers the System Management Mode (SMM) of the CPU and interferes with our experimentation. SMM is a special, privileged CPU mode that suspends execution of other modes, including the operating system, to execute firmware code that typically reboots the machine to avoid catastrophic failures. However, disabling ECC has the advantage of enabling us to study the worst-case scenario effects of reduced refresh rate operation. Using ECC would capture and correct most of the errors that occur when relaxing the refresh requirements. Evaluating the potential power and performance overhead of the ECC is out of the scope of this paper and it is left for future work.

Before going into the details of the changes on the software stack, we would like also to propose some useful extensions to the hardware interfaces, motivated by our implementation. Although, memory controllers already have an interface to control refresh per-channel, the range of values of T_{REFI} is very conservative, up to 336ms. As we have shown experimentally, most memory cells have retention times of seconds. A straightforward extension is to allow larger values for refresh, which in the case of the iMC translate to larger bit-width for the configuration register, possibly with no other extensions in hardware logic. Also, providing an interface to control refresh per-DIMM, or even per-rank, can unlock more opportunities for fine-grain, selective refreshing.

4.2 Software Stack

In this section we discuss the software platform architecture for enabling variably-reliable memory operation. This includes modifications to the Linux operating system to ensure crash-free system operation, and a system API to control refresh and data allocation on variably-reliable memory. The hardware view of the OS is the one shown in Figure 7. In this setup, the physical address space is divided in two reliability domains. The *Reliable memory* domain contains memory locations that are always refreshed with the nominal refresh period. On the other hand, the *Variably-reliable memory* domain contains memory locations for which the refresh rate can be relaxed at runtime.

In a similar fashion to reliability domains, software characterizes data as critical or non-critical. Operating system kernel data are critical, since even a single error in them may result in a catastrophic failure at the system level. At the application level, data pertaining to the code and stack sections of applications are extremely vulnerable to errors too, i.e., errors in them can lead to an illegal instruction exception or a segmentation fault which is

catastrophic for the application. As a result, those data are critical too. Moreover, static data, allocated at load time, are deemed critical too. This is to avoid changing the system loader and to simplify programming abstractions for allocating data on the variably-reliable memory. Our software platform design provides an API to the programmer for selecting the criticality of dynamic memory allocations on the heap, similarly to other approaches (Liu et al. (2011)). Critical data are always allocated from the reliable memory domain, whereas non-critical data allocate from the variably-reliable memory. Figure 7 also depicts our software platform architecture.

4.2.1 Linux Kernel Modifications and Interface to Variably-Reliable Memory We modify the Linux kernel in order to create the software abstractions for the two reliability domains. This means rendering the kernel reliability-aware for the memory domains and exporting a userspace interface to applications for allocating memory either from the reliable or the variably-reliable domain.

For our implementation, we build on the preexisting concept of *memory zones* (Love (2010)) that the Linux kernel uses to group pages of physical memory for different allocation purposes. Existing Linux system typically include a DMA zone for data operations of DMA devices, a Normal zone for virtual addresses directly mapped to physical ones and in 32-bit systems a High memory zone for pages that go through memory translation to map to physical pages. At the kernel-level, the page allocation interface includes zone modifier flags to indicate the zone the kernel allocates pages from. For example, when a kernel component needs memory from the DMA zone, it request an allocation using the `GFP_DMA` flag.

Moreover, allocation from memory zones follows some priority orderings. For example, when pages from the Normal zone are requested, and the request cannot be satisfied by the free pages of the Normal zone then the allocator is allowed to fall back to free pages of the DMA zone. However, allocations from the DMA zone cannot fall back to the Normal zone.

We define a new zone, namely the *variably-reliable zone*, additionally to the DMA and Normal zones used on our platform. The size of the variably-reliable zone is defined at boot time, by kernel boot arguments denoting the start and end address of the physical address space of variably-reliable memory. In our testing platform, the variably-reliable zone pools pages from the second NUMA domain, which has a relaxed refresh rate to be the variably-reliable memory. Nonetheless, the zone extension we propose is generic enough to allow any subset of the physical address space to be part of the variably-reliable zone. Moreover, allocations from the variably-reliable zone may fall back to the reliable zone in case variably-reliable memory is depleted, but not vice versa. The in-kernel page allocator requests a page from the variably-reliable zone using the `GFP_VREL` modifier flag, any other allocation will be exclusively served by the rest, reliable memory zones.

For the userspace interface, we extend the *mmap* system call for dynamic memory allocation with a new flag, namely `VM_VREL`, which instructs the kernel that the requested memory can be allocated to physical pages of the variably-reliable zone. For the actual allocation, we extend the kernel's page fault handler, so that virtual pages flagged as `VM_VREL` map to physical page frames of the variably-reliable zone. Note that the `VM_VREL` is valid only for heap allocations, otherwise ignored, to ensure that code, stack and static data during application loading are always stored reliably.

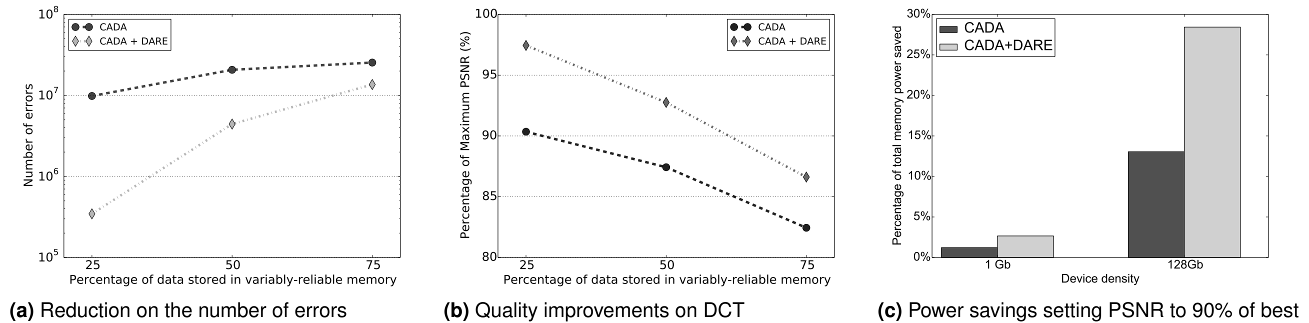
Lastly, we expose a system interface to set the refresh period on the variably-reliable memory domain. For our setup, we extend the `sysfs` kernel interface to have a `refresh_period` entry in millisecond for each NUMA domain under its respective `/sys/devices/system/node/nodeX` device tree. Setting the `refresh_period` entry of a NUMA domain changes the refresh period of the software refresher kernel thread for this domain – the special value 0 means no refresh at all. A similar interface is usable even if reliability domains are decoupled from the NUMA architecture, by adding memory domain proxies in the device tree. Notably, the `sysfs` interface is at the system level, thus it needs administrator privileges to be set. We do not envision this interface to be used by application programmers but rather from system administrators. The presented modifications allow to plugin DARE on any Linux based system.

5 Evaluation

In this Section, we evaluate the efficacy of the DARE approach on minimizing the manifestation of errors and compare it with CADA only techniques. For performing the evaluation, we use the hardware and software platform presented previously and carefully select a range of applications from various domains with different error-resilient properties. Note that both DARE and CADA are evaluated for the first time on a real system. Therefore, our evaluation campaign reveals interesting results and shows how the time dependencies affect application error-resilience and the true performance of DARE and CADA schemes. For each benchmark, we use CADA as the baseline technique to customize data allocation and augment it with DARE, amenable to the specific characteristics of each application. For experimentation, we vary the percentage (PR) of data stored in reliable memory, using conventional refresh, to create different vulnerability scenarios. Moreover, the variably-reliable memory domain operates with no refresh at all, to test the efficacy of error-resilience techniques at the extreme. For each experiment configuration we quantify the output quality in terms of the related metric for each benchmark and measure the time overhead of the applied technique. Each experiment is repeated 10 times and present the average of the results across runs. The compiler used is GCC version 4.8.5. Finally, we discuss the impact of our approach on power and performance for current and future DRAM densities and to compare the gains between approaches under fixed, iso-quality comparisons.

5.1 JPEG and Discrete Cosine Transform (DCT)

JPEG is a widely used application for image compression based on DCT, composed of two parts: (i) compression,



(a) Reduction on the number of errors **(b)** Quality improvements on DCT **(c)** Power savings setting PSNR to 90% of best

Figure 8. Comparison of application resilience techniques for DCT using variably-reliable memory without refresh

during which DCT is applied on the input image and its output is then quantized and stored on memory, and (ii) decompression, during which the stored compressed image in the form of quantized coefficients is dequantized and reconstructed using inverse DCT (IDCT). In the experiments, the input of DCT is a grid of 64×64 image tiles, while each image is 512×512 pixels. The memory consumption of the DCT output, to which we focus, amounts to 8GB.

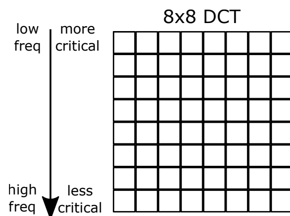


Figure 9. Data criticality within an 8×8 DCT block

CADA It was shown (Karakonstantis et al. (2010)) that in DCT some coefficients, i.e., low-frequency ones, are more significant for determining the output quality than the rest, high-frequency ones. This means DCT is resilient when errors manifest in high-frequency coefficients, since their impact on output quality is limited. The application of CADA to DCT is straightforward in this case. The input image is set as critical data, thus stored in the reliable memory, together with the low-frequency coefficients of each of the 8×8 DCT computation blocks that constitute our kernel. Figure 9 shows the location of the critical coefficients within a DCT block. The lowest frequency, top left coefficient is the most critical and criticality reduces significantly, moving towards the bottom right part of the block. Each task in our experiments reads a 8×8 block of the reliable stored input image, computes the DCT coefficients for a row and stores them to an output array. There is not an exact threshold to how many of the top rows need to be stored reliably to decisively affect output quality. For experimentation, we vary this threshold, translated as the percentage of the output data stored reliably, to investigate its impact on quality. For a given output quality, the more data stored in the variably-reliable memory, the less the power consumption due to relaxing refresh on larger parts of memory.

DARE Following our task model, all tasks read from the reliable memory but depending on the threshold for storing rows reliably, some of them write their output on the variably-reliable memory. Note that DARE reorders the execution of tasks schedule to ensure that tasks writing on variably-reliable memory execute as late as possible, to minimize manifested errors from relaxed refresh execution.

Quality Metric The quality metric for DCT is the Peak Signal-to-Noise Ratio (PSNR), calculated by comparing the original to the reconstructed image. For the tiled images input, we take the minimum PSNR among all tiles in the grid to perform a worst-case analysis.

Results Figure 8 shows results on the number of errors and output quality, varying PR ratios to 25%, 50% and 75% of the data store in the variably-reliable memory. Note that a PR of 0%, meaning all data are stored in the reliable memory, does not utilize the inherent error resilient properties; whereas a PR of 100%, meaning all data are stored in the variably-reliable memory, cannot clearly show the effects of DARE since all tasks read and write variably-reliable data, hence no reordering is meaningful.

Figure 8a shows the number of manifested errors in case CADA-only applies versus augmenting CADA with DARE. DARE reduces the number of errors, up to an order of magnitude when 25% of the data are stored in the variably-reliable memory. Figure 8b depicts the quality as a percentage of the maximum PSNR achievable when refresh is enabled. Notably, the combination of CADA and DARE improves quality consistently, up to 8% in case of PR=25%. From another point of view, CADA+DARE achieves the same quality as CADA, but by storing a larger portion of data in variably-reliable memory, hence enabling relaxed refresh operation on a larger part of memory. An iso-quality comparison fixing the target PSNR to 90% of the best possible shows that CADA achieves this target by having at most 25% of the data versus 50% of the data for CADA+DARE stored in variably-reliable memory. Importantly, CADA+DARE saves more power compared to CADA-only, by enabling to store more data to the variably-reliable domain. Indicatively, Figure 8c shows power savings for the PSNR 90% iso-quality target for current, 1Gb, memory technology and future, 128Gb, technology. Power savings project to be up to 28% by combining CADA+DARE techniques compared to about 14% of CADA-only.

5.2 Sobel filter

Sobel is an image processing filter that is used frequently in edge detection applications. Similar to DCT, the input consists of a grid of 200×200 images, each 512×512 pixels, while the output is a grid of the same dimensions, resulting in a total memory consumption of 20GB.

CADA For Sobel, we relax criticality for both the input and the output data, allocating parts from both data sets to variably-reliable memory. In contrast to DCT, there is no a priori algorithmic property to indicate which parts of the input or output are more significant in determining the output quality. For this reason, we randomly select data to store in the variably-reliable domain using a uniform distribution. Moreover, experiments vary the amount of data stored in the variably-reliable memory, to show the impact on output quality in relation to the application resilience techniques.

DARE In order to compute one pixel (i, j) of the output image, a task reads the (i, j) and all the neighboring pixels of the input image. As a result, to compute and write out the i-th row of the output image, a task reads rows $i - 1$, i and $i + 1$ from the input. Depending on the randomly selected data allocation, a task may only read or only write or both read and write data in the variably-reliable domain. Following the principle of DARE scheduling, tasks that read from variably-reliable memory execute first, followed by tasks that both read and write on it, while write-only tasks execute last, to minimize the expected number of manifested errors.

Quality Metric The output quality is quantified in terms of the PSNR between the output image from relaxed refresh operation and the reference output produced with full reliability enabled. In our evaluation we report the minimum PSNR across image tiles for performing a worst-case analysis.

Results Figure 10 shows the results for Sobel. The PR ratios allocated on the variably-reliable memory are 25%, 50% and 75% of the aggregated input and output data. Applying CADA+DARE reduces consistently errors and improves the PSNR of the output compare to CADA-only. Under an iso-quality comparison, setting the target PSNR to 35dB, CADA achieves this target by having at most 25% of the data stored in the variably-reliable memory, whereas CADA+DARE achieves that even when 75% of the data stored in the variably-reliable memory. As with DCT, CADA+DARE significantly improves application resilience, enabling to relax data criticality to a greater extent than CADA alone, thus allowing to minimize the data that need to be stored reliably. Figure 10c shows the power savings for the existing and future memory technologies under an iso-quality comparison of CADA versus CADA+DARE, setting PSNR equal to a minimum 35dB target. Savings reach up to 35% in future DIMMs, leveraging the ability of CADA+DARE to place more data on the variably-reliable domain, thus requiring less reliable memory, compared to 18% of CADA-only techniques.

5.3 Pochoir Stencil Algorithms

We demonstrate DARE resilience techniques on stencil algorithms extending the Pochoir stencil compiler (Tang et al. (2011)). Pochoir optimizes the computation of stencil iterations using a cache-oblivious divide-and-conquer strategy. The algorithm decomposes the space-time iteration domains using trapezoidal shapes, to improve memory locality. In terms of memory consumption, Pochoir allocates two large array data structures to store the previous and current values of physical quantities to compute each grid point.

For experimentation, Pochoir simulates a 2-dimensional heat dissipation problem with mirroring boundary conditions. Moreover, we vary the grid size (spatial dimensions) from 14K to 30K.

CADA In stencil algorithms no errors are tolerable. This is because errors propagate to neighboring grid points, violating the algorithm's correctness and convergence criteria. Thus, criticality-aware data allocation techniques are unable to enable error resilience from data placement. However, refresh-by-access resilience techniques can relax data criticality, as we discuss next.

Refresh-by-Access The data values stored at the grid points are refreshed regularly due to the iterative execution of the algorithm, reading and writing those points. In case the problem size is small, less than the retention time of memory, this iterative read and write refresh is sufficient to execute error-free.

However, for larger problem sizes, the effectiveness of the refresh-by-data access diminishes because the default Pochoir decomposition favors memory locality, updating the same set of grid points at later time steps before computing another set. This computation strategy delays accessing grid points in the space domain, hence reduces the implicit refresh-by-access. Next, we discuss how the DARE technique applies to Pochoir to facilitate refresh-by-access.

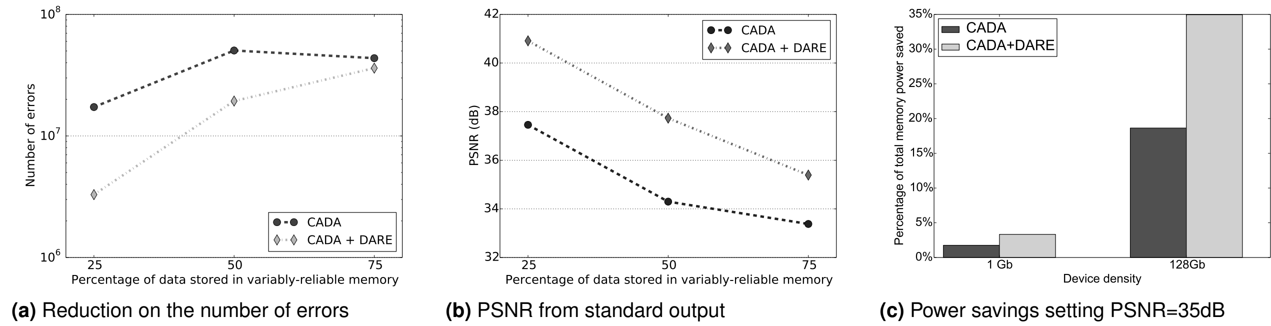
DARE Our modified version of Pochoir performs time-cuts crossing the full spatial domain at regular intervals to enable refresh-by-access. Figure 11 shows this computation strategy. All trapezoids below Δt_i are visited before the algorithm proceeds to later time steps to trade cache locality for error-resilient operation. Nevertheless, the time-cut interval Δt_i is tunable to incur the least possible performance loss while ensuring correctness, we demonstrate this in our results.

Quality Metric Due to the fact that stencil algorithms are not resilient to errors, the quality metric is measured as the percentage of correct runs over the total number of runs of the program.

Results Figure 12 shows the results by contrasting the original to DARE computation decompositions for various problem sizes. The original version of the Pochoir stencil compiler is intolerable to errors. Scaling the problem size to more than 18K results in almost none correct runs. By contrast, the DARE version of Pochoir tolerates more errors and has a smooth degradation as the problem size grows. Notably, the overhead in execution time for the DARE decomposition is less than 4%, averaged across grid sizes.

6 Conclusions

This paper exposed and systematically exploited refresh-by-access as another key property of application resilience to enable aggressive relaxation of DRAM refresh. We proposed



(a) Reduction on the number of errors **(b)** PSNR from standard output **(c)** Power savings setting PSNR=35dB

Figure 10. Comparison of application resilience techniques for Sobel using variably-reliable memory without refresh

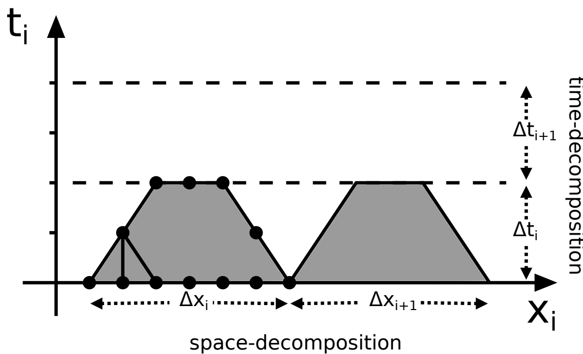


Figure 11. DARE trapezoidal decomposition applied in the Pochoir compiler

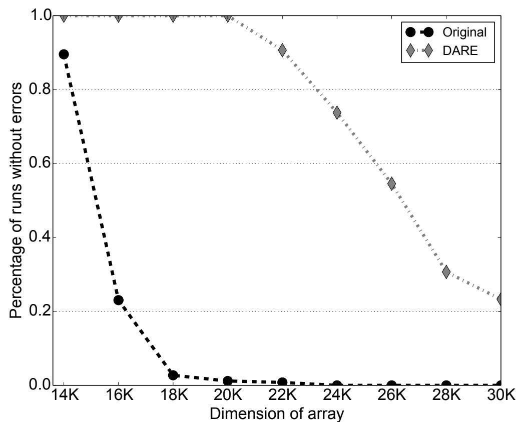


Figure 12. Comparison of DARE Pochoir versus the original

DARE, a novel, non-intrusive method that facilitates refresh-by-access to reduce the number of manifested errors under aggressively relaxed or even completely turned-off refresh. We realized DARE on a complete system stack of an off-the shelf server, to capture for the first time the time-dependent system and data interactions, which is infeasible on existing simulators. The developed system stack is a key instrument to compare and combine multiple techniques for DRAM refresh relaxation, including DARE and CADA techniques. Experimentation results, for a variety of applications with different resilience characteristics, show that is possible to eliminate refresh completely while avoiding avoid catastophic failures and having acceptable output quality as a result of our DARE techniques. Such benefits come with imperceptible performance overhead and with minor output quality degradation that ranges from from 2% to 18%, which is in all cases much less than CADA-only schemes.

References

Bergman K, Borkar S, Campbell D, Carlson W, Dally W, Denneau M, Franzon P, Harrod W, Hill K, Hiller J et al. (2008) Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep* 15.

Bhati I, Chang MT, Chishti Z, Lu SL and Jacob B (2016) DRAM Refresh Mechanisms, Penalties, and Trade-Offs. *IEEE Transactions on Computers* 65(1): 108–121. DOI:10.1109/TC.2015.2417540. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7070756.

Bhati I, Chishti Z, Lu SL, Jacob B, Bhati I, Chishti Z, Lu SL and Jacob B (2015) Flexible auto-refresh. *ACM SIGARCH Computer Architecture News* 43(3): 235–246. DOI: 10.1145/2872887.2750408. URL http://dl.acm.org/citation.cfm?doid=2872887.2750408.

Chippa VK, Chakradhar ST, Roy K and Raghunathan A (2013) Analysis and characterization of inherent application resilience for approximate computing. In: *Proceedings of the 50th Annual Design Automation Conference*, DAC '13. New York, NY, USA: ACM. ISBN 978-1-4503-2071-9, pp. 113:1–113:9. DOI: 10.1145/2463209.2488873. URL http://doi.acm.org/10.1145/2463209.2488873.

Cui Z, McKee SA, Zha Z, Bao Y and Chen M (2014) DTail. In: *Proceedings of the 28th ACM international conference on Supercomputing - ICS '14*. New York, New York, USA: ACM Press. ISBN 9781450326421, pp. 43–52. DOI: 10.1145/2597652.2597663. URL http://dl.acm.org/citation.cfm?doid=2597652.2597663.

Ghosh M and Lee HHS (2007) Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3d die-stacked drams. In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. pp. 134–145. DOI:10.1109/MICRO.2007.13.

Giridhar B, Cieslak M, Duggal D, Dreslinski R, Chen HM, Patti R, Hold B, Chakrabarti C, Mudge T and Blaauw D (2013) Exploring dram organizations for energy-efficient and resilient exascale memories. In: *Proceedings of the*

International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13. New York, NY, USA: ACM. ISBN 978-1-4503-2378-9, pp. 23:1–23:12. DOI: 10.1145/2503210.2503215. URL http://doi.acm.org/10.1145/2503210.2503215.

Han Y, Wang Y, Li H and Li X (2014) Data-aware dram refresh to squeeze the margin of retention time in hybrid memory cube. In: *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '14. Piscataway, NJ, USA: IEEE Press. ISBN 978-1-4799-6277-8, pp. 295–300. URL http://dl.acm.org/citation.cfm?id=2691365.2691425.

Hennessy JL and Patterson DA (2003) *Computer architecture: A Quantitative Approach*. 3rd edition. Morgan Kaufmann.

Intel (2012) Intel xeon processor e5-1600/2400/2600/4600 (e5-product family) product families datasheet, volume two. *Intel* URL http://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-1600-2600-vol-2-datasheet.html.

JEDEC (2010) DDR3 sdram specification. *JEDEC* .

JEDEC (2013) DDR4 sdram specification. *JEDEC* .

Jung M, Mathew DM, Weis C and Wehn N (2016a) Efficient reliability management in SoCs - an approximate DRAM perspective. In: *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. ISBN 978-1-4673-9569-4, pp. 390–394. DOI:10.1109/ASPDAC.2016.7428043. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7428043.

Jung M, Mathew DM, Weis C and Wehn N (2016b) Invited - approximate computing with partially unreliable dynamic random access memory - approximate dram. In: *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16. New York, NY, USA: ACM. ISBN 978-1-4503-4236-0, pp. 100:1–100:4. DOI:10.1145/2897937.2905002. URL http://doi.acm.org/10.1145/2897937.2905002.

Karakonstantis G, Banerjee N and Roy K (2010) Process-Variation Resilient and Voltage-Scalable DCT Architecture for Robust Low-Power Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 18(10): 1461–1470. DOI:10.1109/TVLSI.2009.2025279. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5325663.

Kinam Kim K and Jooyoung Lee J (2009) A New Investigation of Data Retention Time in Truly Nanoscaled DRAMs. *IEEE Electron Device Letters* 30(8): 846–848. DOI:10.1109/LED.2009.2023248. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5165089.

Lin CH, Shen DY, Chen YJ, Yang CL and Wang CYM (2015) SECRET. *ACM Transactions on Architecture and Code Optimization* 12(2): 19:1–19:24. DOI:10.1145/2747876. URL http://dl.acm.org/citation.cfm?doid=2775085.2747876.

Liu J, Jaiyen B, Kim Y, Wilkerson C and Mutlu O (2013) An experimental study of data retention behavior in modern dram devices: Implications for retention time profiling mechanisms. In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13. New York, NY, USA: ACM. ISBN 978-1-4503-2079-5, pp. 60–71. DOI:10.1145/2485922.2485928. URL http://doi.acm.org/10.1145/2485922.2485928.

Liu J, Jaiyen B, Veras R and Mutlu O (2012) RAIDR: Retention-aware intelligent dram refresh. In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12. Washington, DC, USA: IEEE Computer Society. ISBN 978-1-4503-1642-2, pp. 1–12. URL http://dl.acm.org/citation.cfm?id=2337159.2337161.

Liu S, Pattabiraman K, Moscibroda T and Zorn BG (2011) Flikker: Saving dram refresh-power through critical data partitioning. In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI. New York, NY, USA: ACM. ISBN 978-1-4503-0266-1, pp. 213–224. DOI:10.1145/1950365.1950391. URL http://doi.acm.org/10.1145/1950365.1950391.

Love R (2010) *Linux Kernel Development*. 3rd edition. Addison-Wesley Professional. ISBN 0672329468, 9780672329463.

Mukundan J, Hunter H, Kim Kh, Stuecheli J and Martínez JF (2013) Understanding and mitigating refresh overheads in high-density ddr4 dram systems. In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13. New York, NY, USA: ACM. ISBN 978-1-4503-2079-5, pp. 48–59. DOI:10.1145/2485922.2485927. URL http://doi.acm.org/10.1145/2485922.2485927.

Nair PJ, Chou CC and Qureshi MK (2014) Refresh pausing in DRAM memory systems. *ACM Transactions on Architecture and Code Optimization* 11(1): 1–26. DOI:10.1145/2579669. URL http://dl.acm.org/citation.cfm?doid=2591460.2579669.

Nair PJ, Kim DH and Qureshi MK (2013) Archshield: Architectural framework for assisting dram scaling by tolerating high error rates. In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13. New York, NY, USA: ACM. ISBN 978-1-4503-2079-5, pp. 72–83. DOI:10.1145/2485922.2485929. URL http://doi.acm.org/10.1145/2485922.2485929.

Qureshi MK, Kim DH, Khan S, Nair PJ and Mutlu O (2015) Avatar: A variable-retention-time (vrt) aware refresh for dram systems. In: *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15. Washington, DC, USA: IEEE Computer Society. ISBN 978-1-4799-8629-3, pp. 427–437. DOI:10.1109/DSN.2015.58. URL http://dx.doi.org/10.1109/DSN.2015.58.

Raha A, Jayakumar H, Sutar S and Raghunathan V (2015) Quality-aware data allocation in approximate DRAM? In: *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE. ISBN 978-1-4673-8320-2, pp. 89–98. DOI:10.1109/CASES.2015.7324549. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7324549.

Sampson A, Dietl W, Fortuna E, Gnanapragasam D, Ceze L and Grossman D (2011) Enerj: Approximate data types for safe and general low-power computation. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11. New York, NY, USA: ACM. ISBN 978-1-4503-0663-8, pp. 164–174. DOI:10.1145/1993498.1993518. URL http://doi.acm.org/10.1145/1993498.1993518.

Tang Y, Chowdhury RA, Kuszmaul BC, Luk CK and Leiserson CE (2011) The pochoir stencil compiler. *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures - SPAA '11* : 117DOI:10.1145/1989493.1989508. URL http://portal.acm.org/citation.cfm?doid=1989493.1989508.

Venkatesan R, Herr S and Rotenberg E (2006) Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM. In: *The Twelfth International Symposium on High-Performance Computer Architecture, 2006*. IEEE. ISBN 0-7803-9368-6, pp. 157–167. DOI:10.1109/HPCA.2006.1598122. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1598122.