Sparkk: Quality-Scalable Approximate Storage in DRAM

Jan Lucas, Mauricio Alvarez-Mesa, Michael Andersch and Ben Juurlink

Abstract—DRAM memory stores its contents in leaky cells that require periodic refresh to prevent data loss. The refresh operation does not only degrade system performance, but also consumes significant amounts of energy in mobile systems. Relaxed DRAM refresh has been proposed as one possible building block of approximate computing. Multiple authors have suggested techniques where programmers can specify which data is critical and can not tolerate any bit errors and which data can be stored approximately. However, in these approaches all bits in the approximate area are treated as equally important. We show that this produces suboptimal results and higher energy savings or better quality can be achieved, if a more fine-grained approach is used. Our proposal is able to save more refresh power and enables a more effective storage of non-critical data by utilizing a non-uniform refresh of multiple DRAM chips and a permutation of the bits to the DRAM chips. In our proposal bits of high importance are stored in a high quality storage bits and bits of low importance are stored in low quality storage bits. The proposed technique works with commodity DRAMs.

Index Terms—DRAM, Refresh, Approximate Computing

1 Introduction

Refresh is projected to consume almost 50% of total DRAM power only a few generations ahead [1]. For mobile devices DRAM refresh is already a big concern. At the same time the DRAM capacity of modern smart phones and tablets lags just slightly behind regular PCs. Complex multimedia application are now used on phones and often use large parts of their DRAM usage to store uncompressed audio or picture data. DRAM refresh must be performed even if the CPU is in sleep mode. This makes reducing refresh energy important for battery life. Short refresh periods such as 64 ms are required for error-free storage. Most bit cells can hold their data for many seconds, but to ensure error-free storage all cells are refreshed at a rate sufficient for even the most leaky cells. But not all data requires an error-free storage, some types of data can accept an approximate storage that introduces some errors. This provides an opportunity to reduce the refresh rate. Uncompressed media data is a good candidate for approximate storage. However not all media data suitable for approximate storage is equally tolerant to the errors caused by the storage. In this paper we propose a lightweight modification to DRAM-based memory systems that provides the user with an approximate storage area in which accuracy can be traded for reduced power consumption. We show how data and refresh operations can be distributed over this storage area to reach better quality levels than previously proposed techniques with the same energy or the same quality with less energy. Our technique allows an analog-like scaling of energy and quality without requiring any change to the DRAM architecture. This technique could be a key part of an approximate computing system because it enlarges the region of operation where useful quality levels can be achieved.

2 Approximative Storage

Reducing the energy consumption is a topic of ever increasing importance. By relaxing the normal correctness constraints, approximate computing opens many possibilities for energy reduction. Many applications can tolerate small errors and still provide a great user experience [2].

- *All authors are with the Embedded Systems Architecture Department of TU Berlin, Germany.*

This does not only apply to correctness of calculations but also applies to the storage of data values. Often a bit-exact storage is not required and small deviations do not hurt. The amount of variation applications can tolerate depends on the application [2]. We therefore argue that a practical approximative storage system should be able to provide different quality levels. This allows programmers or tools to find a good trade-off between power consumption and quality. Even within a single application, many different storage areas with vastly different quality requirements can exist. Previous work often used binary classifications such as critical and non-critical data [2][3]. This binary classification, however, limits the usefulness of approximative storage. With only a single quality level for approximate storage, applications cannot choose the best accuracy and power consumption trade-off for each storage area, but are forced to pick a configuration that still provides acceptable quality for the data that is most sensitive to errors.

3 Related Work

Among others, Jamie Liu et al. [1] recognized that most DRAM rows do not contain high leakage cells and thus can tolerate lower refresh rates. Most cells retain their data for a much longer time than the short regular refresh period, that is required for error-free storage [4]. They proposed a mechanism named RAIDR to refresh these rows at a lower rate. Because different refresh periods cannot be achieved with the conventional DRAM internal refresh counters, they add a refresh counter to the memory controller. This refresh counter runs at the rate necessary to achieve bit-error free operation, including rows that contain cells with high leakage. The memory controller then generates activate and precharge commands to manually refresh the rows. Manual refresh cycles are skipped if the memory controller determines that they are not necessary. The manual refresh by the memory controller needs slightly more power per refresh operation than the normal auto refresh as the row addresses need to be transmitted to memory. But the authors show that the power saved by the reduced refresh frequency outweighs the power consumed by the more complex refresh signaling. The idea of a memory controller managed refresh and memory controller

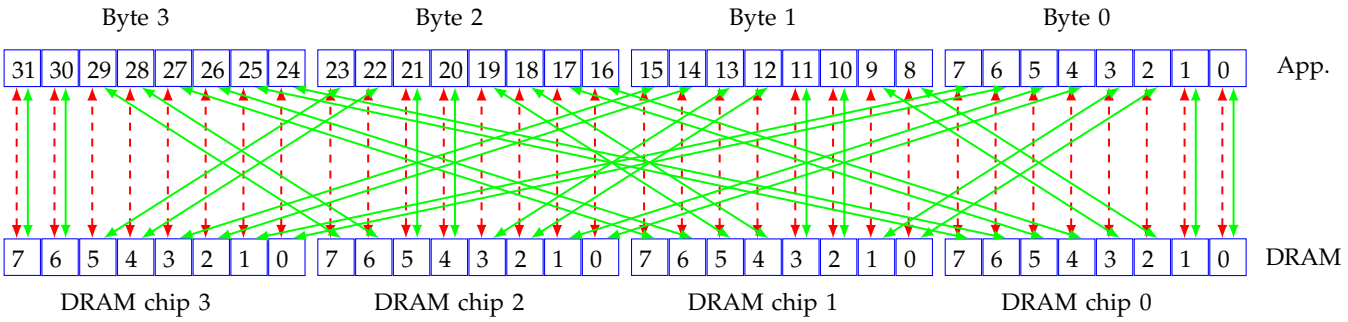

Fig. 1: Mapping of bits to 4 DRAM chips for approximate byte storage

internal row counter is also used in this paper. In RAIDR, the memory controller uses bloom filters to classify row addresses into different refresh bins. Depending on a row's refresh bin, the memory controller issues an actual refresh command only every fourth, second or every time the refresh counter reaches it. This results in rows getting refreshed at 256, 128 or 64 ms periods depending on their refresh binning. Our proposal extends RAIDR by introducing additional refresh bins for approximate data.

Song Liu et al. propose Flikker [3], a memory area with reduced refresh periods for non-critical data. They propose a modified DRAM to enable longer refresh periods on a part of the DRAM. The authors of RAIDR recognized that their work can be combined with the Flikker proposal of Liu et al. Our approach can been seen as an extension of Flikker. It provides an additional reduction of refresh activity for non-critical data. Different from the storage area proposed by Liu et al. this storage area uses varying refresh periods for different bits based on their importance.

Ware and Hampel proposed threaded memory modules [5]. In a threaded memory module, a DRAM rank is split into multiple subranks. The subranks have separated chip select(CS) lines but otherwise share the address and control signals. The CS line controls whether the DRAM chip reacts to transmitted commands or ignores them. By providing multiple CS lines instead of a single CS signal per rank, commands can be issued to a subset of the DRAM rank. Ware and Hampel list various advantages, such as finer granularity transactions, higher performance and lower power consumption. Our proposal also relies on subranks, but uses them to provide bits with different energy/error rate trade-offs simultaneously.

Sampson et al. worked on approximate storage in multi level cells in Flash or PCM [6]. They recognized that storage bits from one cell have different reliabilities and errors can be minimized with their so called striping code. This striping code is very similar to the permutation proposed in this paper.

4 Sparkk

In this section, the first key ideas behind the Sparkk storage are described. Then, it is explained how the DRAM controller manages the refresh of these storage areas.

4.1 Sparkk Storage

Our proposed extension to Flikker and RAIDR is based on two key observations:

1. Even within a single storage area, not all bits are equally critical.
2. Most memory systems require multiple DRAM chips (or multiple dies in a single package) to reach the required bandwidth and capacity.

Applications mostly use and store multi-bit symbols such as bytes. A bit error in the most significant bit of a byte changes the value of the byte by 128, while a bit error of the least significant bit will only change the value by one. Many applications can tolerate errors that change the least significant bit, but will provide an unacceptable quality if the most significant bit fails often.

Regular DDR3 chips have 4, 8 or 16 bit wide interfaces, but most CPUs use wider interfaces. Building a 64-bit wide DRAM interface with regular DDR3 requires at least four 16-bit wide chips or eight 8-bit wide chips. Chip select signals are normally used to control multiple ranks that share the same data lines. The chip select signals for all DRAM chips of a single rank are usually connected together. Only the whole rank can be enabled or disabled. Thus a command is always executed on the whole rank. We propose that the memory controller provides separate CS signals for every chip of the DRAM. This way commands can be issued to a subset of the DRAM chips of a rank. While many additional uses are possible [5], Sparkk uses this to gain more fine grained control over the refresh. With this modification to the traditional interface, different DRAM chips of a rank do not need to share the same refresh profile, as refresh commands can be issued selectively to DRAM chips. Using manual refresh does not work together with self-refresh, as during self-refresh mode the interface to the memory controller is disabled and self-refresh is performed autonomously by the DRAM chip. Requiring one CS line per subrank can also be problematic in systems with many subranks. These problems can be solved by making changes to the DRAM refresh signaling. This, however, requires modifications to the DRAM chips. The exact details of such a scheme remain future work.

Different refresh periods for the same row of different DRAM chips of one rank can make multiple storage bits with different quality levels available simultaneously. But without additional modifications to the usual memory system this does not solve the problem: Some high quality storage bits would used to store low-importance bits and vice versa. It is therefore necessary to permute the data bits so that high-importance bits are stored in high-quality storage bits and low quality storage bits are only used for bits of low importance.

Figure 1 shows how four bytes can be mapped to a 32-bit wide memory interface built from four DRAM chips. The

red dashed lines show a commonly used mapping between application bits and DRAM bits. The green lines show the proposed mapping: The two highest significance bits from all four bytes transmitted per transfer are mapped into DRAM chip 3. The two smallest significance bits are all mapped into DRAM chip 0. Bit errors in DRAM chip 3 now have a much higher impact on the stored values. They will change the stored values by 64, 128 or 192, while bit errors in DRAM chip 0 will only cause small errors in range of 1-3. Data types with more than 8 bits per value have even larger differences in significance between the bits. To find a good trade-off between the number of refresh operations and quality, more refresh cycles need to be allocated for the refresh of DRAM chip 3 than for DRAM chip 0.
Which bits are important varies between data types; here, an examples is shown for bytes. We propose to add a permutation stage to the memory controller that provides a small number of permutations selected for the most common data types. The used permutation could be controlled by the type of instruction used for access, additional page table flags or a small number of high address bits of the physical address. Permutations are not limited to blocks equal to the width of the memory interface: It is also possible to permute the bits within one burst or one cache line. With a 32-bit wide DDR3 interface one burst is 256-bit long, so it may be used to map four 64-bit values to DRAM chips in a way that minimizes errors.

4.2 Controlling the refresh

Aside from the permutation, an effective way for the memory controller to decide whether a row needs to be refreshed is also required. Unfortunately, the solution used by RAIDR cannot be adapted for this task. In RAIDR, the memory controller uses bloom filters to decide which rows need to be refreshed more often. These bloom filters are initialized with the set of rows that contain high leakage cells. A bloom filter can produce false positives, but refreshing some rows more often than needed does not hurt but only causes additional power usage. For schemes such as Sparkk or Flikker, a bloom filter cannot be used: Areas falsely binned as approximate would cause data corruption in critical memory locations and a bloom filter containing all non-approximate rows would use large amounts of memory.
A different solution is required. The authors of Flikker proposed a single boundary in memory space between critical and non-critical data. While this solution is simple, it does not offer enough flexibility as it does not allow for multiple approximate storage areas with different quality levels at the same time. It should be possible to run multiple applications each using multiple approximative storage areas at different quality levels. A practical approximate storage system should thus offer the ability to configure multiple memory areas with different quality levels. This way, a trade-off between power and accuracy can be chosen for each area. Using our proposal, it is also possible to turn off refresh completely for unused bits, e.g. if 16-bit data types are used to store 12-bit data. It is also possible to built approximate storage that guarantees that errors directly caused by the memory stay within configured bounds: More significant bits can be configured to be refreshed at the rate required for error free operation. Only less significant bits are then stored approximately.

Fortunately, we can exploit the characteristics of the refresh counter to build a flexible mechanism that enables the handling of hundreds or thousands of memory areas with different refresh characteristics with a very simple hardware unit and low storage requirements. Every time the refresh counter switches to the next row, the memory controller must decide if a refresh operation should be triggered in the memory and, if so, on which subranks. The refresh counter in the memory controller counts through the rows in a monotonic and gap-less fashion. A mechanism that provides retrieval of the refresh characteristics of arbitrary addresses is therefore not required. At every point in time, it is only necessary to have access to the refresh properties of the current block and information about the end of current block. We propose to store the refresh properties in an ordered list. Each entry contains the address of the end of the current area and informations about the refresh properties of this block. When the refresh counter address and the end of the block address match, we advance to the next entry in the table. Figure 2 shows the proposed hardware unit. The unit uses a small single ported memory to store the list of memory areas. The exact storage requirements depend on the number of DRAM chips per rank, the maximum number of DRAM rows and the maximum refresh period at which a DRAM chip still provides at least some working storage bits. Normally less than 100 bits are required per memory area. Because of the list structure each storage area can be composed from any number of consecutive rows. For each area, the refresh period of each DRAM chip can be freely configured to any multiple of the base refresh rate.

The refresh settings table stores phase and period information for each DRAM chip. Every time a new area is accessed, the controller determines which memory chips should be refreshed within this area: The phase counters for each memory chip are decremented and a counter reads zero, the chip is refreshed and the phase is reset to the value of the period information. A special period value can be used to indicate to omit refresh completely, this be can be another useful feature for some applications: Reading or writing data from a row also causes a refresh of the data. A GPU that renders a new image into a framebuffer every 16.6 ms does not need to refresh the framebuffer. The access pattern guarantees refresh, even without explicit refresh cycles.

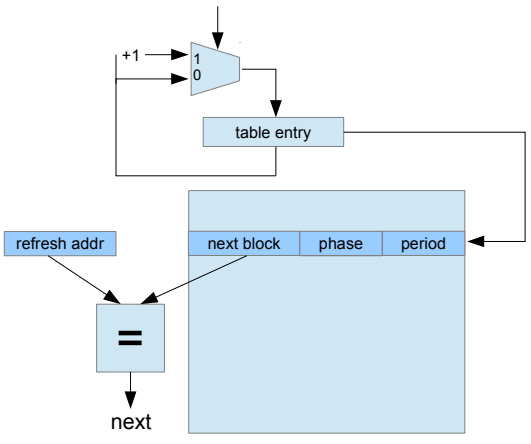

Fig. 2: Refresh settings table

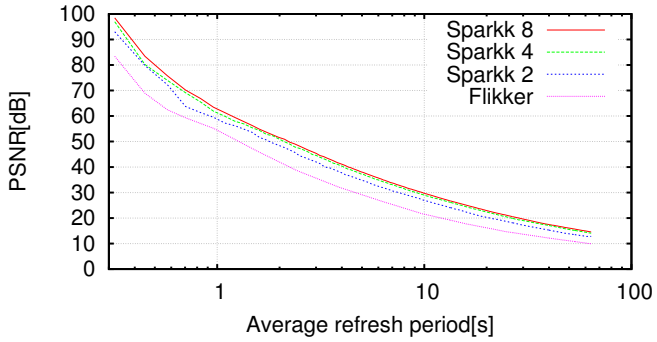

Fig. 3: PSNR for Flikker and Sparkk with 2, 4, 8 DRAM chips

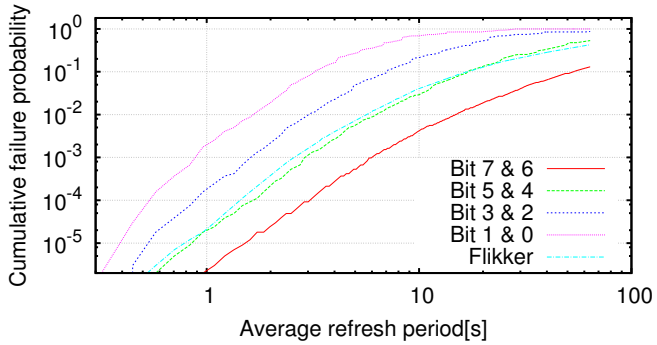

Fig. 4: Effect of variable refresh on failure probability for individual bits for Sparkk with 4 DRAM chips

Different software interfaces to such a storage system are possible: It can be part of a system that is designed from top to bottom for approximate computing and uses specialized languages, specialized instruction sets and specialized hardware such as proposed by Esmaeilzade et al. [7]. In such a system specialized load and stored instructions could select different permutations for each access based on the data type. A compiler aware of these permutation rules could create data structures containing a mix of data types. Even with such a specialized instruction set, refresh settings can only be selected on a per-page granularity. Applications that organize data in arrays of structures thus often need to make problematic trade-offs if the structure members have different accuracy requirements. If applications operate on structures of arrays as it is popular with GPU or DSP applications or data such as pictures or audio, the interface to software can be much simpler: Regular languages such as C, C++ or OpenCL can be used and approximate memory areas may simply be allocated using a specialized malloc. The application must provide informations about size, data type and quality requirements to the specialized malloc. The operating system then calculates refresh settings for meeting the quality requirements, reserves memory and inserts the appropriate entries into the refresh list. If not enough space is available in the refresh list to allocate a new block with a new refresh list entry, the operating system can always choose to provide better than requested storage quality. Adjacent list entries can be merged by choosing the maximum refresh rate from each of the two merged blocks. One important limitation of the memory management of approximate storage blocks should be noted: Approximately stored memory blocks should not be moved from one approximate location to another, as this will cause an accumulation of errors. In two different blocks, different bit cells will fail at a selected refresh rate and bit errors already added to the data will not disappear by moving these bits into new bit cells. In some cases, it might be required to reserve error headroom to allow for data movement. Another possibility to allow data reallocation is to ask the application to restore data to an error-free state by, for example, recalculating the data or reloading it from a compressed file.

5 Modeling of Sparkk

We model the expected number of non-functional DRAM cells at a given refresh period using data for a 50nm DRAM provided by Kim and Lee [4]. Our model assumes that non-functional cells will statically stick to either zero or one on readout. We model cells as equally likely to stick to zero or one. Thus, even a non-functional cell will return a correct result in half the cases.

$$P_{bytechange}(k) = \prod_{i=0}^{7} \begin{cases} P_{bitflip}(i) & \text{if } bit_i \text{ in } k = 1 \\ 1 - P_{bitflip}(i) & \text{if } bit_i \text{ in } k = 0 \end{cases} \quad (1)$$

As shown in Equation 1, the peak signal to noise ratio(PSNR) can be estimated by first calculating the probabilities of the 255 possible changes to a byte, based on the probabilities of changes to the individual bits. $P_{bitflip}(i)$ is the probability of a bitflip in bit i. As already mentioned, this is estimated using the data from Kim and Lee [4]. $P_{bytechange}(k)$ is the probability of the change of a byte by mask k, e.g. $P_{bytechange}(129)$ is the probability of a byte with a bitflip in bit 7 and bit 0 and all other bits without storage errors.

$$MSE = \sum_{k=1}^{255} k^2 P_{bytechange}(k) \quad (2)$$

These probabilities are then weighted by their square error to calculate the mean square error (MSE) as presented in Equation 2. This is slightly simplified and assumes that additional bit flips always increase the error. This is true, if a single bit per byte flips, but not necessarily true if multiple bits in a single byte flip., e.g.: If 128 should be stored and bit 7 flips, the absolute error is 128, but if bit 5 flips as well, the error is reduced to 96. On the other hand, if 0 should be stored and bit 7 flips the error is 128 and if bit 5 flips as well, the error increases to 160. We found that this effect is negligible in practice. This simplification makes it possible to pick good refresh settings for a given quality without knowledge of the data statistics.

$$PSNR = 10 \log_{10} \left(\frac{255^2}{MSE} \right) \quad (3)$$

From the mean square error the PSNR can be calculated, using the well known equation 3. To compare Sparkk with Flikker, the harmonic means of the per chip/subrank refresh rates are calculated. Refresh schemes with an identical harmonic mean trigger the same number of refresh operations per chip/subrank.

Before we can estimate the benefits from Sparkk, it is necessary to find suitable refresh settings that maximize quality at a given energy. With Sparkk, the rows of one approximate storage area within every DRAM subrank can be refreshed at

(a) Flikker (b) Sparkk 8

Fig. 5: Pictures stored in DRAM with 8 seconds average refresh

multiples of the base refresh rate of 64ms. Thus a suitable set of multiples of the base refresh rate must be found. We used hill climbing to find our refresh settings: Starting from the base refresh period the refresh periods of single DRAM chips is gradually increased until a solution is found that meets the average refresh requirements. During this process the refresh period of the DRAM subrank is increased by one base period length that offers the best ratio of newly introduced error and energy saved by the prolonged refresh.

To enable a visual evaluation of Sparkk, we simulated the effect of the approximate storage on image data. Our model was used estimate how many bits of each significance would flip at a given average refresh period. Then, this number of bits of each type was randomly selected and flipped. Kim and Lee modeled in their simulations that retention times of the bit cells are distributed randomly over all cells [4]. Rahmati et al. verified this assumption experimentally and did not find any specific patterns in the retention times of the measured DRAM [8]. There might be patterns regarding which bits tend to stick to zero or one if their retention time is not met. We assume that there are no such patterns and that both stuck-at cases have the same likelihood. In case real DRAM shows patterns, an address based scrambler could be used to prevent patterns caused by the internal DRAM array architecture.

6 Evaluation

Figure 3 shows the expected PSNR as predicted by our stochastic model for Sparkk and Flikker. This model predicts the mean PSNR over an infinite set of samples. The PSNR in single samples can be better or worse, depending on the exact distribution of weak bitcells within the used DRAM rows and stored data, but large derivations from the expected PSNR are unlikely. At all tested refresh rates, Sparkk performs better than Flikker. Even Sparkk with just two subranks provides benefits over Flikker. Sparkk with 4 subranks provides almost all the benefits of Sparkk with 8 subranks.

Figure 4 displays how Flikker and Sparkk distribute the errors over the bits on memory interface with 4 subranks. In Flikker, all bits have the same error rate. In Sparkk, the subrank storing Bits 0-3 is refreshed at lower rate than in Flikker, the subrank with bit 4 & 5 is refreshed at approximately the same rate as in Flikker and the subrank with the two most important bits is refreshed at a higher rate than in Flikker.

To test if the PSNR also provides a good estimate of subjective quality, we generated pictures simulating the effects of Sparkk and Flikker. These pictures can be seen in Figure 5. For Sparkk a configuration with 8 subranks was used. The average refresh rate in both cases was 8 seconds. The image saved in the Flikker storage shows many easy-to-spot bit errors of the most significant bit. The Sparkk storage shows only a few of those errors and despite the extremely long refresh period, the image still seems to have an acceptable subjective quality for some applications such as texturing. The background of the Sparkk stored picture looks grainy which is the result of the high number of bit errors in the less important bits.

Sparkk is able to reduce the number of refresh operations on the DRAM arrays at a given quality level. This reduces the power required for the chip internal refresh operation. However Sparkk requires a more complex refresh signaling and the additional energy consumed by this could potentially mitigate the energy advantage in some use cases. While we proposed one refresh signaling scheme that can be used using unmodified DRAMs, with modifications to the DRAM many other schemes are possible and likely more efficient. It remains an open research question how much energy could be saved exactly.

7 Conclusion

We proposed Sparkk, an effective approximate storage using commodity DRAMs. It achieves more than 10dB PSNR improvement over Flikker at the same average refresh rate or reaches the same quality at less than half the refresh rate of Flikker. We also proposed a simple, small and flexible hardware unit to control how the memory controller refreshes multiple configurable memory areas for approximate storage.

Acknowledgments

This project receives funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the LPGPU Project (www.lpgpu.org), grant agreement n° 288653. We like to thank the anonymous reviewers for their valuable comments.

REFERENCES

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

[2] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-power Computation," in *Proc. of the Conference on Programming Language Design and Implementation, PLDI*, 2011.

[3] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning," in *Proc. of the Conference on Programming Language Design and Implementation, PLDI*, 2011.

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

[5] F. Ware and C. Hampel, "Improving Power and Data Efficiency with Threaded Memory Modules," in *Proceedings of the International Conference on Computer Design, ICCD*, 2006.

[6] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate Storage in Solid-State Memories," in *Proc. of the International Symposium on Microarchitecture, MICRO*, 2013.

[7] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture Support for Disciplined Approximate Programming," in *Proc. of the international conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2012.

[8] A. Rahmati, M. Hicks, D. E. Holcomb, and K. Fu, "Refreshing Thoughts on DRAM: Power Saving vs. Data Integrity," in *Workshop on Approximate Computing Across the System Stack, WACAS*, 2014.