DRAM Decay: Using Decay Counters to Reduce Energy Consumption in DRAMs

Mrinmoy Ghosh Hsien-Hsin S. Lee

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
{mrinmoy, leehs}@ece.gatech.edu

ABSTRACT

Dynamic random access memories (DRAMs) require periodic refresh for preserving data stored in them. The refresh interval for DRAMs depends on the the vendor and the design technology they use. For each refresh in a DRAM row, the stored information in each cell is read out and then written back to itself as each DRAM bit read is self-destructive. The refresh process often incurs large power and bandwidth overhead, however, it is inevitable for maintaining data correctness.

This paper proposes an innovative scheme to address the energy issue in DRAMs. By using a decay counter for each memory row of a DRAM memory module, all the unnecessary periodic refresh operations can be eliminated. The basic concept behind this scheme is that a memory row that has been recently read or written to by the processor (or other devices that share the same DRAM) does not need to be refreshed again by the periodic DRAM refresh operation, thereby eliminating excessive refreshes and the energy dissipated. Based on this concept, we propose a low-cost technique in the design of the memory controller for DRAM power reduction. The simulation results show that our technique can reduce 23% of all refresh operations. This saved 20% of the energy consumed for refresh operations. DRAM system energy savings of up to 17% and an average of 6% were obtained for SPEC2000 integer benchmark programs.

1. INTRODUCTION

Dynamic Random Access Memory (DRAM) is used as the bulk of the main memory in most computing systems for its high density, low cost, and low power consumption. Due to the dynamic, leaky nature of a DRAM cell, periodic refresh operations are required for keeping the data stored. Such regular refreshes account for a large energy consumption in DRAMs even in the Standby mode. For instance, a detailed power analysis of the ITSY computer [13] shows that even in the lowest power mode, the power needed to keep the DRAM contents is about one third of the total DRAM power dissipated. The refresh rate for DRAMs depends on the memory vendor and the design technology they use. A typical refresh interval is 32ms [1]. In other words, for every 32ms, a refresh operation takes place by reading each DRAM cell out and writing back to the same cell. This refresh often incurs large power and bandwidth overhead, nonetheless, it is inevitable for the sake of data correctness.

This paper describes a novel technique called *DRAM decay*, which can eliminate all the unnecessary DRAM refresh overheads. This technique uses a simple decay counter for each row in a memory module, tracks the normal memory transactions, and eliminates the excessive refresh operations. The basic concept behind our scheme is that a memory row that has been recently read out or written to does not need to be refreshed again by a periodic refresh. By simply exploiting such dynamic information, the number of regular row-sweeping refresh operations in a conventional DRAM can be substantially reduced.

2. MOTIVATION

To motivate the case for our DRAM Decay technique, a conjured memory access pattern in Figure 1 is used to demonstrate the requirement for refresh operations. To simplify our illustration, we assume that there are only 8 rows in the DRAM.

In this example, we assume that the DRAM memory is accessed by the processor with a regular access pattern such that each memory row is accessed (represented by vertical solid lines) right before the row is to be refreshed (represented by vertical dashed lines). For a normal, periodic refresh policy, all the memory rows will be, anyhow, refreshed by the memory controller without the knowledge of those recent accesses. Note that each access to a memory row initiated by the processor, in fact, performs an operation equivalent to a regular refresh from the standpoint of data preservation. In other words, if a row has been recently read or written to, there is no urgent need to refresh the row immediately as shown in this figure. For the above example, in an ideal situation, there is no need to perform refresh at all since these regular memory accesses have already accomplished the same effect.

Our DRAM Decay technique exploits such energy savings opportunities by keeping a decay counter for each row in the memory controller to minimize the required refresh cycles. Basically, the decay counters of those rows being accessed will be reset to a default value (e.g. the refresh interval) and any following periodic refresh operation before the counter decays to zero will be aborted. When applying such mechanism to the access pattern shown in Figure 1, the DRAM will not be refreshed at all by the default periodic refresh, without affecting the correctness. In theory, the best pos-

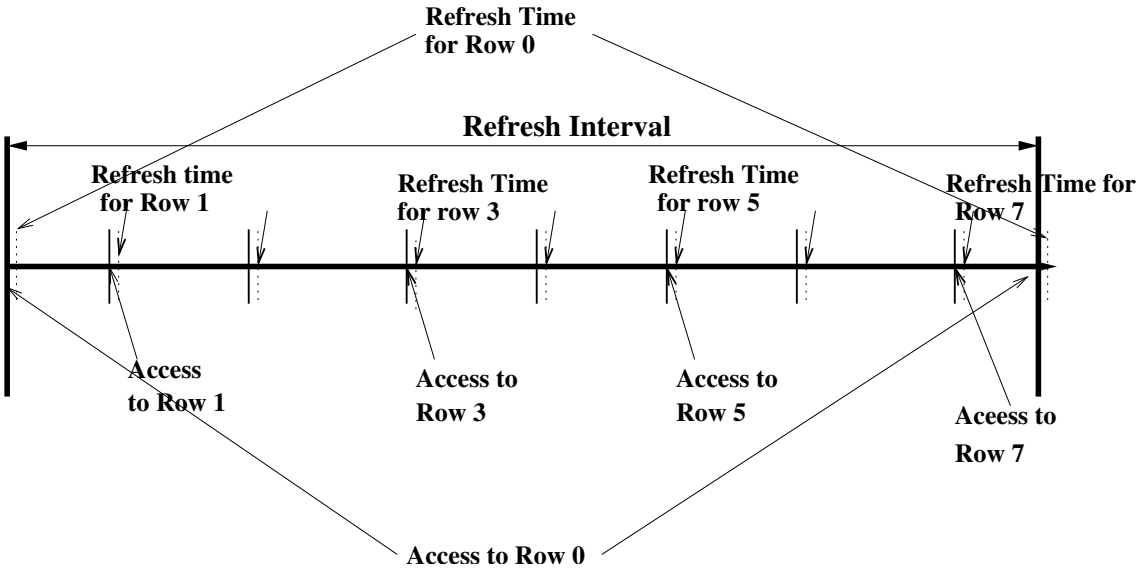

Figure 1: Best Case for DRAM Decay

sible energy savings that can be achieved by using DRAM decay is 50% of the entire DRAM memory.

The rest of the paper is arranged as follows. Section 3 describes different schemes of refreshing a DRAM. Section 4 explains how we will apply our DRAM Decay scheme to reduce the refresh overheads and estimates the overhead. Section 5 discusses two alternatives of the implementation. In Section 6, we describe the evaluation methodology. Section 7 analyzes the results and Section 8 discusses related work. Finally, Section 9 concludes the paper.

3. DRAM REFRESH TECHNIQUES

There are two commonly used refresh modes in commercial DRAM designs:

- Burst Refresh: In burst refresh, the entire refresh operation of all the rows are done one after the other in a bursty fashion. The scheme is less desirable since it increases the peak power consumption of the DRAM. Furthermore, during the time of the refresh operations, the DRAM memory module cannot handle normal access requests, thereby causing potential performance degradation.
- Distributed Refresh: In distributed refresh, the memory controller spreads out the refresh cycles for different rows evenly throughout the refresh interval. This method is preferred than the burst refresh scheme for the following reasons. In addition to maintaining correctness and refreshing all the DRAM cells on time, the memory controller ensures that a large number of refresh operations are done while the DRAM is idle and thus minimizes the overall impact to performance. Also, since the refresh cycles are not adjacent to each other, they do not have a significant effect on the peak power as is the case in the burst refresh scheme.

The following is a description of a DRAM refresh cycle according to a Micron DRAM specification [12]. A DRAM refresh cycle may be implemented in two distinct ways. We must note that a refresh cycle be executed in either the distributed mode or burst mode explained above.

- RAS ONLY REFRESH: To perform a RAS ONLY REFRESH, a row address is put on the address lines and then the RAS signal[1] is dropped. When the RAS falls, that row will be refreshed as long as the CAS signal[2] is held HIGH. It is the DRAM controllerś function to provide the addresses to be refreshed and make sure that all rows are being refreshed at the appropriate times. It is important to note that for refresh operations the row order of refreshing does not matter; however, each row must be refreshed before the data stored by the cell is destroyed.
- CAS BEFORE-RAS REFRESH: This is also known as CBR REFRESH, and is a frequently used method for refresh because it is easy to use and offers the advantage of lower power. A CBR REFRESH cycle is performed by dropping the CAS signal and then dropping the RAS signal. One refresh cycle will be performed each time the RAS signal falls. The Write Enable (WE) signal must be held HIGH while the RAS signal falls. The memory module contains an internal counter which is initialized to a preset value when the device is powered up. Each time a CBR REFRESH is performed, the device refreshes a row based on the counter, and then the counter is incremented. When CBR REFRESH is performed again, the next row is refreshed and the counter is incremented. The counter will automatically wrap and continue when it reaches the maximum allowable value that is equal to the number of rows. There is no way to reset the counter once set after initializing. CBR REFRESH is the more favorable method of refreshing, as it consumes lower power because the address does not have to be put on the bus.

[1]Row Address Strobe
[2]Column Address Strobe

4. DRAM DECAY

4.1 Basic Operation

Similar to the *Cache Decay* technique [8], our DRAM Decay applies the idea of decay counters in the context of the refresh operation for a DRAM to reduce dynamic energy consumption. Before we discuss DRAM Decay we would discuss the basic operation of a DRAM access in more detail. Any DRAM read or write operation initiated by a bus agent (e.g., the processor) starts with the memory controller selecting a bank and dropping the RAS signal. It simultaneously posts the row address on the address bus. This causes the corresponding memory module to activate the sense amplifiers for the entire row, and the data from the given row is brought into the sense amplifiers. We should note here that this read operation has essentially destroyed the data present in the actual DRAM cells. Next, the CAS signal is dropped and the column address is placed on the address bus. This causes the column decoder to multiplex the data out for a read operation. In the case of a write operation, the data on the data bus is written to the correct set of the sense amplifiers. The data for the open row stays in the sense amplifiers till there is an access to another bank or a different row. In either case the data in the sense amps is written back to the original cells and the new row is precharged. To summarize, whenever a row is accessed, it does not need to be refreshed for another refresh interval. This brings us to the concept of DRAM Decay.

The basic idea of our technique is to associate a decay counter for every (bank, row) pair of a memory module. The proposed array of decay counters is stored and updated in the memory controller. Each decay counter is simply a 2-bit or-3 bit binary down counter, whose value, after counting down to zero, indicates that the particular row must be refreshed. The counter is reset to its maximum value whenever the corresponding bank and row in memory is accessed, hence that particular row for the accessed bank need not be refreshed during the regular refresh period. This means that whenever a row is accessed for a normal memory operation (e.g., one induced by cache misses), the refresh operation for that row is delayed. In the best case if every row is accessed right before it needs to be refreshed, there will be no need for a separate, default refresh operation.

4.2 Staggered Decaying

Each counter is decremented after a particular number of memory controller cycles. The counter for a row decays to zero within the refresh interval of that row, if the same row is not accessed again within the refresh time interval. We need to make sure that all the counters are not decremented simultaneously. When this happens, a large number of the counters could become zero at the same time. As a result, all these rows will need to be refreshed simultaneously that leads to a situation with issues similar to the "burst-refresh" mode described in Section 3. This increases the peak power of the DRAM system and during such a situation, normal accesses to the DRAM will be hampered, affecting the overall performance.

To avoid the scenario when all the decay counters are decremented simultaneously in the same cycle, the operation of decrementing counters should be staggered across the refresh interval. We explain the process of staggering the operations of decrementing counters with the help of an example. In this example, we consider a simple DRAM module with 8 rows. Each row has to be refreshed in 16 memory controller cycles to maintain correctness. The events of updating the decay counters and refresh operations for each row are illustrated in Figure 2. We assume that for the time interval illustrated in Figure 2, the DRAM is not accessed. The numbers above the timeline denote the memory controller cycle numbers. We can see that for this example Row 0 is refreshed at cycle 0 and then again after 16 cycles at cycle 16. Similarly Row 1 is refreshed at cycle 2 and will be refreshed again at cycle 18 (not shown in the figure). Since our decay counter used in this example is 2-bit wide, it has to count down three times, i.e. from 3 to 0 within the 16-cycle window. So the countdown of the counter associated with Row 0 is distributed evenly within the 16-cycle interval. We can see that the counter for Row 0 is updated at cycles 4, 8, and 12, respectively where it decreases from 3 at cycle 0 to 0 at cycle 12. In the same way, the counter for Row 1 is only updated at cycle 2, 6, 10, and 14. By doing this, the updates to the counters for different rows are staggered and do not take place during the same cycle. With the scheme illustrated in the figure, for a 2-bit counter, four counters are updated in the same cycle. Likewise, if we use 3-bit counters instead of 2, eight counters will be updated in the same cycle.

The timeline in Figure 2 does not show any memory access to any row within the illustrated interval. However, for example, if Row 2 is accessed at cycle 2, then the counter associated with Row 2 will be reset to 3. Then it will not be (and have no need to be) refreshed at cycle 4 as shown. In this case, Row 2 will only be refreshed at cycle 18 instead of 16 when the counter value eventually decays to 0. In this way the refresh to a row is delayed using the decay counter associated with the row. Also the operation of updating the counters is staggered, thus at most four counters can decay to zero at the same time given 2-bit counters are employed.

4.3 Area Overhead

We now explain the storage overhead for maintaining the decay counters. In our design, we refresh all the ranks and all the channels for a specific bank and row number in one single refresh operation. Hence, we need to maintain a counter for each row in each bank. The total number of counters needed for our design having 16,384 rows and 8 banks is therefore 131,072. Since each counter is 3-bit long, the extra storage overhead will be 48KB. Compared to the DRAM capacity of 256MB, the overhead to support our counters in memory controller is relatively small, less than 0.02%.

5. DESIGN SPACE

5.1 DRAM decay for RAS only refresh

The schematic of the circuitry controlling the refresh operation in the memory controller is shown in Figure 3. We can see that the counter update circuitry updates a specific number of decay counters in a memory controller cycle. If one of the counters that needs to be updated has decayed to zero, then the row number and bank number corresponding to the counter are inserted into the pending refresh requests queue. The memory controller reads the addresses in the pending queue and puts the least recent row address on the data bus and issues an RAS only refresh command. This does not require any change in the DRAM module itself nor

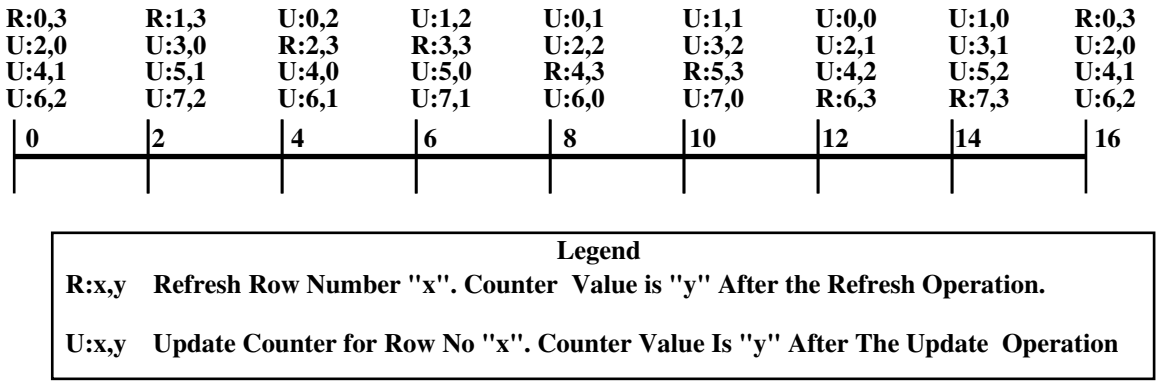

Figure 2: Distributed Row refresh using decay counters

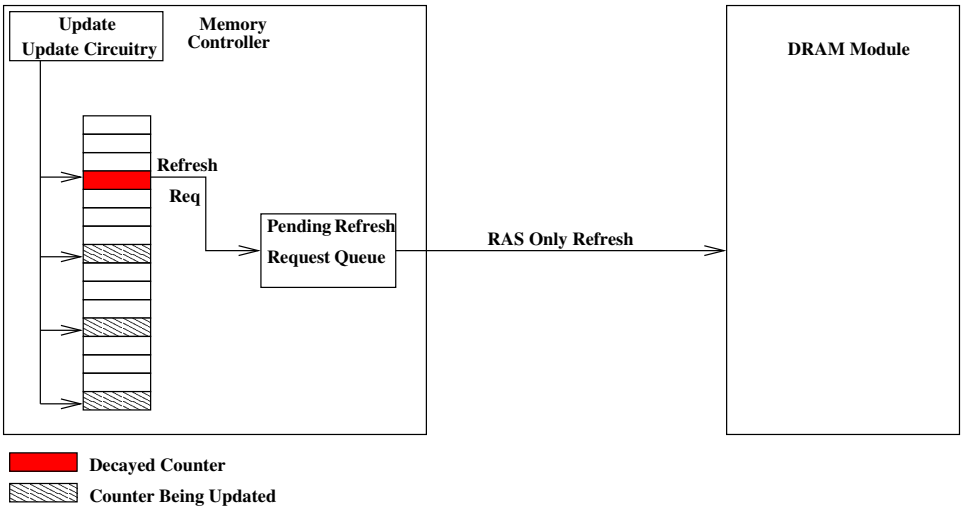

Figure 3: DRAM decay control schematic

the interface between the DRAM module and the memory controller.

One issue of having a queue to store pending refresh operations is to find out any possible case of DRAM access patterns where the queue may overflow. We show that this is not possible. A typical time taken to refresh a row is 70ns [2]. As explained earlier, if the refresh interval is 32ms and there are 8192 rows in the device, the counters are accessed every 4us. Now if a counter has 3 bits, then at most 8 refresh operations may be triggered at a time. To avoid overflow for a queue having eight entries, it is essential that all the eight refresh transactions are handled till the next time the counters are accessed. Since refreshing a row takes 70ns and the counters are accessed every 4us, if there is no normal DRAM access, the number of rows that may be refreshed between successive counter accesses will be 57. Nevertheless, in the worst case, we only need to refresh 8 rows at a time. Thus a queue of length 8 is sufficient for the purpose and it will never overflow.

We would like to emphasize that DRAM decay for RAS only refresh does not change the interface between the Memory Controller and the DRAM module. In the next section, we discuss one possible solution for implementing DRAM decay in the CBR refresh system. The implementation and results of the CBR based DRAM decay technique are however future work and outside the scope of this paper.

5.2 DRAM decay for CBR

In CBR, the problem is that the row address to be refreshed is stored in a register in the DRAM module and not in the memory controller. Every time a CBR command is issued, the DRAM memory module fetches the register, refreshes the corresponding row, increments the register and stores it back. To implement DRAM decay successfully, however, the memory controller also needs to know the value of the register. We will need a special signal that will increment the row address in the register but not refresh the row. We call this special signal *Virtual CBR (VCBR)*. The following is the implementation of the DRAM decay assuming the presence of the VCBR signal. For example, if the memory controller wants to refresh row "K", where the register in the module has a value of "L", the idea is to issue as many VCBR signals as required to make the register value equal to "K". It can be easily seen that the number of VCBR signals required is K-L if K > L, OR (num_rows + L -K) if L > K.

6. SIMULATION METHODOLOGY

Our simulation infrastructure is based on Simics [6], a full system emulator. Simics is an extremely fast full system

Table 1: DRAM Configuration

Parameter	Value
Type	DDR2 Fully-Buffered DIMM
Size	256 MB
Rows	16384
Frequency	667 MHz
Banks	8
Rank	16
Columns	1024
Channel Width	16
Row Buffer Policy	Open Page
Refresh Interval	64ms

emulator that can run unmodified production software like full blown operating systems. This infrastructure was used to emulate a "Sun" virtual machine called "sarek" running a version of Solaris 8. The SPEC INT 2000 benchmark programs were compiled for the Solaris machine and installed in the virtual disk. Although Simics is a full system emulator, we only use it for functional simulation. To simulate memory and cache behavior in details, *Ruby* [11] module developed by Wisconsin was loaded into Simics. Ruby leverages the full system infrastructure of Simics and provides timing simulation for the memory hierarchy. However, Ruby does not faithfully simulate the DRAM behavior. The characteristics of DRAM memory were, on the other hand, simulated using another simulator called DRAMsim [17]. DRAMsim can be used either as a standalone trace driven simulator or as a module that can be integrated into Ruby. The complete implementation of our DRAM decay technique was done in DRAMsim. Table 1 shows the DRAM and other machine configurations used in our simulation. The processor model assumed was an in-order processor model with blocking caches. Each benchmark program was simulated for 600 million instructions.

The calculation of power involved two distinct components: the power consumption of the DRAM module, and the power overhead of the decay counters. To calculate power consumption for the DRAM module we use the power model provided by DRAMSim [17]. For the decay counters we assume a design consisting of an array of SRAM bits storing the counter values and a logic circuit for the decrement operation. In this design, the energy consumption of storing and accessing the array of SRAM bits would be much larger than the energy consumed by the logic circuitry to decrement the respective values. Thus the energy consumption of the logic circuitry was neglected in our energy calculations. The SRAM array was designed using the *Artisan* 90nm SRAM library [4] to get an estimate on the dynamic energy required to access it. The Artisan SRAM generator is capable of generating synthesizable Verilog code for SRAMs using 90nm technology. The generated data-sheet gives an estimate of the read and write power of the generated SRAM. The counter arrays may be accessed in two different situations. First, when a specific row is accessed and its corresponding decay counter needs to be reset. This is considered as a write operation to the SRAM array. The second case is that when a counter is checked against zero value for triggering a refresh. When the value is positive, it is decremented. As explained in Section 4, whenever the counters are accessed for decrementing, eight counters are decremented at the same time. Therefore, in our design we count eight reads and eight writes for each such counter access operation. The results of the simulation will be presented in the next section.

7. RESULTS

Figure 4 shows the number of refresh operations taking place for each benchmark program. We can see that the number of refresh operations differ greatly across different benchmark programs even though they are simulated for the same number of instructions. The reason behind this is that even if the benchmark programs are simulated for the same number of instructions, each benchmark takes a different amount of time to complete due to stalls caused by memory transactions. Indeed we can easily see that the memory intensive benchmarks like bzip have an order of magnitude more refresh operations. This is just because they have more memory transactions and thus take much longer to execute than compute intensive benchmarks like gcc.

Figure 4 also shows that the DRAM decay technique is effective in reducing the number of regular refresh operations. Though the relative reduction in refreshes is heavily dependent on the memory behavior of an application, we can easily see that our technique is successful in reducing the number of refresh operations. The reductions in refresh operations range from as low as 5% for eon to as high as 56% in perl. On the average our technique can reduce more that 222% of all refresh operations over all SPEC2000 integer benchmark suite. We must note that the above results are just for running a single thread. Better results can be expected if multiple threads are executed simultaneously, since multiple threads would access memory more and increase the chance of DRAM decay to perform better.

Another insight from our simulations was that we consistently found that our technique performed better than the baseline. This is because lower refresh operations provide more bandwidth to the DRAM to service normal DRAM operations. If multiple masters controlled the DRAM, this effect will be enhanced and performance improvement would be even more significant.

Figure 5 shows the relative energy consumption for DRAM Decay for refresh operations. We can see that DRAM Decay is successful in saving a significant percentage of energy consumed in refreshing the DRAM. The savings range from 7% in parser to as much as 50% for perlbmk. On an average DRAM decay saves 20% of energy consumed in DRAM Refresh.

Figure 6 shows the relative energy consumption for the DRAM. The right-hand bar showing relative energy for the DRAM decay technique includes the energy consumption for maintaining the counters. We can see that the total energy consumption easily corroborates with the relative number of refresh operations. Thus benchmark programs such as twolf and perl whose relative number of refresh operations was low also show high energy savings of 10% and 17%. The power savings range from as low as 2% in eon to as much as 16% for perl. On average, the total savings of DRAM energy is around 6%.

8. RELATED WORK

Using countdown timers for tracking DRAM refresh was proposed in a patent disclosure [7]. This patent describes a timer based circuitry to reduce the number of refresh op-

erations in a DRAM based cache. This technique is different from ours as it is mainly aimed towards DRAM caches and save power by invalidating lines which have not been modified for a large number of cycles. Venkatesan et al. in [16] introduced RAPID, a retention-aware placement algorithm. This work tries to reduce refresh operations to the DRAM by experimentally identifying that different rows require different refresh times. Our technique is orthogonal to this technique and can be applied on top of the retention-aware DRAM technique. Kim et al. in [9] exploits multiple DRAM refresh times and ECC codes to reduce the number of refresh operations. As in the case of [16], our technique is orthogonal to this technique and thus may be used on top of it. Ohsawa et al. used several techniques in [14] to reduce refresh operations required. One of the techniques used by [14]is to statically declare a line to be dead. This may also be done with the help of the OS. The lines marked as dead in the DRAM are not refreshed. Another scheme is called VRA where counters are used to handle variable data refresh times. We should point out that VRA is different from our scheme as it is done only in the context of handling different refresh times and not to optimize refreshes based on access patterns.

9. CONCLUSION

This paper presented a simple, low cost technique using decay counters to save power in DRAMs. This technique does not involve any change in the interface between the memory controller and the DRAM, making it highly feasible. All additional hardware goes in the memory controller that controls and issues the needed refresh operations. The paper demonstrates that many refresh transactions are indeed not needed for their corresponding rows were recently accessed due to cache misses. This technique saved up to 17% and on an average 6% of the energy consumed in DRAMs. Modern computing systems like CMP, CMT, SMP and SMT would try to exploit MLP and would have increasing number of threads trying to access memory. In this case, the DRAM decay technique will be instrumental in saving energy as it is very light weight and would increase the bandwidth availability and reduce energy consumption for refresh operations in DRAMs. The emerging 3D stacked ICs [5, 10, 15] will enable the accesses to the DRAM memory at a much lower latency. Also, AMD's licensing of ZRAM technology [3] indicate that future AMD processors may use DRAM type memory using SOI technology for their caches. The DRAM Decay technique would be very useful for such DRAM type caches. This is because caches will be accessed more frequently than the DRAM memory in current computing systems. Our technique not only will help save power but also facilitate extra bandwidth that would otherwise be wasted for performing redundant refresh operations.

10. REFERENCES

[1] 128Mb: x32 SDRAM data sheet. http://download.micron.com/pdf/datasheets/dram/sdram/128MbSDRAMx32.pdf.

[2] 512Mb D-Die DDR SDRAM Specification. http://www.samsung.com.

[3] AMD licenses Innovative Silicon's SOI memory. http://www.eetimes.com/news/latest/showArticle.jhtml?articleID=177101749.

[4] Artisan sram generator. http://www.artisan.com.

[5] Tezzaron Semiconductor, FaStack Technology. http://www.tezzaron.com/technology/FaStack.htm.

[6] Virtutech Simics. http://www.simics.net.

[7] P. G. Emma, W. R. Reohr, and L.-K. Wang. Restore Tracking System for DRAM, U.S Patent No 6,839,505 B1, 2002.

[8] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 240–251, New York, NY, USA, 2001. ACM Press.

[9] J. Kim and M. C. Papaefthymiou. Dynamic memory design for low data-retention power. In *PATMOS '00: Proceedings of the 10th International Workshop on Integrated Circuit Design, Power and Timing Modeling, Optimization and Simulation*, pages 207–216, London, UK, 2000. Springer-Verlag.

[10] C. C. Liu, I. Ganusov, M. Burtscher, and S. Tiwari. Bridging the Processor-Memory Performance Gap with 3D IC Technology. *IEEE Design and Test of Computers*, pages 556–564, November-December 2005.

[11] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[12] Micron. Various Methods of DRAM Refresh. http://download.micron.com/pdf/technotes/DT30.pdf.

[13] M.Viredaz and D. Wallach. Power Evaluation of a Handheld Computer: A Case Study. Technical report, Compaq WRL, 2001.

[14] T. Ohsawa, K. Kai, and K. Murakami. Optimizing the DRAM refresh count for merged DRAM/logic LSIs. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pages 82–87, New York, NY, USA, 1998. ACM Press.

[15] K. Puttaswamy and G. H. Loh. Implementing caches in a 3d technology for high performance processors. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 525–532, Washington, DC, USA, 2005. IEEE Computer Society.

[16] R. Venkatesan, S.Herr, and E. Rotenberg. Retention-aware placement in dram (rapid):software methods for quasi-non-volatile dram. In *Proceedings of the Twelfth Annual Symposium on High Performance Computer Architecture*, pages 155–165, Nov. 2006.

[17] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. Dramsim: a memory system simulator. *SIGARCH Comput. Archit. News*, 33(4):100–107, 2005.

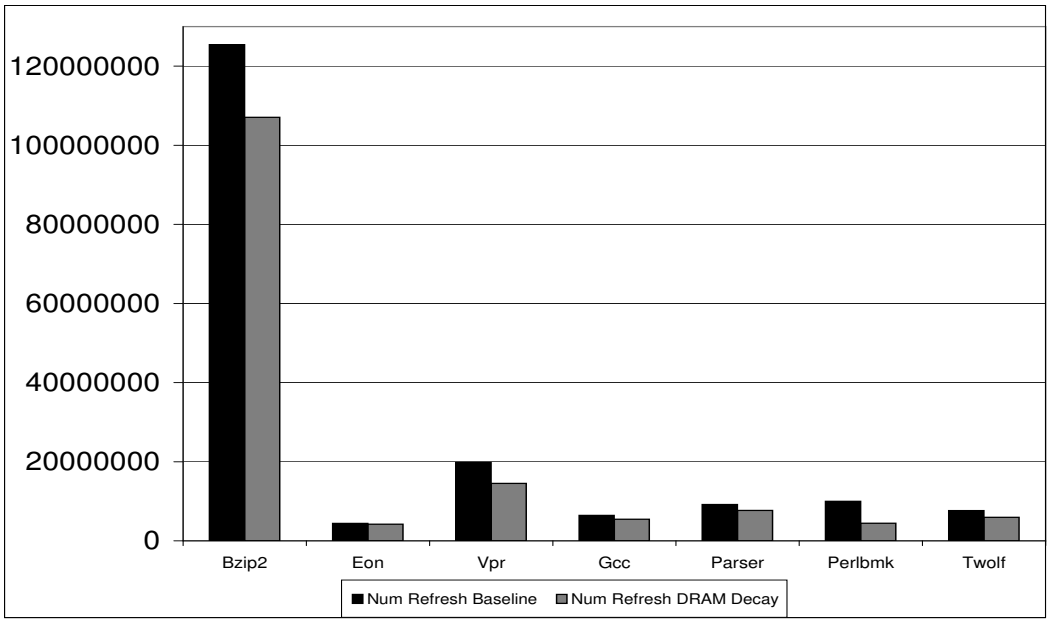

Figure 4: Comparison of Number of Refreshes

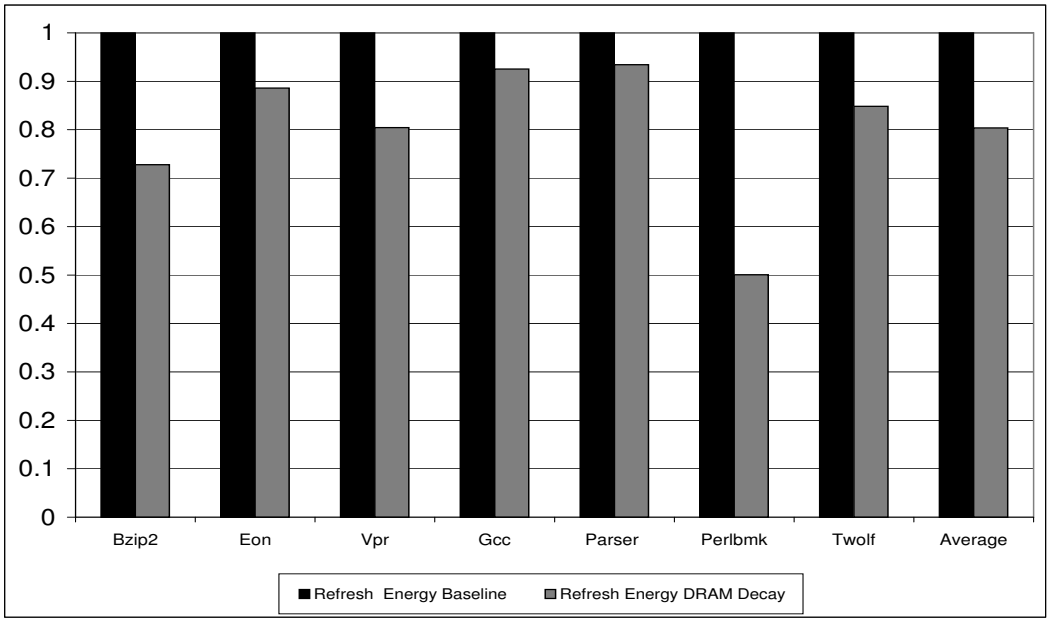

Figure 5: Refresh Energy Consumption

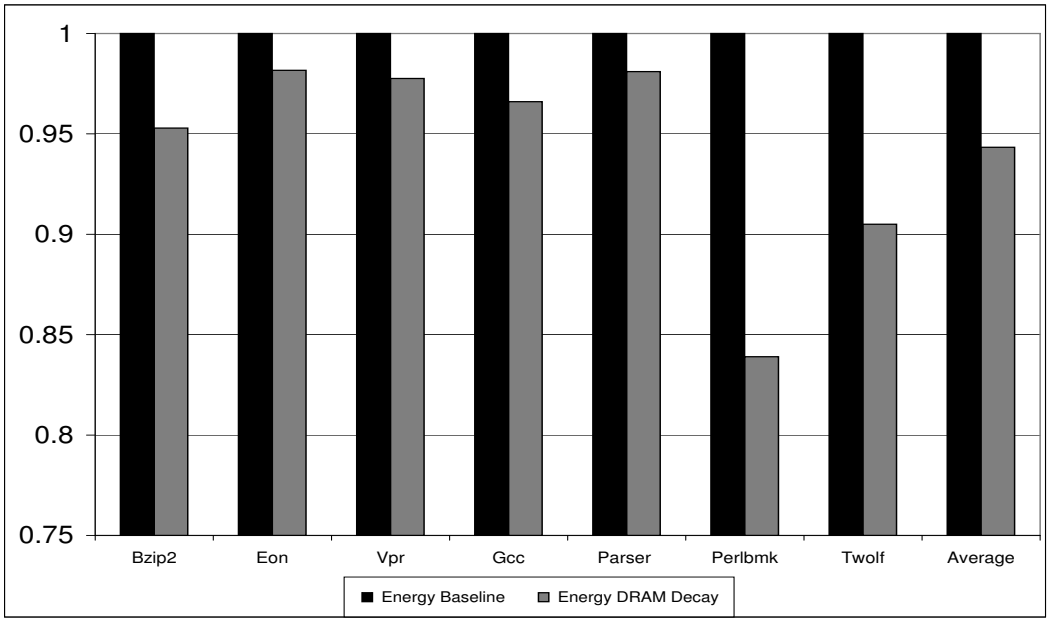

Figure 6: Total Energy Consumption