2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

Reducing Refresh Power in Mobile Devices with Morphable ECC

Chiachen Chou Prashant Nair Moinuddin K. Qureshi

School of Electrical and Computer Engineering
Georgia Institute of Technology
Altanta GA,USA
{cc.chou, pnair6, moin}@ece.gatech.edu

***Abstract*—Energy consumption is a primary consideration that determines the usability of emerging mobile computing devices such as smartphones. Refresh operations for main memory account for a significant fraction of the overall energy consumption, especially during idle periods, when processor can be switched off quickly; however, memory contents continue to get refreshed to avoid data loss. Given that mobile devices are idle most of the times, reducing refresh power in idle mode is critical to maximize the duration for which the device remains usable. The frequency of refresh operations in memory can be reduced significantly by using strong multi-bit error correction codes (ECC). Unfortunately, strong ECC codes incur high latency, which causes significant performance degradation (as high as 21%, and on average 10%).**

To obtain both low refresh power in idle periods and high performance in active periods, this paper proposes *Morphable ECC (MECC)*. During idle periods, MECC keeps the memory protected with 6-bit ECC (ECC-6) and employs a refresh period of 1 second, instead of the typical refresh period of 64ms. During active operation, MECC reduces the refresh interval to 64ms, and converts memory from ECC-6 to weaker ECC (single-bit error correction) on a demand-basis, thus avoiding the high latency of ECC-6, except for the first access during the active mode. Our proposal reduces refresh operations during idle mode by 16x, memory power in idle mode by 2X, while retaining performance within 2% of a system that does not use any ECC.

Keywords*—*Mobile DRAM, DRAM Refresh Rate, Mobile Memory System, Error Correction Code, DRAM Power Consumption, Memory Reliability

I. INTRODUCTION

The past few years has seen a paradigm shift in computing platforms. Emerging handheld devices such as Smartphones and Tablets have become one of the most common devices for computing in everyday use. Energy consumption is one of the prime considerations that influence the development of mobile hand-held devices, as it determines the duration for which the device remains usable on battery power [1][2]. The usage patterns for devices such as smartphones are quite different from traditional computing devices such as workstations. These devices are used in short bursts of few minutes, over extended period of time, as shown in Figure 1. Recent studies [3] have indicated that the idle periods account for 90%-95% for these devices. Therefore, reducing the energy consumption during idle mode of operation has become vital. However, users expect these devices to provide instant response when they are activated; it is also important to retain the application state at the point where it was last used in order to reduce system wake-up time.

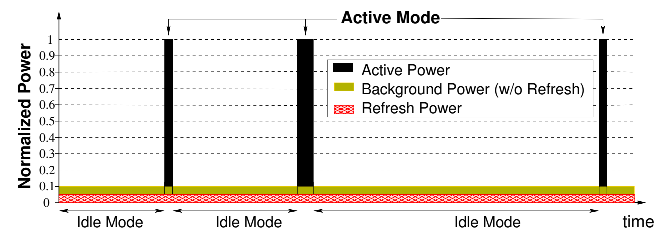

Fig. 1. The typical usage pattern of handheld devices is bursty with long idle periods. During active mode memory consumes 9x more power than idle mode. The contribution of refresh to overall power is small during active mode, but significant only during the idle periods.

One of the main sources of energy consumption during idle periods is the main memory system [1, 4], which takes up to 30% of the energy. The main memory system is typically made of Dynamic Random-Access Memory (DRAM) devices, which requires periodic refresh operations to retain data. When the mobile device becomes idle, the processor can be switched off in less than one millisecond [5] and the memory is put into self refresh mode, where an internal circuitry performs refresh operations. Thus, the memory continues to consume power for retaining data even during idle mode and this is one of the significant source of idle power consumption that needs to be reduced. One option is to restore the memory contents into non-volatile storage (i.e., flash or external SD card). This may be feasible in systems where the memory capacity is small and storage bandwidth is high. However, the memory capacity on current smartphones is already in the 3GB range [6]. The bandwidth on Flash storage in mobile devices is in the regime of 32MB-64MB per second, which means saving and restoring state of memory from storage would incur several seconds of delay [7], resulting in large response time and degrading user experience. Therefore, platforms such as Android try to retain the state of recently accessed apps in the main memory to reduce application loading times [8].

The refresh power of memory can be reduced by exploiting the variability in retention times of DRAM cells. By using

978-1-4799-8629-3/15 $31.00 © 2015 IEEE
DOI 10.1109/DSN.2015.33

IEEE computer society

multi-bit Error Correction Code (ECC), one can correct the few bits that fail with slower refresh operations [5]. Our analysis shows that correcting for up to six bits of errors per line (ECC-6) can allow the system to reliably reduce the refresh interval from 64ms to 1 second. Unfortunately, such multi-bit error correction incurs several tens of cycles of delay for decoding operations, which increases the memory latency and reduces system performance. We found that for our baseline system, the latency overhead of ECC-6 degrades performance by as much as 21%, and on average by 10%. Ideally we want to retain high performance of weak ECC (e.g., SECDED or No ECC) and the refresh power savings of ECC-6.

To obtain high performance of weak-ECC codes and the refresh saving of strong-ECC codes, this paper proposes *Morphable ECC (MECC)*. MECC is based on two key observations. First, the idle periods in mobile devices are long, in the range of several minutes. Second, the contribution of refresh power to overall memory power is small during active mode and significant during idle mode. Therefore, we can have the best of both worlds by trying to optimize refresh power only during idle mode (using strong ECC code) and optimize for performance during active mode (using weak ECC code).

MECC appends each line with *ECC-mode* bits, which indicates whether the line uses strong-ECC or weak-ECC code. When the system becomes idle, MECC ensures that the entire memory uses ECC-6, and reduces the refresh rate by 16x to 1 second. Thus, the refresh power and the memory idle power are reduced significantly. When the system becomes active, and a line is accessed from memory, ECC-status is checked. Given that for the first access, the line would have ECC-6, it is decoded with ECC-6 decoder and written back with weak-ECC. We refer to this conversion form strong-ECC to weak-ECC as an *ECC-Downgrade*. All subsequent access to the line in the active period is decoded with the weak-ECC decoder, which has much lower latency, and avoids the performance degradation of the strong-ECC. Thus, with Morphable ECC, the system pays the latency overhead of strong-ECC only on the first access, but not the subsequent access. When the active mode finishes, the system becomes idle, MECC converts the line to ECC-6 and marks the ECC-mode bits associated with lines as such. We refer to this conversion from weak-ECC to strong-ECC as an *ECC-Upgrade*.

MECC is a purely hardware proposal that does not require changes to the source code and does not compromise application reliability for power saving [7]. Our evaluations with 28 applications shows that MECC reduces refresh operation in idle mode by 16x, while providing a performance that is within 2% of a system that does not incur any latency overhead from error correction.

On entering idle mode, MECC tries to convert all the lines in memory to ECC-6. This may be wasteful if majority of the lines in memory were not accessed since the last idle period, and therefore were already equipped with ECC-6. To avoid such wasteful conversions, we propose a simple *Memory Downgrade Tracking (MDT)* scheme that tracks memory regions that have been downgraded from ECC-6. When the system goes to idle mode, only the memory regions indicated by MDT are converted to ECC-6. We found that a simple MDT with 128 bytes storage reduces the system upgrade latency from approximately 400ms to 50 ms.

When a mobile device is not used, it may still get frequently invoked by periodic operations such as interrupts interrupts from I/O, network devices, bluetooth signal check etc. Fortunately, such periodic operations tend to be quite short (few milli seconds) and are typically not bounded by memory peformance. Such periodic system activity willl incur transitions of ECC-Downgrade and ECC-Upgrade frequently, which may ruin the benefits of MECC. To avoid this, we propose *Selective Memory Downgrade (SMD)* which can avoid the transitions between ECC-6 and ECC-1 for such processes. SMD periodically checks the memory traffic and starts ECC-Downgrade *only* if the application has memory traffic above a certain threshold. We found that for minor degradation in performance ($< 2\%$), this extension of MECC does not enable ECC-Downgrade in active mode for 7 out of 28 applications, and all of these 7 applications have small memory footprint and their performance is not sensitive to memory latency.

II. Background and Motivation

Ideally users want the mobile devices to be energy proportional, in that they consume power when used and do not consume any power when idle. This is especially important given that these devices are idle most of the time. One of the major components that make mobile devices non-energy-proportional is the main memory system that is made of DRAM. DRAM relies on periodic refresh of data to maintain data integrity. Even when the device is idle, refresh operations are done to maintain the contents of main memory. The energy overheads associated with refresh is proportional to the capacity of the main memory system, as all lines must be refreshed in a given time period. To enable mobile devices to execute a large variety of applications and to reduce the load times of applications, the memory capacity of smartphone is on the rise. While the first generation smartphones had 128MB-256MB of DRAM, current smartphones (such as Samsung Galaxy Note 3 [6]) already have 3GB of DRAM, and the next generation devices are expected to have 4GB DRAM [9]. Thus, the power consumption due to memory refresh is only going to increase for future mobile platforms.

In this section, we first discuss the various modes of doing refresh in DRAM systems, then we explore the trade-off of DRAM cell failure versus refresh rate, next we describe the usage of strong Error Correction Codes (ECC) to mitigate the failures due to refresh, and finally discuss the shortcomings of always using strong ECC.

A. DRAM Refresh Modes

Refresh operations are performed by simply activating and precharging the particular row. JEDEC specification dictates that the contents of the DRAM device must be refreshed every 64ms to maintain data integrity. Existing standards provide several implementations to perform refresh in DRAM systems, each geared for different system requirements. We describe the refresh implementations below.

1) Auto Refresh (AR). This is the typical mode of refresh, where the memory controller sends a refresh pulse every 64ms (burst mode) or 7.8μs (distributed mode) to the DRAM device. In DRAM devices, there

is an internal register to keep track of the address of the row(s) to be refreshed.

2) Self Refresh (SR). This mode of refresh is employed in idle periods where the processor and the memory controller are turned off. The responsibility of generating periodic refresh pulse is relinquished to the DRAM device. DRAM array cannot be read while in self refresh mode.
3) Partial Array Self Refresh (PASR). A type of self-refresh mode where only a portion of memory is refreshed (other contents get lost). Thus, PASR reduces the useful capacity of DRAM memory system.
4) Deep Power Down (DPD). An ultra low-power mode where DRAM is not refreshed. The contents of DRAM cells are lost. Before coming out of the Deep Power Down mode, the DRAM cells are initialized.

Ideally we want to use the main memory capacity for maintaining the working set of active applications, and to retain the recently/frequently used applications in memory in order to reduce the application loading time. Therefore, we want the power savings close to PASR or DPD, and yet have a usable capacity of Auto/Self Refresh. We can obtain the dual goals of power savings and useful memory capacity if we can significantly reduce the refresh rate in Self Refresh Mode without compromising data integrity.

B. Increasing DRAM Refresh Period

The time for which a DRAM cells retains its data is called the *retention time*, which is typically 64ms specified by JEDEC standards. This rate is determined such that even the weakest bit in the memory array can get refreshed in time. Thus, the refresh rate is inherently determined by the retention characteristics of the weakest cell. On average, DRAM cells have a retention time in the range of few (tens of) seconds. However, there is variability in retention time which causes a few weak bits in the DRAM array to determine the memory retention time. There are several device level studies that characterize the retention time of the DRAM cells. Figure 2 shows the bit failure probability for DRAM cells as the retention time is changed (data derived from [10]).

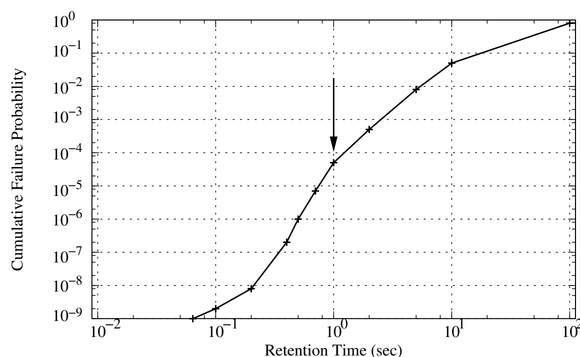

Fig. 2. Retention Time Distribution for DRAM cells (60nm Technology). Y axis shows the bit failure probability for a given retention time. (Figure derived from [10])

The bit failure probability at 64ms is in the regime of 10^{-9}, which means 1 out a billion bits is expected to fail. Such weak bits are decommissioned with device testing and replaced by spare rows and columns, and the shipped DRAM chip is expected to have all bits working at 64ms refresh period. Unfortunately, spare rows are expensive means of decommissioning bad bits, as they require an overhead of 1KB or more for sparing each failed bit; therefore, it is not practical to use when the bit error rate is high (in the regime where we have few tens or more failed bits) [11, 12].

As the refresh period is increased, the bit error rate increases significantly. At 1 second retention time, the bit error rate is in the range of 10^{-4} to 10^{-5} regime. We will use a raw bit error rate of $10^{-4.5}$ as a default in our studies for a refresh period of 1 second. This means if we simply increase the refresh period to 1 second, we can expect approximately 32K bits to fail in a 1Gb array, and 256K bits in a 1GB memory. If we are to employ a 1 second refresh rate to reduce the memory refresh power, the system must be provisioned with means to tolerate high bit error rates.

C. Error Correction for Reducing Refresh

Error Correcting Code (ECC) is often used to tolerate soft-errors due to alpha particle strikes. A similar approach to tolerating refresh errors can be provided in memory systems by appending them with stronger levels of error correction. So, in general, each line can be provided with the capability to correct say K errors. A key question is what should be the error correction strength that is required to ensure a refresh period of say 1 second. For this analysis, we make two assumptions, both of which are consistent with the recent literature on memory reliability studies. First, the errors are uncorrelated, that is each bit has a uniform and independent probability of error[13, 5, 14]. Second, we deem the mechanism to be useful if the likelihood of a system with an erroneous line is less than 1 system out of 1 million systems (this is much stronger guarantee than what is employed in cache studies, so chance of data corruption is negligible).

TABLE I. LINE FAILURE AND SYSTEM (1GB MEMORY) FAILURE PROBABILITY FOR BIT ERROR RATE OF $10^{-4.5}$ (64B CACHE LINE SIZE)

ECC strength	Line Failure	System (1GB) failure
No ECC	$1.8 \cdot 10^{-2}$	1.0
ECC-1	$1.6 \cdot 10^{-4}$	1.0
ECC-2	$9.8 \cdot 10^{-7}$	1.0
ECC-3	$4.5 \cdot 10^{-9}$	$7.2 \cdot 10^{-2}$
ECC-4	$1.6 \cdot 10^{-11}$	$2.7 \cdot 10^{-4}$
ECC-5	$4.9 \cdot 10^{-14}$	$8.1 \cdot 10^{-7}$
ECC-6	$1.2 \cdot 10^{-16}$	$1.8 \cdot 10^{-9}$

Table I show the probability of failure of a line and for a 1GB memory system when the raw bit error rate of each cell is $10^{-4.5}$. The error correction level per line is varied from zero to six. We denote the error correction code that corrects up to K bit per line as *ECC-K*. Supposed the memory has 16 million lines, the probability of line failure must be well below 1 in several tens of million to get a low probability of system failure. To achieve our target system failure probability of 1 in a million, we will need to provision the system with

ECC-5. However, to prevent the system from soft-errors and from the infrequent episode of few bits changing retention time intermittently, we also deem it necessary to provision the line with an extra ECC code for soft-error protection. Therefore, to reliably operate the system with a refresh rate of 1 second, the system needs to provision with ECC-6 code per line.

D. Drawbacks of Strong Multi-bit ECC codes

While strong multi-bit error correction codes, such as ECC-6, can reduce refresh power, they suffer from two major overheads: storage and latency. The storage overhead required for error correction is linearly proportional to the number of errors that we want to correct. Therefore, ECC-6 will require six times as much storage overhead as ECC-1 for the same granularity of data-bits that we want to protect with the code. To make our solution practical, we would like to reduce the storage overhead required to implement ECC-6.

The second overhead of strong multi-bit ECC is the latency associated with encoding and decoding the line. While single bit error correction is typically implemented with Hamming codes, strong multi-bit ECCs are implemented with BCH codes, involving complex steps of syndrome decoding. The latency associated with decoding of strong multi-bit errors typically ranges in few tens of cycles [5]. The decode latency is in the critical path of memory access; therefore it increases the effective memory latency and degrades performance.

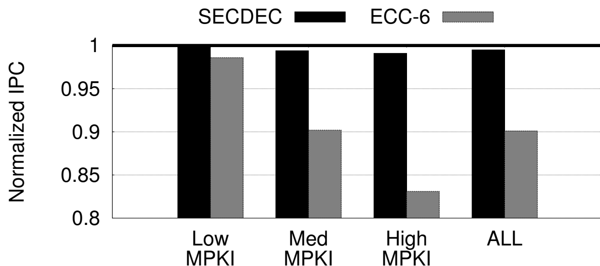

Fig. 3. Performance Impact of Decoding Latency of Error Correction compared to a system that has no error correction. SECDED incurs negligible performance impact, but ECC-6 degrades performance significantly (methodology in Section IV).

Figure 3 shows the system performance of ECC-1 (SECDED) and ECC-6 compared to a system that does not perform any error correction. On average, ECC-1 codes have negligible impact on performance (less than 1% on average) as the decoding latency is only a couple of cycles. In our system, we assume that ECC-6 decoding takes 30 cycles, which degrades performance significantly up to 22%, and on average 10%.

III. MORPHABLE ECC

We want to save refresh power by employing strong ECC, while avoiding the performance impact of decode latency for strong ECC. To get these conflicting benefits, we exploit the observation that the refresh power contributes to a smaller fraction of memory power (and system power) when the system is actively used. However, during long idle periods, the refresh power contributes significantly to the memory power (and system power). We can get both high performance and low refresh power if we optimize the ECC separately for active mode and idle mode. During active mode, it is preferred that memory be decoded with weak-ECC to avoid the latency impact. Whereas, during idle mode, memory is not accessed so it is desirable to use strong ECC and save refresh power. Based on this insight, we propose *Morphable ECC (MECC).*

A. MECC: Concept and Overview

MECC consists of two levels of ECC codes: Strong ECC and Weak ECC. Strong ECC is chosen to optimize for refresh power. There is no requirement for weak ECC, except that it has low latency overheads. One can substitute no ECC for weak ECC. However, to ensure robustness against soft errors, we use SECDED. For strong ECC, we use ECC-6. Figure 4 captures the overview of working of MECC.

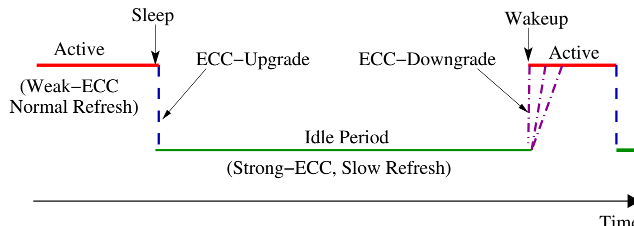

Fig. 4. Overview and Working of Morphable ECC.

During the active mode, the system uses normal refresh rate and accesses memory with the latency of weak ECC. When the system becomes idle, memory is converted from weak ECC to strong ECC. We call this conversion from weak ECC to strong ECC as *ECC-Upgrade*. Once the memory is upgraded, the memory is transitioned into self refresh mode, but with a period of 1 second (instead of 64ms). Thus, in the idle period, the refresh operations get reduced by 16x. When the system is activated, the memory refresh rate is increased to 64ms. The first access to a line gets the line in strong-ECC state; however this line is then converted to weak-ECC state and written back to memory. The conversion from strong ECC to weak ECC is referred to as *ECC-Downgrade*. This conversion ensures that subsequent memory request to the same data block would not pay the latency overheads of strong ECC; therefore in the active mode the common-case latency overhead becomes that of weak ECC. Note that lines undergo ECC-Downgrade on a demand basis, which avoids wasteful transitions of ECC status for unused lines.

B. MECC: Design

Figure 5 shows an overview of the system that supports MECC. MECC requires that the processor chip contains encoders and decoders for both weak ECC and strong ECC. The DRAM module must support the storage overhead required for both weak ECC and strong ECC as well. When a line is accessed in the active mode, the memory controller needs to know which decoder should be employed to decode the line. To provide this information, the line is appended with status bit called *ECC-mode*. When ECC-mode is 0, the line is decoded with weak ECC and when it is 1, the line is decoded with strong ECC.

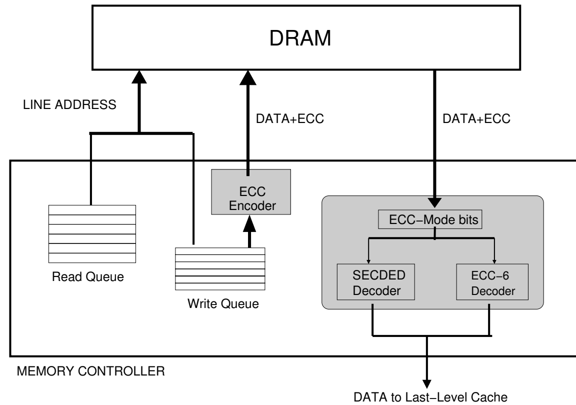

Fig. 5. System Support for Morphable ECC (Newly added parts for error correction are shaded.) Figure not to scale

When the memory controller encounters a line that has ECC-mode bits set to strong ECC, it generates the contents of weak-ECC and writes the line back to memory with the ECC corresponding to weak ECC and marking the ECC-mode bit as such. Note this ECC-Downgrade is not in the critical path of memory access.

When the system becomes idle, the OS can turn off the processor chip (after flushing the caches), and send an self refresh signal to memory and subsequently grounding the *clock* to the memory. When this happens, the memory lines are read, and converted to ECC-6, and the ECC-mode bit associated with the line is marked as such. When the ECC-Upgrade process finishes, the memory is put into self refresh mode, and uses long refresh intervals.

MECC also relies on support from the memory device to change the refresh frequency in idle mode, by simply having an internal counter, which would be incremented on each refresh pulse, and the outgoing refresh pulse is sent to the DRAM array only on counter overflow. The size of the counter will then modulate the refresh frequency. We assume that such support will be available from future DRAM module to optimize refresh power. A 4-bit counter is required to increase the refresh rate from 64ms to 1 second.

C. ECC Support for Mobile Memories

MECC relies on having the ECC code (for both SECDED and ECC-6) stored in the DRAM arrays. Current mobile memories (and even the commonly used Desktop memories) are typically not equipped with ECC support. However, as a recent paper [15] from Intel and Samsung shows, that to tolerate the failure modes at smaller technology nodes, even the commodity memories will need to be provisioned with the ECC support. Therefore, we assume that our baseline mobile memory system is supported with SECDED using the (72,64) code. While the (72,64) code requires that the number of x8 chips in Desktop DIMMs be increased from 8 to 9, having an extra chip in mobile memories is harder as such memories typically have only two x32 chips (so adding an extra chip would incur 50% storage). We observe that the (72,64) code be supported easily even for mobile memories by having x36 chips, or by having a burst length of 9 (instead of 8) to obtain the extra 8 ECC bits required for the 64 bit of data. For the remainder of the paper we will assume that our baseline mobile memory system supports SECDED at a word granularity. We show how MECC can be implemented on such a memory system without requiring any additional storage.

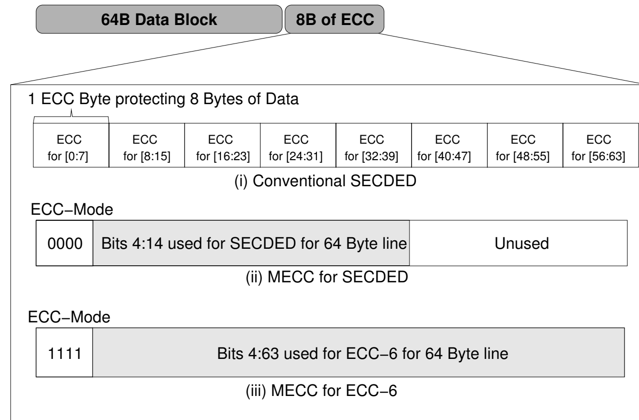

Fig. 6. Morphable ECC Design on ECC memory (i) Conventional SECDED operating at 8 byte granularity (ii) MECC storing the line with Weak ECC, operating at 64 bytes (iii) MECC storing the line with ECC-6, operating at 64 bytes

D. Reducing Storage Overhead for MECC

Instead of strong SECDED on 8-byte granularity (which is done for traditional reasons, owing its origins to when cache lines were small), we propose to have both SECDED and ECC-6 on a line size granularity (64 bytes). For SECDED, we would need 11 bits, and for ECC-6 we would need 60 bits (61 bits if we want to implement 6-bit Error Correction and 7-bit Error Detection). Note that, we would not need storage for both SECDED and ECC-6 at the same time, as the line can either be using SECDED or ECC-6. Thus, we need 60 bits for ECC. The traditionally used (72,64) code provides 8 ECC bits for 64 bits of data, which amounts to 64 bits of ECC for a 64 byte cache line. We propose to use all the 64 bits in conjunction to repair the entire line. The first four bits in this 64 bit ECC space would indicate ECC-mode, implemented with 4-way redundancy for fault tolerance. The remaining 60 bits are used for either SECDED or ECC-6, as shown in Figure 6. Thus, MECC can be implemented easily with a memory system that supports the traditional (72,64) code, without the need of any modification to existing storage array.

During idle mode, ECC-mode bits might flip at the raw bit error rate $10^{-4.5}$, and later in active mode, the memory controller would fail to identify the ECC-mode, causing an unreliable system. To solve this, ECC-mode bit is duplicated four times to tolerate errors. All the data bits and ECC-mode bits are covered by the ECC-6. When there is a mismatch in the replicated copies of ECC-mode bits, we try both SECDED and ECC-6 decoder. The one which gives correct result indicates the ECC-mode of the current data block. Therefore, the ECC-mode bits are well-protected to ensure the correct identification of ECC modes.

E. Complexity of ECC Encoder and Decoder

MECC relies on ECC encoder and decoder to tolerate different levels of bit error rate. SECDED is a widely adopted code in industry, so we do not describe the details. Multi-bit error correction takes longer to decode, and larger area than SECDED. Typically, SECDED is based on Hamming Code , while ECC-6 uses BCH code [5].

BCH code is a class of cyclic error-correcting codes, which is able to correct random multi-bit errors [16, 17]. Let's assume the input data d is a k-bit data. To correct t errors and detect t+1 errors, BCH will require a code word of t*m+1 bits, where $d < 2^m - 1$.

ECC Encoder: To encode the data, BCH encoder generates the code word r and concatenates r with data d by simply multiplying d by the generator matrix G, which is a predefined matrix, XORing the data to generate the code. Therefore, the encoding latency for both SECDED and ECC-6 is a few XOR gate delay, which can be completed in one 1.6GHz-processor cycle [5].

ECC Decoder: The decoding logic can correct and detect any errors in the concatenated data and code word. We calculate the latency and area cost from [18], which gives that both the decoder latency and area complexity in proportional to t, when the data length remains the same. We estimate that SECDED would incur a logic overhead of approximately 3K XOR gates and 2 cycle latency. In general, one can trade-off latency and area for decoding multi-bit ECC. Similar to [5], we estimate that ECC-6 would incur a logic overhead of about 100K-200K gates, and latency of 30 cycles.

The latency of multi-bit error correction is dependent on the number of errors encountered in the line. Fortunately, even with a BER as high as 10^{-6}, the likelihood of encountering a line with two or more errors is less than 1 in a million, so the performance impact of the slow error correction (ECC-2 or beyond) is negligibly small ($< 0.001\%$). Our evaluations assume an "average" latency of 30 cycles for ECC decoding and correction. We conduct a sensitivity study on ECC latency in Section V.

IV. Experimental Methodology

A. System Configuration

We use the memory system simulator USIMM [19] from the recently conducted Memory Scheduling Championship [20]. USIMM models DRAM system in detail, enforcing the various timing constraints. We modify USIMM to conduct a detailed study for error correction, including latency and power issue. When the memory request is serviced by the memory system, the latency depends on the error correction scheme.

TABLE II. Baseline System Configuration

Processor	in-order core, 2-wide retire, 6-stage depth
Cache	1MB, 64B cache line
Memory	1GB LPDDR, 200MHz bus speed, double data rate, 1 channel, 1 rank, 4 banks, 16K rows and 1K columns

We model a system consisting of an in-order core system operating at 1.6GHz. The memory system is 1GB configured operating at 200MHZ, as same as the LPDRAM provided by Micron [21]. We assume when the memory system is idle, the DRAM devices are in self refresh mode and the processor is turned off. The parameters of system configuration are shown in Table II. The baseline in our experiments uses aggressive power down saving scheduling, in which the scheduler issues a power-down command whenever it is possible. For ECC decoder latency, SECDED takes 2 cycles, while ECC-6 takes 30 cycles.

B. Workloads

We use 28 benchmarks [1] from the SPEC2006 suite for our study. Although SPEC2006 is not designed for mobile platforms, it has several meaningful benchmarks, such as image rendering (povray), speech recognition (sphinx3), compression (bzip2), and video processing (h264). Furthermore, for our studies we simply need memory access patterns to determine the sensitivity of performance to different ECC configurations. We assume that each workload is executed for 4 billion instructions on a single-core processor, after skipping 10 billion instructions. We classify the benchmarks into three categories based on MPKI: (a) Low-MPKI (MPKI $<$ 1 (b) Med-MPKI (MPKI between 1 and 10) and (c) High-MPKI (MPKI $>$ 10). To provide insights in our analysis, we will refer to this classification. Table III shows average key characteristics of the workloads used in our study, including Misses Per Kilo Instruction (MPKI), Baseline IPC (without error correction latency), and memory footprint. Footprint is calculated as the number of unique 4KB pages touched by the workload slice. Note the average IPC is 0.72, which translates into an execution time of approximately 5.5 seconds for the active period of the running application.

TABLE III. Benchmark Characterization.

Name	IPC	MPKI	Footprint(MB)
Low-MPKI	1.514	0.3	26
Med-MPKI	0.887	4.7	96.4
High-MPKI	0.359	23.5	259.1

C. Power Calculator

The power consumption of the memory devices are calculated based on the power parameters shown in Table IV. We use the Micron DRAM power calculator to derive the power statistics [22, 23]. We also calculate the energy dissipated by the ECC decoders and encoders in the active mode. Compared to the typical processor in mobile devices and memory access, ECC decoder and encoders consume negligible power (ECC-6 decoder consumes approximately 40 pJ to decode a line, while reading a line from memory requires 12 nJ).

[1]The footprint of mcf is 1.4GB, which makes it unusable for studying a memory system of 1GB, therefore we do not include mcf in our studies. If we ignore page fault latency, then MECC gets performance within 2% of baseline with no ECC, so including mcf does not have any impact on the average performance of our proposal.

TABLE IV. POWER PARAMETERS FOR OUR MEMORY SYSTEM

Parameters	Values	Description
VDD	1.7 V	Operating Voltage (Volts)
IDD0	95 mA	1 bank active precharge current
IDD2P	0.6 mA	Precharge power-down standby current
IDD3P	3 mA	Active power-down standby current
IDD4	135 mA	Burst read/write: 1 bank active
IDD5	100 mA	Auto refresh
IDD8	1.3 mA	Self refresh

D. Figure of Merit

As we have dual objective of both high performance and low power, we will use the following metrics as figure of merit in the evaluation.

Performance: Performance is measured in terms of Instruction Per Cycle (IPC) for each system and normalized with respect to the baseline system that does not incur any error correction latency. Thus, the difference in performance is mainly from the latency overheads of error correction.

Power Savings in Idle Mode: The power consumption in the idle mode is based on refresh power and background power, shown in Equation 1[7]. For the energy evaluations in idle mode, we assume that the activity of daemon processes (such as Bluetooth, sync, etc.) is negligibly small and hence we do not include the energy from these activities in both the baseline as well as our proposal. While this assumption generally holds true, it may not hold in extreme cases when there is a pathological daemon process that continues to consume significant energy even when the device is idle. Examples of such undesirable daemon processes includes *mm-qcamera-daemon* [24] and *Unified-daemon (EUR)* [25], which have been known to quickly drain the battery even when the device is idle. The software community has looked at averting such daemons using OS patches. For unpatched systems that run such pathological daemons, we can simply assume that the devices is always active and thus offers no scope for idle power reduction.

$$
\begin{aligned}
IdleP_{MECC} &= P_{MECCRefresh} + P_{Other} \\
&= \frac{T_{Original}}{T_{MECC}} \cdot P_{OrignialRefresh} + P_{Other}
\end{aligned} \tag{1}
$$

Note that a more precise evaluation of activity for such daemon processes can be done using a hardware infrastructure that can (a) measure the frequency of daemon processes during idle period (b) measure the duration of each such daemon process (c) generate the trace of memory access stream of each such process. Furthermore, the study will need to be done over several users as the daemon activity is typically user dependent. Doing such an extensive study is out of the scope of this paper.

Power and EDP in Active Mode: For active mode, we must take into account both performance and power; otherwise, a scheme that degrades performance may seen to be beneficial from purely a power savings perspective. So, we use Energy Delay Product (EDP) as a figure of merit and calculate it as shown in Equation 2.

$$EnergyDelayProduct = DissipatedEnergy \times ExecutionTime \tag{2}$$

V. RESULTS AND ANALYSIS

A. Impact on Performance

The latency incurred in performing error correction increases effective memory latency, and degrades performance. We compare the performance of different error correction schemes in active mode of operation. Figure 7 shows the IPC of SECDED, ECC-6 and MECC, normalized with respect to the baseline which does not incur any latency overheads from error correction. The bar labeled *ALL* is the geometric mean over all workloads.

The performance impact of SECDED is quite small across all the workloads. On average SECDED has 0.5% slowdown compared to no error correction. The performance impact of ECC-6, however, depends on the memory behavior of workloads. For Low-MPKI benchmarks, the performance impact compared to no error correction is quite small for ECC-6. For MED-MPKI and High-MPKI, using ECC-6 causes significant performance degradation. The slowdown for *libquantum* is as high as 21%. On average, ECC-6 causes a slowdown of 10%. In contrast, MECC bridges the performance gap between SECDED and ECC-6, with performance very close to SECDED. The average slowdown with MECC is only 1.2%. Thus, the performance of MECC is within 1% of SECDED, whereas ECC-6 of 10% performance degradation is significantly worse.

B. Power Saving in Idle Mode

We employ strong ECC in idle mode to reduce refresh power. For both ECC-6 and MECC, we assume that the refresh rate is reduced from 64ms (in baseline) to 1 second. This translates to a linear reduction in refresh power. Figure 8(left) shows the refresh power of ECC-6 and MECC, normalized to the baseline system. Both ECC-6 and MECC reduce the refresh power by 16x.

However, refresh power is only a portion of memory idle power. When memory is idle it still consumes background power. Figure 8 (right) shows the breakdown of idle power in terms of refresh power and background power, normalized to the baseline. Both ECC-6 and MECC reduce refresh power by 16x, and the overall power reduction is about 43% given that refresh power accounts for only half the idle power. Thus, MECC and ECC-6 are effective at reducing idle power by almost 2X.

C. Power and Energy in Active Mode

We also analyze the power characteristics of different schemes when the system is active. Figure 9 shows the power, energy consumption, and energy-delay product for baseline, SECDED and MECC. MECC has approximately 1% higher

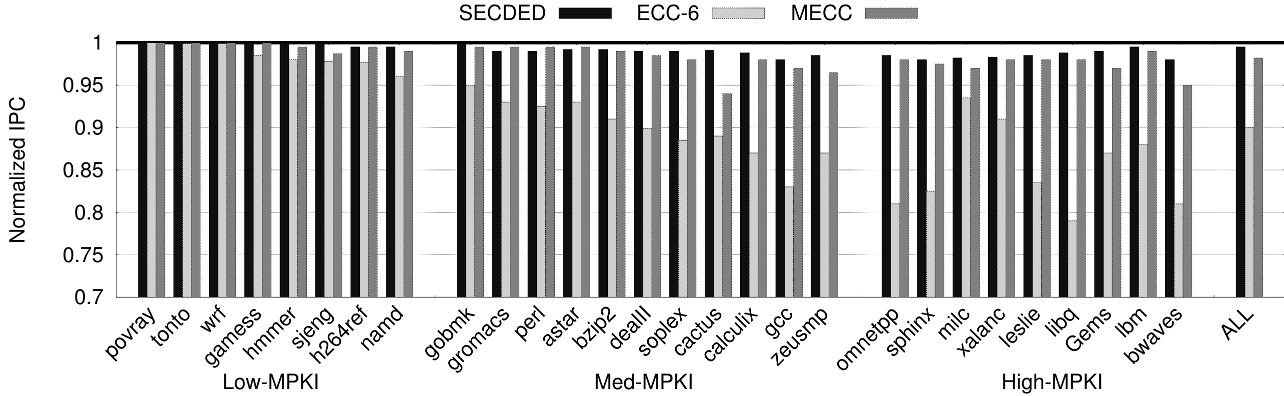

Fig. 7. Performance Comparison of SECDED, ECC-6, and MECC. All performance numbers are IPC values normalized to the baseline that does not incur any error correction latency.

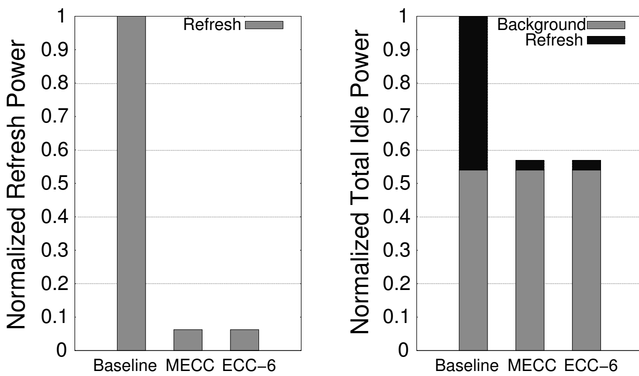

Fig. 8. Comparison of (left) Refresh Power and (right) Total Idle Power of Memory System

power than baseline, which is because of the extra memory traffic. ECC-6 seems to have lower power, but that is primarily because it takes 10% more time to execute the workload. When energy consumption is considered in active mode, all the three schemes are similar. In terms of EnergyDelayProduct (EDP), MECC is similar to baseline whereas ECC-6 is 10% higher. Thus, MECC provides similar energy consumption as ECC-6 but a better energy-delay product.

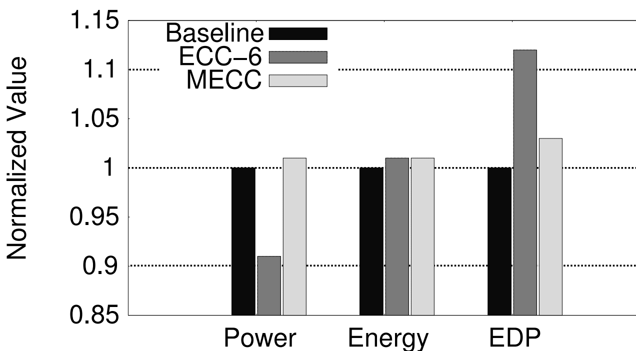

Fig. 9. Power and Energy Metrics in Active Mode

D. Impact on Overall Memory System Energy

The total energy consumption of the memory system is determined by both the active power and idle power. Our proposal reduces the idle power by 2X while having similar active power. While the idle power of a typical system is typically much lower than active power (20X or more lower), the idle period lasts for a longer time. Therefore, the overall battery drain from idle periods can still be significant. To do the overall energy analysis, we assume that the idle period accounts for 95% of the system time, which is in accordance with the recent user studies on smartphone activity [3].

Figure 10 shows the total energy consumption for our baseline, divided into energy from active use and from idle periods. The energy spent in idle periods accounts for approximately one-third of the total system energy consumption. MECC reduces the idle power by almost 2X, thus reducing the overall memory system energy consumption by 15%.

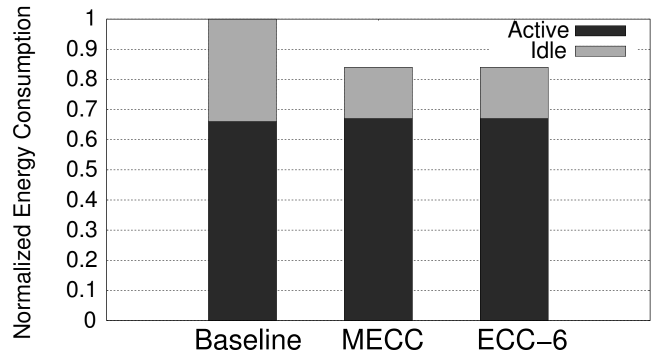

Fig. 10. Total Energy Usage in Memory System

E. Sensitivity to ECC Decode Latency

We have used a latency of 30 cycles for ECC-6 decode. In general, there is a design trade-off between area of decoder and latency. We conduct sensitive study of the ECC decoder latency, varying from 15 to 60 processor cycles. Figure 12 shows the performance impact of this latency variation on ECC-6 and MECC. The performance impact of MECC is

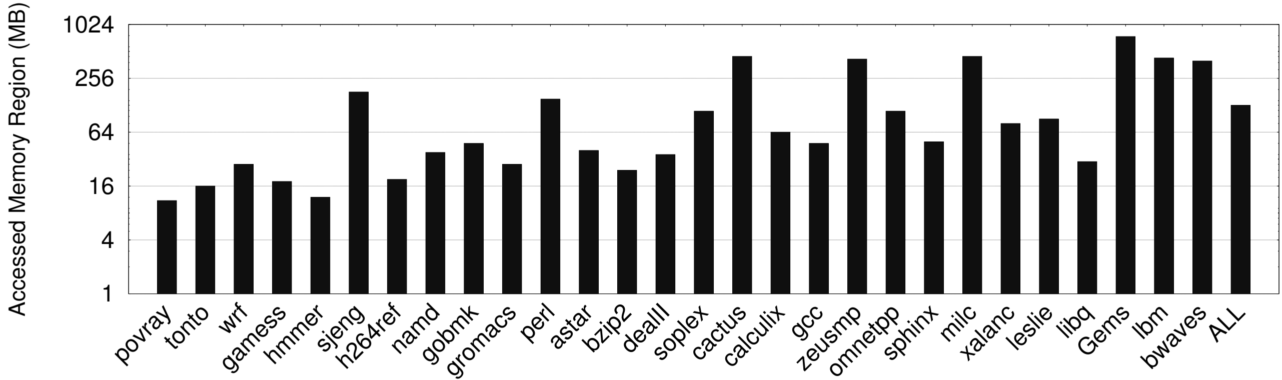

Fig. 11. Effectiveness of Memory Downgrade Tracking. The size of memory touched by the application, as estimated by MDT (1K regions). Note Y-axis is in log scale.

not very sensitive to decoder latency. For example, even with the decoder latency of 60 cycle, MECC is still able to limit the slowdown to within 2% of the baseline with no error correction. The main reason behind is that MECC only pays the long decoder latency once; after the data block has been accessed, it is no longer stored with the strong ECC but weak ECC, reducing the latency overhead for future access. However, the performance impact of ECC-6 is quite sensitive to decode latency, becoming 18% when the decoder latency is 60 cycles. Therefore, MECC can allow the designer to implement multi-bit decoding with simpler hardware with low area overhead, and still not have a significant slowdown.

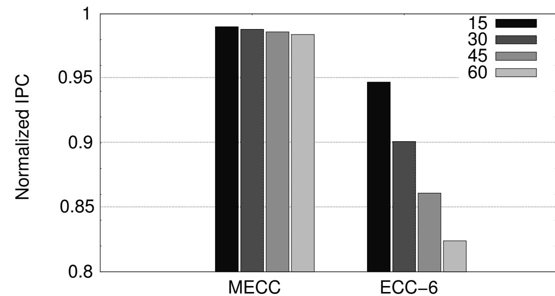

Fig. 12. Sensitive Study of ECC Decoder Latency

F. Transition Time for MECC

Our measurement of the performance is on the overall slowdown for the 4 billion instruction slice for each benchmarks. It is worthy of understanding how long it takes for MECC to reach the performance of SECDED. Figure 13 shows the normalized IPC with different instructions slices. The x-axis are the number of executed instructions, which are 0.5 billion, 1billion, 2 billion, 3 billion, and 4 billion instructions.

Up to 1B instructions, MECC is slower than the baseline by 2%; however, as the workload continues to execute, the gap shrinks, and MECC reaches at 1.2% after 4B instructions. Thus, the long latency of ECC-6 decode and additional writes due to ECC-Upgrade happen primarily in the first 1 billion instruction (first 1 second) of the application execution, and after that the application executes mainly with SECDED.

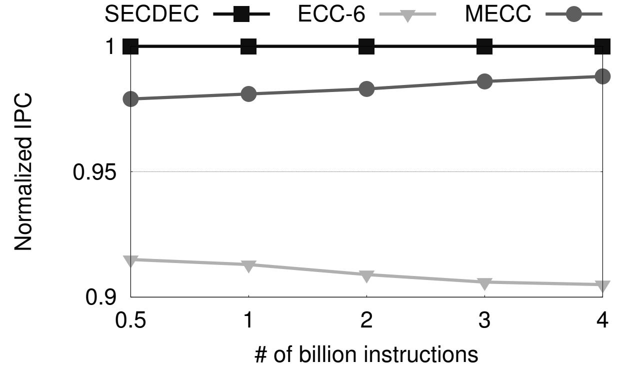

Fig. 13. Effectiveness of MECC for Different Program Length. The performance impact of MECC becomes similar to SECDED after approximately the first one second of program execution.

VI. Enhancements for MECC

The MECC design described thus far is a simple one, in that it always performs ECC-Downgrade on every access in active mode, even if the application performance is not limited by memory. Furthermore, when the system enters idle mode, it performs ECC-Upgrade on the entire memory, even if the majority of the memory was not accessed since the last idle period, and hence was already protected with ECC-6. This section describes simple and effective hardware techniques that can further increase the efficacy of MECC.

A. Region-Based Memory Downgrade Tracking

When the system transits to idle mode, MECC needs to ensure that *all* of the memory is converted to ECC-6, before reducing the refresh rate to 1 second. So, MECC performs ECC-Upgrade for the entire memory before reducing refresh rate. Given the memory has 16 million lines, it will take 640 million cycles, or 400ms to perform ECC-Upgrade of entire memory. While, this is short period given that the typical idle period lasts for several minutes, we would still like to make the ECC-Upgrade more efficient. The observation that makes this possible is that not all memory gets accessed during the active period, so if we can efficiently track the memory that

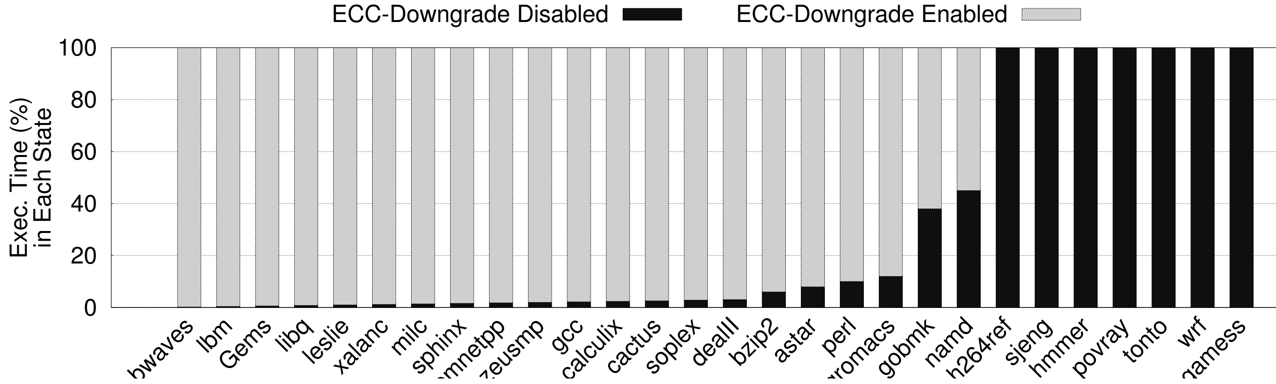

Fig. 14. The fraction of execution time for which ECC-Downgrade remains disabled for each benchmark (using SMD with treshold of MPKC=2)

was accessed (and hence went through ECC-Downgrade), then we can do ECC-Upgrade for only those regions of memory. We propose *Memory Downgrade Tracking (MDT)* to make ECC-Upgrade more efficient. MDT is implemented with a table where each entry is a single bit and corresponds to a memory region, as shown in Figure 15.

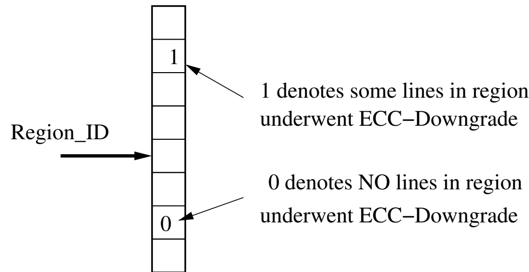

Fig. 15. Memory Downgrade Tracking (Region ID is obtained from top few MSB bits of line address).

When a line undergoes ECC-Downgrade, the MDT entry corresponding to this region gets set. When the memory undergoes ECC-Upgrade, only the lines, for which the corresponding memory region is set to 1, goes through ECC-Upgrade (otherwise the lines continue to retain their ECC-Upgrade status from the past idle period). The MDT table is reset after the ECC-Upgrade process finishes. We evaluate a default MDT configuration that has 1K entries (128 bytes storage), so each MDT entry tracks a memory region of 1MB, given that our memory size is 1GB. Note that 128 byte storage is not a huge overhead, and hence can be put in the memory controller, since memory controller is the center of making ECC-upgrade and ECC-downgrade operations.

In Section IV we provide the memory footprint of all the benchmarks we use in our study. On average the memory footprint of all the benchmarks is 128MB, which is 8x smaller than the 1GB memory system. Although memory access may not be continuous throughout the physical memory space, the memory region that a program would use is still much smaller than the entire memory. Figure 11 compares the memory capacity tracked by MDT, if it has 1K entries. Note that Y-axis is in log scale. With MDT, we can reduce the size of memory that undergoes ECC-Upgrade by almost 8x, reducing the time to perform ECC-Upgrade from 400ms to 50ms. Moreover, the energy spent in the encoder is also 8x lower as a result of reduced number of ECC coding. Thus, a hardware table of 128 bytes is sufficient for MDT.

B. *Efficiently Handling Periodic Processes in Idle Mode*

ECC-Downgrade avoids the long latency overheads of decode operations for strong ECC. However, it is not necessary to do ECC-Downgrade for all the applications. For example, even when the mobile device is idle, it might still perform periodic activities such as bluethooth check, network inerrupts etc. Fortunately, such activities are short and have a very small footprint. Furthermore, a small slowdown may be acceptable for such non performance oriented workloads. For our proposed MECC implementation though, such events can lead to frequent transitions of ECC-Upgrade and ECC-Downgrade, which may ruin the benefits from MECC. For such short, non-memory intensive applications, we can tune MECC to avoid ECC-Downgrade. As memory performance is not critical for such tasks, MECC can still use a refresh rate of 1 second even in active mode, and keep the entire memory protected with ECC-6.

We propose to extend MECC to avoid ECC-Downgrade during active mode, when memory performance is not critical for overall system performance. We call this *Selective Memory Downgrade (SMD)*, as ECC-Downgrade decision is dependent on the memory intensity of the workload. SMD is implemented by periodically tracking the memory activity of the workload. When the processor exits from idle state, ECC-downgrade is disabled, and the refresh interval remains at 1s. Every 64ms (approximately 100 Million cycles), we track the number of memory accesses generated by the workload in the previous time quanta of 64ms. If the memory traffic (measured as Miss Per Kilo Cycle, or MPKC) is greater than a predefined threshold, ECC-Downgrade gets enabled, and all subsequent accesses to memory perform ECC-Downgrade. Thus, for memory intensive workloads, SMD will enable ECC-Downgrade (to reduce memory latency), whereas for not memory intensive workloads SMD will disable ECC-Downgrade (and save power). The proposed SMD implementation requires only two registers to record the number of memory accesses and to track the time since last check.

To understand the effectiveness of SMD, we examine the time at which ECC-Downgrade gets enabled during the program execution. Figure 14 shows the fraction of execution time for which ECC-Downgrade remains disabled for each workload, given a threshold of MPKC equal to 2. Note, several benchmarks, such as, $povray$, $tonto$, wrf, $gamess$, $hmmer$, $sjeng$, and $h264ref$, do not enable ECC-Downgrade through the entire duration, thus optimizing for refresh power even during active mode. The average performance with SMD is within 2% of a baseline that does not perform error correction.

VII. Related Work

Several studies have looked into reducing the refresh power. We describe the work that is most closely related to ours, comparing and contrasting when appropriate.

A. Tolerating Refresh Errors in Software

A recent proposal, *Flikker* [7], describes a hardware-software co-operative mechanism that trades-off data integrity for refresh power saving. The memory is divided into critical-region and non-critical region. The critical region is refreshed at the normal rate, whereas the non-critical region is refreshed at much lower rate. The software is modified to ensure that data structures that influence correctness of program output are kept in critical region and data structures that are resilient to errors are kept in non-critical region.

There are three key differences between Flikker and MECC. First, Flikker requires programmer to change the source code, which may not always be possible, whereas MECC is purely a hardware based technique that is useful for existing programs as well. Second, a system with Flikker is vulnerable to data errors and hence it is useful only for class of applications that are inherently resilient to data errors, whereas MECC does not have this restriction. Therefore, MECC does not compromise program correctness to save power. Third, Flikker still has a sizable region that is deemed critical which ends up determining the effective refresh rate (akin to Amdahl's law). For example, if one-fourth of memory is refreshed at a rate of 1 and three-fourth at a rate of 1/16, the effective rate is still approximately 1/3. Whereas, MECC can provide an effective refresh rate of 1/16 for entire memory in idle mode. Thus, MECC is more effective at reducing refresh power, and it obviates the programmer effort to partition data into critical and non-critical regions.

B. Retention-Aware DRAM Optimizations

The power overheads associated with DRAM refresh can be reduced by modulating the refresh rate depending on the retention time of the memory row. This can avoid the worst-case refresh rate for all the memory rows (or pages), when only a very small fraction of memory bits have low retention time. This is the idea behind three prior proposals: *RAPID* [13], *RAIDR* [14], and *SECRET* [26].

RAPID is a software technique that allocates memory pages depending on the retention characteristic of each page. Pages with low retention are either disabled from OS pool, or are given low priority of allocation. The refresh rate of the system is thus determined by the allocated page with the lowest retention time. RAIDR classifies memory rows into low retention and high retention portions using runtime profiling. It uses low refresh rate for rows with high retention time, and normal refresh rate for the rows with low retention time. SECRET uses error correction code to save refresh energy. Their proposal requires an off-line investigation to identify memory cells with retention errors given a target error rate, and corrects the errors when lowering refresh rate. However, to reduce the refresh rate significantly, it requires the use of strong error correction, and thus always incurs the performance overhead of strong error correction.

All three proposals rely on the premise that the retention characteristics of a cell do not change. While this may be true for most cells, a small fraction of cells are known to exhibit a phenomenon called *Variable Retention Time (VRT)* [11][27][28], whereby a cell with high retention time can randomly turn into a cell with low retention time. Therefore, such not retention-aware schemes are vulnerable to data errors in practice [11]. MECC, on the other hand, does not rely on explicit retention characteristics of each cell. Instead it allows for a large number of cells to fail randomly in the memory space and tolerates such failures with strong ECC code. Nonetheless, the two concepts of multi-rate refresh and MECC are orthogonal, and can be combined for more efficient and effective solutions.

C. Multi-bit ECC for Tolerating Errors

MECC uses strong ECC (ECC-6) to tolerate refresh errors, and changes the ECC strength depending on the activity rate of the system. Alternative means of reducing the latency penalties of multi-bit codes have been proposed. The most closely related work is *Hi-ECC* [5]. To reduce cache leakage power in a storage efficient manner Hi-ECC keeps the strong ECC code over a granularity of 1KB lines. Thus, it suffers from the problem of significant overfetch, and read-before-write requirements. Furthermore, some of the techniques used to optimize Hi-ECC, such as cache line disable, are not easily applicable to main memories. Disabling an arbitrary cache line would not alter software correctness; however having "holes" in memory space necessitates special handling from OS to ensure correctness.

Yoon et al. [29] proposed *Virtual and Flexible ECC*, which allows flexibility in error correction level across memory space. Rather than using uniform error correction across the entire memory space, it allows the user to specify stronger levels of ECC for high-priority applications, and weaker levels of ECC for low-priority applications. However, unlike MECC, it does not modulate the error correction level depending on system activity. Furthermore, Virtualizing the ECC space has the advantage that one can still use commodity Non-ECC DIMM to support arbitrary error correction schemes; nevertheless, this comes at the expense of two memory accesses - one for data and the other for ECC. Nonetheless, MECC is compatible with both Virtual and Flexible ECC, as these concepts are orthogonal.

VIII. Summary

The problem of reducing idle power has become important for emerging computing platforms such as smartphones, as it often dictates the usable duration for these "mostly idle

but instantly active" devices. One of the significant sources of energy consumption during idle periods is the refresh power in memory system. We investigate strong multi-bit error correction for reducing refresh operations. Unfortunately, the high latency of multi-bit ECC results in increased memory access latency and lower performance, making it less appealing for practical implementations. We exploit the observation that we can get high performance as well as refresh power savings by modulating the ECC level depending on the level of activity in the system. and make the following contributions:

1) We propose *Morphable ECC (MECC)* that uses strong error correction (and slower refresh rate) in idle mode to save power, and weak ECC (and normal refresh rate) during active mode to optimize for performance.
2) We propose *Memory Downgrade Tracking (MDT)* to reduce the time taken to convert memory to strong ECC, and also the energy spent in ECC coding. MDT tracks the memory regions that get converted from strong ECC to weak ECC during active mode, and converts only these regions to strong ECC when system enters idle mode.
3) We present an extension of MECC that makes it possible to reduce refresh power even in active mode, for applications that are not sensitive to the memory system performance.

The access latency in common cases for MECC is dictated mainly by the latency of weak ECC (SECDED for example), which means the system can employ low-area high-latency implementations for encoders/decoders for strong ECC. On average, strong ECC causes a slowdown of 10% (as high as 21%); however, with MECC, the average slowdown is reduced to 1.2%. MECC reduces refresh power in idle periods by 16X and reduces idle power by 2X. While we have used ECC-6 as strong ECC and SECDED for weak ECC in our evaluations, the MECC scheme is useful for morphing between arbitrary levels of ECC, which trades off robustness with performance or power savings.

Acknowledgements

This work was supported in part by NSF grant 1319587 and the Center for Future Architecture Research (C-FAR), one of the six SRC STARnet Centers, sponsored by MARCO and DARPA. We thank our shepherd Helia Naeimi and the anonymous reviewers for their valuable feedback.

References

[1] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *the 2010 USENIX annual technical conference*, 2010.
[2] M. Gomony *et al.*, "Dram selection and configuration for real-time mobile systems," in *DATE-2012*.
[3] A. K. Karlson *et al.*, "Working overtime: Patterns of smartphone and pc usage in the day of an information worker," in *the International Conference on Pervasive Computing*, 2009.
[4] J. Flinn *et al.*, "Power and energy characterization of the itsy pocket computer," HP Labs, Tech. Rep., 2000.
[5] C. Wilkerson *et al.*, "Reducing cache power with low-cost, multi-bit error-correcting codes," in *ISCA-37*, 2010.
[6] Wikipedia: Comparison of smartphones.
[7] S. Liu *et al.*, "Flikker: saving dram refresh-power through critical data partitioning," in *ASPLOS-6*, 2011.
[8] M. Murphy, *Beginning Android*. Apress, 2009.
[9] Microsoft: Surface pro specification. [Online]. Available: http://www.microsoft.com/Surface
[10] K. Kim and J. Lee, "A new investigation of data retention time in truly nanoscaled drams," *Electron Device Letters*, 2009.
[11] B. L. Jacob, S. W. Ng, and D. T. Wang, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.
[12] B. Gu *et al.*, "Challenges and future directions of laser fuse processing in memory repair," in *the International Conference on Semiconductor Equipment and Materials International*, 2003, pp. 452–456.
[13] R. Venkatesan *et al.*, "Retention-aware placement in dram (rapid): software methods for quasi-non-volatile dram," in *HCPA-12*, 2006.
[14] J. Liu *et al.*, "Raidr: Retention-aware intelligent dram refresh," in *ISCA-39*, 2012.
[15] U. Kang *et al.*, "Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling," in *Memory Forum*, 2014.
[16] C. Chen *et al.*, "Error-correcting codes for semiconductor memory applications: a state-of-the-art review," *IBM Journal*, vol. 28, no. 2, March 1984.
[17] E. Fujiwara and D. K. Pradhan, *Error-Control Coding in Computers Systmes*. Prentice-Hall, Inc., 1989.
[18] R. T. Chien, "Cyclic decoding procedures for bose-chaudhuri-hocquenghem codes," in *IEEE Transactions on Information Theory*, vol. 10, no. 4, Oct 1964.
[19] Rajeev Balasubramonian and others, "Usimm: the utah simulated memory module," University of Utah and Intel Corp, Tech. Rep. UUCS-12-002, Feb. 2012.
[20] (2012) Memory scheduling championship (msc). [Online]. Available: http://www.cs.utah.edu/ rajeev/jwac12/
[21] *1Gb-DDR-Mobile-SDRAM-t68: Mobile Low-Power DDR SDRAM: Rev. G 9/11 EN*, Micron Technology Inc., 2010.
[22] *TN-46-12: Mobile DRAM Power-Saving Features/Calculations*, Micron Technology Inc, 2009.
[23] *TN-46-03: Calculating DDR Memory System Power Introduction*, Micron Technology Inc., 2010.
[24] mm-qcamera-daemon cpu usage. [Online]. Available: https://code.google.com/p/android/issues/detail?id=60058
[25] Unified-daemon (eur). [Online]. Available: http://skp.samsungcsportal.com/integrated/unified-daemons
[26] D.-Y. Shen *et al.*, "Secret: Selective error correction for refresh energy reduction in drams," in *ICCD-2012*.
[27] J. Liu *et al.*, "An experimental study of data retention behavior in modern dram devices: Implications for retention time profiling mechanisms," in *ISCA-40*, 2013.
[28] S. Khan *et al.*, "The efficacy of error mitigation techniques for dram retention failures: A comparative experimental study," in *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2014.
[29] D. H. Yoon and M. Erez, "Virtualized and flexible ecc for main memory," in *ASPLOS-15*, 2010.