Coded Data Storage for Dense Memory Architecture

ABSTRACT

This paper explores the use of coding theory in DDR and high bandwidth memory (HBM) system. Conventional DDR systems use timing optimization techniques around DDR protocol to improve the efficiency of accesses. In this work we propose increasing redundancy to help distribute access across dram pages. This will help in efficient retrieval of data. These schemes will be evaluated and optimized with several iterations. The aim is to find the best performing code design scheme given the performance requirement compared to alternatives. Our proposed scheme results in a decrease in the of the number of contended memory accesses between cores, therefore reducing the overall latency of the system. The reduction in the latency can be seen directly as an increase in the overall system performance. We show that with a memory overhead of 15%, we can enable 4 extra access to a bank while remaining within the given design parameters.

1. INTRODUCTION

In the era of disruptive development in the field of data science and machine learning, computing systems with high processing capacity are in huge demand. According to Moore's law, computing systems double their overall computation capability every 18 months. The processing power of a system is mainly determined by its compute capability and memory operations. Recent trends in computer architecture systems have shown that memory access speed is a major bottleneck in increasing the processing capability. This is due to the large amount of time required to transfer data between the processor and the memory. The research and industrial communities have improved data access capacity and speed through various innovations in integrated circuit design. However there still remains work to be done for further improvement in memory access efficiency. In this project, we propose a new, innovative solution to improve the access efficiency of next-generation memory systems such as high bandwidth memory (HBM) and hybrid memory cube (HMC). We propose a novel architecture with two different coding methods that uses redundancy in data storage to provide parallel memory access with low latency.

This paper presents solution to two main problems in computing systems. First is low access efficiency for memory read and write. The gap between the memory access speed and CPU speed is widening more than before [**?**]. The efficient integration of multi-cores combined with high speed architectures increases computation capability. However, slow memory fails to keep up with the request from the cores delaying the overall processing speed. The access latency to the shared memory increases due to contention among requests from computing cores. This results in formation of large access request queues waiting to be served by the slow shared memory. The current solutions to these problems in DDR environment is limited to improving command scheduling and clever data addressing. This project will develop a different solution where by introducing compressed redundancy in data, memory accesses are spread across banks. Our unique approach uses ideas from coding theory to achieve optimal compressing and distribution. We will develop this solution for general memory architecture and implement it for DDR and High Bandwidth Memory (HBM) protocols.

2. BACKGROUND AND MOTIVATION

Memory accesses are the most vital part of any program. A program is intrinsically made up of loads and stores to the memory. It is for this reason that it is most beneficial to keep memory accesses latency as short as possible. Long-latency memory accesses occur when there is an instruction miss in the last level cache. This leads to an access to a shared memory having a long latency. The system then issues an access to the shared memory and waits for it to return. This creates a stall in the processor.

In case of multi-core processor architecture, these long-latency accesses to the shared memory encounter additional delay due to interference from accesses by the additional cores. This results in a queue of accesses waiting to be served by the memory. Figure **??** shows a common multi-core architecture where N processor cores use a shared memory consisting of M banks. Requests from each core go to the memory controller, which then arbitrates and issues requests to the memory. Because the memory controller has parallel access to all M banks, a queue forms for each individual bank request. The queues are then served every cycle and the acknowledgement with data (in the case of a read) is sent to the processor. Requests from each core go to the memory controller, which then arbitrates and issues requests to the memory. Because the memory controller has parallel access to all M banks, a queue forms for each individual bank request. The queues are then served every cycle and the acknowledgement with data (in the case of a read) is sent to the processor.

In the scenario where multiple cores request access to the

memory locations which belong to the same bank, the memory controller schedules these request for the queues of that bank. This contention between cores for memory accesses from the same bank is known as a bank conflict. When a conflict occurs, requests are served sequentially from the queues, increasing the latency for the requests later in the queue. As the number of bank conflicts increase, the latency for individual memory accesses to the same bank also increase, resulting in increased latency for the entire system.

In this paper, we try to solve the problem of concentrated accesses to a particular bank by normalizing it across several banks. The problem is solved by using ideas coding theory techniques to create redundancy across banks, increasing the number of parallel accesses per cycle. The queue build up on a bank is serviced through parallel access to several additional banks, known as parity banks. We use two different coding algorithms to two encode the memory storage to achieve different trade-off options between memory overhead and access latency. This results in a decrease in the of the number of contended memory accesses between cores, therefore reducing the overall latency of the system. The reduction in the latency can be seen directly as an increase in the overall system performance. We show that with a memory overhead of 15%, we can enable 4 extra access to a bank while remaining within the given design parameters.

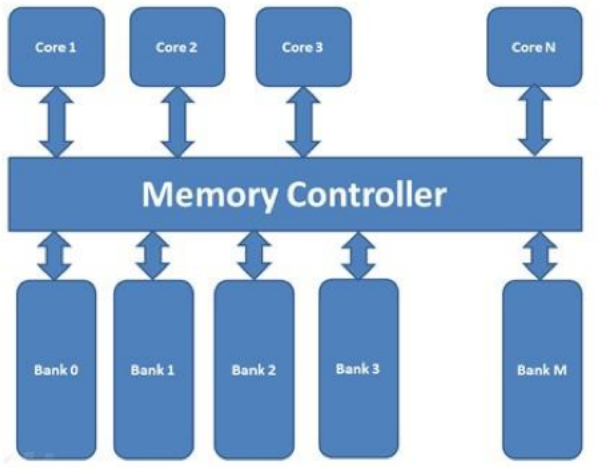

Figure 1: Multi-core Architecture with shared memory

3. CODE DESIGNS FOR DENSE MEMORIES

3.1 Coding theory for Storage

Coding theory is the study of codes and their applications to specific fields. Coding has been used in a variety of computer science applications, from error correcting in the transmitting of data to increased data storage compression. We aim to extend the benefits of coding theory to dynamic random-access memory systems. We propose a memory scheme in which a small portion of memory is reserved for the efficient coding of pre-existing data. In essence, this allows the data of one bank to be duplicated and stored in an additional memory location. Traditionally, when multiple requests to a single bank are issued by the processor, a stall is generated. These types of stalls, known as bank conflicts, result from the fact that only one address from a single bank can be accessed at a time. The processor must wait for the result from the first bank to return before it can serve additional requests to the same bank. This lag can be a major bottleneck in a computer's processing speed. With a coded memory scheme, data present in multiple data banks will be compressed and stored in an extra banks, known as a parity banks. These parity banks will then be accessed concurrently with corresponding data banks to help alleviate stalls from bank conflicts. Ultimately, with the addition of a single parity bank we are able to generate a single additional access to any arbitrary bank without implementing any further logic to the bank itself.

In the following sections, we first describe the design parameters used to design the coding system. We then describe the two coding designs we have implemented, and give the initial results we have achieved through the implementation of these designs. We also describe certain techniques to reduce the cost of coding implementation.

3.2 Design Parameters

In this section, we discuss various design parameters that we use in this phase of the project to design and simulate the efficient code storage.

Memory overhead: The gains of multiple accesses to a memory bank every cycle comes with an associated cost. The cost is to store the compressed redundancy or the codes. The extra memory space used to store these codes is limited to 15 % of the overall memory.

Memory size: is an important parameter for consideration in design. The memory size and the parity storage size decide the code function to be used to essentially compress the redundant data. This design is considered for memory size of 8kB to 256 kB.

Memory Banks: The memory banks essentially are the units which store the data. We consider the code design for 8 memory banks. We consider the memory banks addressed with Least Significant Bits (LSBs) of the address. The last 3 bits of the address decide which bank, the memory address belongs to and the rest of the MSBs decide the location within the bank.

Cache line size: The memory accesses happen in burst as a cache line is evicted from the cache and is requested to be replaced by the cores. The cache controller thus requests a cache line which is a starting address and the length of the cache line. In this design, we consider cache line size of 128 bytes and 256 bytes. However, each core can potentially have a different cache line size and the concept of coding could be extended for various sizes.

Element size: Each memory location in a memory bank stores 256 bit of data. This essentially relates to decoding/understanding the address access request to the memory

bank. The cores request memories to be read or written for multiple elements. For example, a core with 128 bytes of cache line would request 4 elements of read/write for each cache line. The shared traces have two different request patterns, for 128 bytes and for 256 bytes.

Number of Cores: This parameter refers number of cores making access to the memory controller. Although we do not consider this parameter for designing the code scheme. However, we validate the design using the 6 core access trace shared with us for LTE and UMTS.

Access rate: This is the average rate at which the memory controller executes the reads/writes to the memory banks. In this design, we consider 10ns as the access rate. This means that 100 MHz memory clock rate. Though, this parameter does not affect the code storage scheme in any manner. It is required to simulate the performance for the shared traces.

3.3 Code Design

We discuss the design of the codes for creating extra accesses to memory in this section. First we discuss the code designs explored during Phase I. Second, we discuss specific execution strategies to efficiently implement the designs.

3.3.1 Code Design I

In this part, we attempt to design efficient codes based on the memory traces shared by Huawei. The goal of this design was to simulate the efficiency of coding and compare the results to the baseline implementation of not coding. During this design phase, we explored various code functions that could be used to create the codes on the stored data. We decide upon using the XOR function to store the data in the parity banks because of it's low complexity overhead and for preserving the linearity of codes. Linear codes offer the widest range of functionality because any order of the codes may be used to either encode or decode. This lack of dependency allows our design to use the parity banks in the most flexible way possible. We also explore the potential benefits of using different weights to the memory elements for the XOR function. For examples, the memory elements a0 and b0 could be stored as $\alpha a0 + \beta b0$ for any integer value of α and β. The least complex design for the decoder would be for taking $\alpha = 1$ and $\beta = 1$. Another design consideration explored is the compression factor to generate the codes. The codes can be generated by using xor on 2 or more memory elements. For example, suppose there are four banks A, B , C and D. Each of the banks hold a0 to an, b0 to bn , c0 to cn and d0 to dn elements respectively. The possible codes for these memories could be:

$$ai + bi, bi + ci, ci + di and ci + di \; for i = 0 to n$$

This scheme uses the combination of 2 memory elements to generate the codes. Although this requires 100% extra memory overhead, it enables 100% extra memory accesses per cycle, i.e., 4 extra accesses in this case.

Another design could be to compress the codes by combining all 4 memory elements to generate the codes:

$$ai + bi + ci + di \; for i = 0 to n$$

This design gives one extra access per cycle at the cost of 25% memory overhead. However, the decoder here needs to know 3 elements to be able to decode the 4th element. So although we are able to compress more data into a single memory location, it comes with the cost of additional memory logic. The scheme described above codes the memory storage banks using the elements from different banks. We call this type of coding as Interbank Coding. We also explore the orthogonal way of coding, i.e. interbank coding where we use the memory elements from the same bank to generate codes. Due to the simplistic nature of a pairwise coding scheme, we select this design to perform our initial experimentation. Specifically, we explore the benefits that coding provides to the latency of read requests as well as the queue build up for the memory controller. For the baseline test, we use a total of two banks and divide the memory region based on the most significant bit. We also assume that each memory accesses that is issued from the trace only reads from a single bank. In the coding design, we combine the memory present in the same row for the two banks and code this in a third additional parity bank. We simulated this scheme with the traces provided. Figure **??** captures the effect that coding has on both of these parameters for varying numbers of cores. Figure **??**a and **??**b show the read access latency in the case when we do use coding compared to without coding. The results suggest that coding significantly reduces the read access latency for any number of cores. It can also be observed that the benefit of coding is more significant for more number of cores since the number of requests to the coded area is high. Figure **??**c and **??**d show the queue build up for writes and reads. These figures suggest that the queue build up significantly reduces when we use coding compared to not coding.

3.3.2 Code Design II

In this section, we explore the code design for the following objectives:

1. Read access : 4 per bank in one cycle
2. Write access : 2 per bank in one cycle
3. Memory size 8 kB to 256 kB
4. Number of Banks : 8
5. Memory overhead : 15%
6. Parity banks : 5 or 6 shallow banks for code storage

Using the above parameters, we design coding scheme to efficiently store the codes and achieve the objective. We use the concept of batch codes to code specific areas within each of the banks. This allows us to serve multiple accesses to the coded region using the parity banks. With this scheme, we guarantee that any 4 requests to the coded region can be served at any given time. As shown in figure 3, 8 banks are divided into two regions. Each region consists of 4 banks. Each region has 6 parallel shallow banks to store the parity. The colored regions shown in the banks 1to 8 are the coded region. These region are assumed to be of α fraction of the memory.

Figure **??** presents storage scheme for this design.

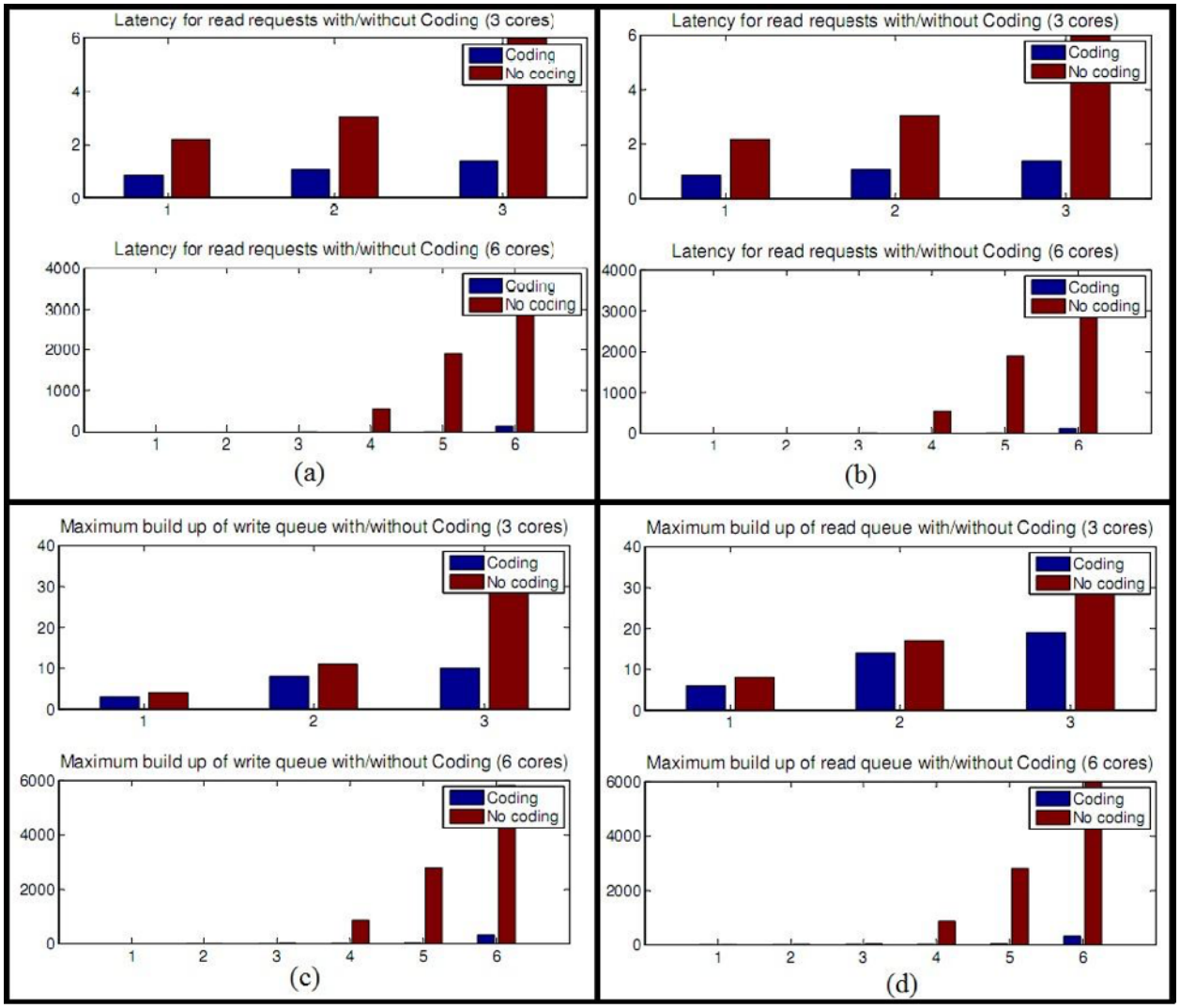

Figure 2: The effect of coding on both of these parameters for varying numbers of cores

Best case analysis: We design this code to achieve maximum performance when sequential accesses to the coded regions are issued. In Figure **??** we show the best case access, we can achieve up to 10 parallel accesses in one cycle. Consider the scenario if we receive accesses to a, b, c, d1, a2, b2, c2, d2, a3, b3, c3, d3. Here, we can serve a, b1, c1, d1 using a1 with the parity banks a1+b, b1+c1, c1+d1 and serve a2, b2, c2, d2 using b2 with the parity banks b2+c2, b2+d2, a2+c2. Lastly, we can serve c3 and d3 using bank 3 and bank 4.

Worst Case analysis : The code scheme falls off to 4 access in a cycle when there are non-sequential and non-consecutive access to the memory banks. For example when the access pattern is to a1, b8,c9,d15. Since we do not code this combination, it does not get benefit of extra access. However in this case, we can use the prefetching mechanism to look ahead in the queue and prefetch codes from parity banks for the subsequent access.

In Figure **??** and Figure **??**, we explore the worst case scenario when the accesses are random. The results show that the queue build up for reads and writes does fall back to no-coding scenario. This asserts that the worst case scenario for a coding scheme performs similar to no-coding scheme.

In the second scheme, we augment the code storage by cross storing the codes from region 1 to region 2 and vice versa. We do this in addition to coding the consecutive memory addresses in a bank. This provides two benefits, first it increases the overall redundancy, and second it allows us to use the parity banks of the other region in case the first regionâĂŹs parity banks are in use. Figure **??** shows the storage pattern of the codes in the bank. The overall overhead in this system is 2.5 α.

Best case analysis: We design this code to achieve maximum performance when sequential accesses to the coded regions are issued. During the best case access, we can achieve up to 9 parallel accesses in one cycle. Consider the scenario if we receive accesses to a1, b1, c1, d1, a2, b2, c2, d2, a3, b3, c3, d3. Here, we can serve a1, b1, c1, d1 using a1 with the parity banks a1+b1, b1+c1, c1+d1 and serve a2, b2, d2 using b2 with the parity banks a2+d2 and b2+d2. Lastly, we

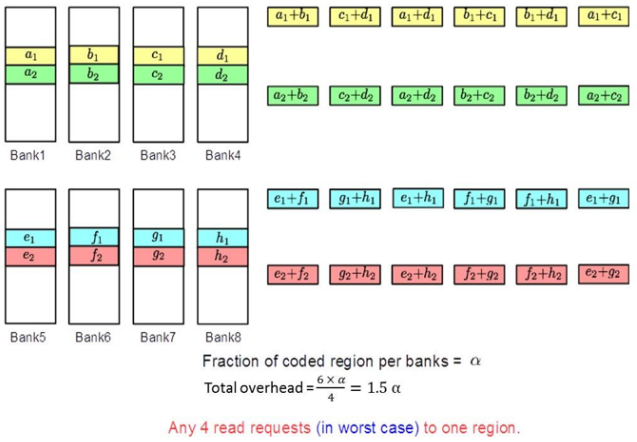

Figure 3: First Storage Scheme

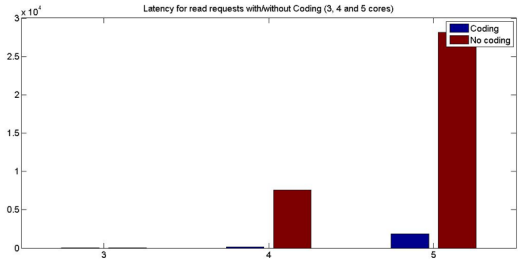

Figure 4: Latency for read requests with/without Coding

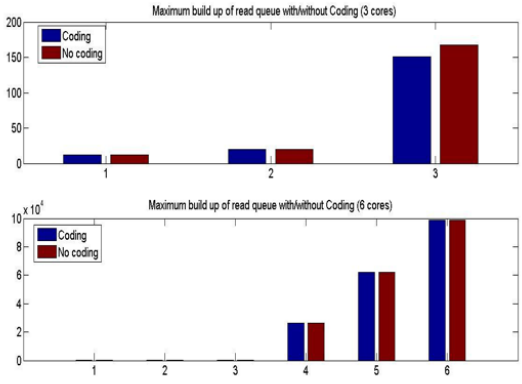

Figure 5: Maximum build up of read queue with/without Coding

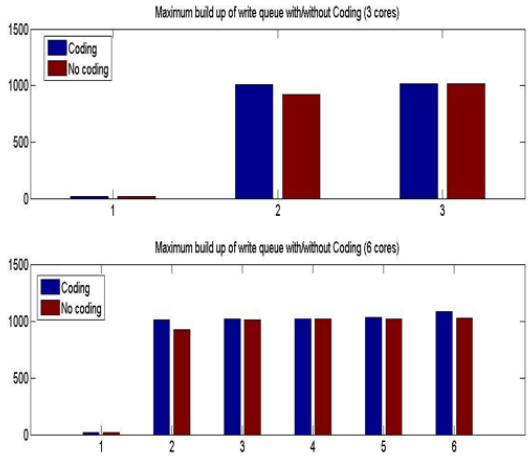

Figure 6: Maximum build up of write queue with/without Coding

can serve c2 and d3 using bank 3 and bank 4.

Worst Case analysis : The code scheme can do 5 access in a cycle for the coded region in worst case. These are non-sequential and non-consecutive accesses to the memory banks. For example when the access pattern is to a1,a6,a9,a15,a20. We can perform these 5 reads with the help of coded banks. We can use the prefetching mechanism discussed later to look ahead in the queue and prefetch codes from parity banks for the subsequent access.

3.3.3 Coding Performance Enhancement

In this section, we explore the technique of dynamic coding in order to reduce the memory and access overhead associated with the parity banks. We first discuss the scheme of dynamic coding and follow it by discussing the potential benefits of prefetching the codes.

Dynamic Coding: The contention in memory accesses from various cores occurs mostly when the access are to shared memory, especially when they are localized to certain memory regions. We explore the locality of the memory access over a period of time to reduce the memory overhead for storing the codes. In a multi-core system, when various cores try to work from a shared memory location, they tend to generate accesses to a localized region of memory. This motivates the idea of coding the localized region during the period of heavy access, and dynamically changing the region whenever there is change in the locality of memory accesses.

Figure **??** shows the access pattern of the LTE cores 0 to 6. The y-axis of the figure shows the address accessed by the LTE cores over a period of time. The x-axis denotes the time in nanoseconds. This plot shows that most of the access from various cores are limited to the memory range from 0x007a1200 to 0x00c65d40 (lower and higher range on the y âĂŞaxis). It also suggests that most (about 60%) of the accesses belong to the memory region of 0x00a037a0 to 0x00b71b00.

We make similar observation from Figure **??** for UMTS. We observe a highly concentrated access pattern in case of UMTS. Here again, all of the access for a duration of approximately 0.2 ms is in the address range of 0x007a1200 to 0x01036640.

From the above observations, we demonstrate the idea of coding the highly accessed portion of the memory. This scheme benefits from a huge reduction of the memory overhead with coding. The reduction the memory overhead can

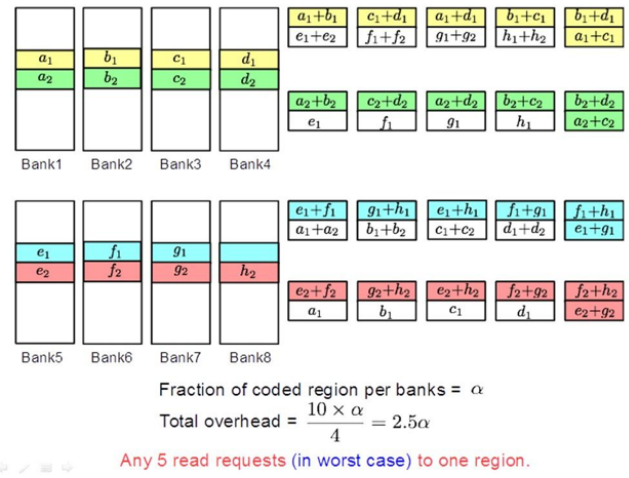

Figure 7: Second Storage Scheme

be used to reduce the complexity of the decoder by using simple coding functions (e.g. xor) and for densely coding (e.g. repeatedly coding a single element using 2 elements).

The scheme of dynamic coding requires that the currently coded region changes when the access pattern changes. That is, the localized memory area that is most heavily accessed can change, and it will require the system to recode the new localized access region. We assume that the working area of a program changes with change in the input parameters to the program. It can be easily observed from the above figures that the working area or the localized area is constant for at least 0.2 ms. This suggests that the switching of the coded region is not very frequent. During these periods of coding switches, it is also guaranteed that the number of accesses served from the memory is at worst equal to the number of banks available. In other words, coding the memory has no performance degradation compared to non-coding during these times. The system also needs to maintain an algorithm to observe the access pattern of the cores, and make a decision when it is time to code a new memory region. To do this, the memory controller tracks the most accessible region during a time period and makes a decision to slide/shift the coded region. This shift in the coded region requires the update of the parity bank for the new region. This process is carried out in conjunction with the ongoing access to the newly coded region. Therefore, this operation only requires writes to the parity banks, since we can use the current reads from the coded region to access the data that is to be coded. In addition, reads are also scheduled in the idle periods, when there is no read or write request to the bank/banks.

Dynamic coding requires the system to divide the memory into subregions and to keep track of accesses in these sub-regions. Once the number of accesses to a sub-region reaches a given threshold, it must then make this region the currently coded area. We propose this mechanism based on window concept. The system maintains a tuple of sub-regions such as [Starting Address, Length]. Each sub-region is thus given a starting address and length. Any access to a particular sub-region is considered as a hit. The system has a hit counter associated with each of the sub-region which is incremented for each hit. The system makes a decision of coding a particular sub-region based on its counter value. The number of coded sub-regions at a particular time is based on the sub-region size and the code storage size. The eviction of a coded region follows the LRU policy similar to cache.

Prefetching Codes: The technique of dynamic coding reduces the memory overhead by exploiting the localized nature of memory accesses from the cores. In this section, we explore prefetching the coded data to reduce the access overhead caused for fetching the codes. This is done by exploiting the gaps in the memory access to any bank and using these gaps to prefetch the code/data for a future memory access. During a program, there are access cycles when certain banks do not have any access scheduled for a read/write. We propose the prefetching technique where we look forward in the queue and anticipate a pre-fetch for the data/code for that bank. We explore the implementation of a memory prefetching unit, similar to an instruction or cache prefetching unit. This unit can detect linear access patterns to regions in memory. For example, if a string of memory accesses are issued in sequential byte sized order, then the prefetching unit will predict the next access to be in byte increments. The memory prefetching works by fetching a predicted address from the parity bank during accesses that the parity bank is idle. When future memory accesses are issued, they are first checked with the pre-fetched data to see if they can be used to decode any subsequent accesses memory accesses. If so, the memory access is obtained from the current accesses and pre-fetched data. For example, say the pre-fetcher sees 2 consecutive memory requests in a row. It then predicts that the next two accesses, locations $a0$ and $b0$, are likely to be accessed in the near future. It reads $a0 + b0$ from the parity bank for future use. Next, access to location $a0$ and $b0$ are issued to the memory. Now, instead of reading both $a0$ and $b0$, only a single location has to be read from in memory, while the other location can be obtained from the pre-fetched data. This allows for an additional access to be issued from the now free memory bank. In these cases, it is possible to obtain up to two additional memory accesses in a given cycle, one from the pre-fetched data and one from the parity bank.

Implementation of a memory prefetch should only require overhead for space and the associated logic to implement it. Since memory accesses are often stalled due to bank conflicts, checking pending accesses to the pre-fetched data should require no additional time overhead. As memory accesses wait to be issued in the bank queues, they can simultaneously be checked with the pre-fetched data. Thus, no extra latency is anticipated by the addition of a memory prefetching unit.

Figure **??** shows two plots of memory accesses to a bank with respect to time. The left figure shows the accesses to the memory bank by various cores. The right side figure shows a zoomed view of the accesses in the dense access region. This figure suggests the linearity of accesses. The system can look ahead in the queue to detect the consecutive address request for a memory bank and schedule a prefetch of the associated code.

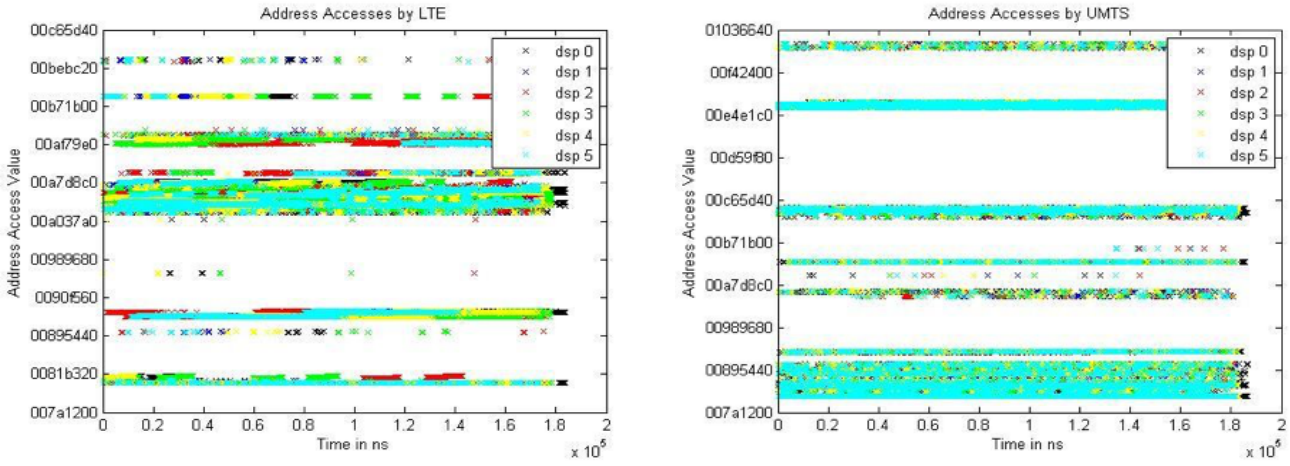

Figure 8: Access Pattern of the LTE cores 0 to 6

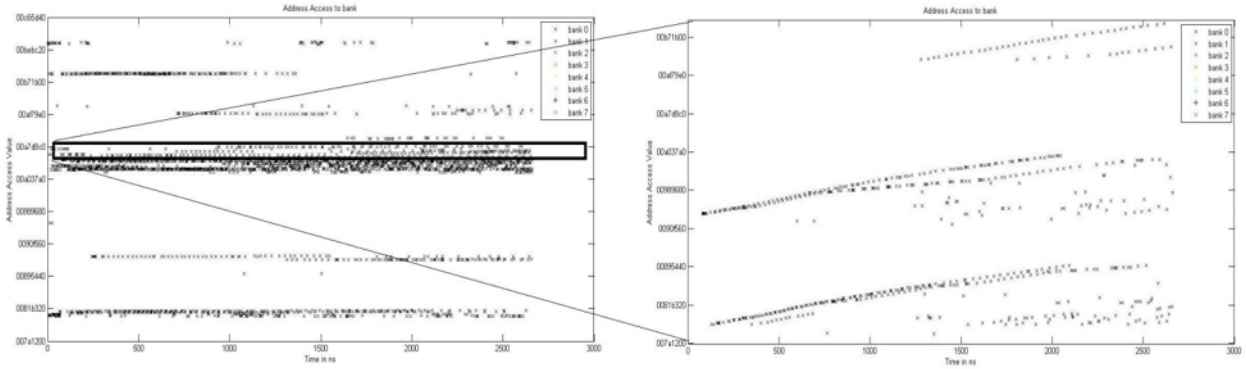

Figure 9: memory accesses to a bank with respect to time

In Figure **??**, we simulate the prefetching of the code by using a window of length 100. That is, we look ahead to 100 request in the queue and find out the occurrence of consecutive address in the window. The plot suggest high occurrence of the consecutive addresses in the bank which can be served by prefetching the codes.

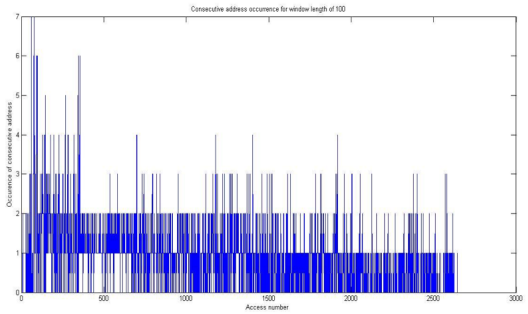

Figure 10: simulation of prefetching of the code by using a window of length 100

4. CODING ARCHITECTURE FOR HBM

4.1 Reed Solomon (RS) Coding Scheme

We propose a coding scheme for HBM DRAM on 16 data banks. The main component is a (6, 4) Reed Solomon code over a Galois Field size of 2^8, denoted GF(2^8). This code's systematic implementation can be used to convert 4 messages into a codeword of length 6 with the ability to recover the original 4 messages using any 4 symbols of the code word. This (6,4) code is a maximum-distance separable (MDS) code that reduces the overall memory transaction to reconstruct the code.

In coding theory literature, MDS codes are defined as codes that meet the Singleton bound: d=n-k+1. Here, d is the distance, i.e. the minimum number of positions in which any 2 code words differ, n is the block length, and k is the dimension. For our RS code d=2, n=6, and k=4. One can correct up to d/2 errors in an MDS code, which corresponds to at most 1 error in our implementation.

This code was chosen for several reasons. First, codes with the MDS property are preferred to ones with distance strictly less than the Singleton bound. When available, Reed

Solomon codes provide an explicit MDS construction for the pair of parameters (n,k). Often this comes at the expense of a large field size, but later in this section we describe a method to map the arithmetic to binary operations on the individual bits of each byte vector [**?**]. Reduced logical complexity means that the vector RS code is amenable to pipelining and may be used in both HBM and HMC systems. Throughout the design process, adjusting the RS parameters will allow us to achieve other points on the overhead-performance tradeoff. In contrast to other coding schemes for memory systems like SEC-DED [**?**], BAMBOO [**?**], and ChipKill [**?**], the proposed coding scheme allows for more intelligent memory controller design by improving data access and error correction capability.

Although some randomized codes are known to satisfy the MDS bound, our hardware implementation benefits from having deterministic encoding and decoding functions. Then the operations can be âĂIJhardcodedâĂİ into the memory controller to improve runtime performance. Efficient encoding and decoding implementations have been studied for decades. Moreover, RS codes have proven to be useful in previous work on distributed storage, both in theory and in practice [**?**][**?**] [**?**]. In fact, the (14,10)-RS code is in production at Facebook in their HDFS-RAID system [**?**]. This makes it a natural starting point for our dense memory storage architecture.

4.2 Combine Coding Algorithm with HBM

The coding scheme is based on a single channel with HBM DRAM operating in the Pseudo Channel Mode. Since this mode divides a channel into two individual 8 bank sub-channels, we assign one sub-channel for data banks and the other sub-channel for parity banks. The two sub-channels share the channel's row and column command bus, but decode and execute commands individually. For commands that are common to both pseudo channels, it is required that the respective timing conditions are met by both pseudo channels when issuing that command. In Pseudo Channel Mode, the burst length is set to 4.

4.2.1 Adress Mapping

The Figure **??** below describes the address mapping for the channels. The blue colored least significant bits of the address signifies the byte level addressing of the data. The higher six LSBs in green indicate column address, the next 14 bits in red indicate row address, and the remaining 3 bits decide the bank.

Figure 11: Address mapping for channels

During encoding, the data is arranged according to the RS(6,4) code. The first two parity banks store the parity data that is generated from the first four data banks. In other words, the parity data generated by the data from data bank 0 to 3 is stored in the parity bank 0 and parity bank 1. Similarly, the data generated from the remaining four data bank is stored in the parity bank 2 and parity bank 3. The parity bank 4 and parity bank 5 contains the parity generated from the code using the even numbered data banks(0,2,4,6). And using the same principle, parity bank 6 and 7 store parities generated from the code for data in odd numbered data banks(1,3,5,7). The upper part of the bank stores the Pseudo Channel 0 and the lower part of the bank stores the Pseudo Channel 1 (Figure **??**).

Figure 12: Data storage layout for coded memory system

4.2.2 Scheduler Algorithm Design

A reorder buffer is added to the scheduler. For each read and write request, the scheduler first looks into the reorder buffer and then schedules according to the state of the reorder buffer. The following Figure **??** shows the structure of the reorder buffer. "Done" is a one bit value, indicates whether the data is ready to be written into the memory. "R/W" is an eight bit value, indicates which bank has been written to. "DValid" is an eight bit value, each bit indicates whether the corresponding data bank is valid. "PValid" is an eight bit value, each bit indicates whether the corresponding parity bank is valid. "Row" is an address indicates which row is the entry in the reorder buffer stores. "DB0-DB7" stores the data of the data bank. "PB0-PB7" stores the data of the parity bank. A head and a tail pointer are used to maintain the validity of the entry in the reorder buffer.

Done	R/W	Dvalid	Pvalid	Row	DB0	...	DB7	PB0	...	PB7

Figure 13: Structure for the reorder buffer

4.2.3 Read Algorithm

The main thrust of Read Algorithm is to maximize the read accesses per cycle in all scenarios. This algorithm runs in the access scheduler unit. The focus here is to use the redundancy in the parity bank in order to serve more accesses. The accesses from parity banks is useful only if one of the corresponding access is made to the data bank.

The following flowchart in Figure **??** shows the procedure of the read algorithm. The decoding algorithm is mainly

used when only the parity bit is valid. There are two cases that could be encountered during decoding: 1) two data banks and two parity banks are valid; 2) three data banks and one parity bank are valid. For the Scenario 1 the decoding algorithm is described in the previous section, and for Scenario 2 the decoding algorithm is similar but and can be decoded more quickly.

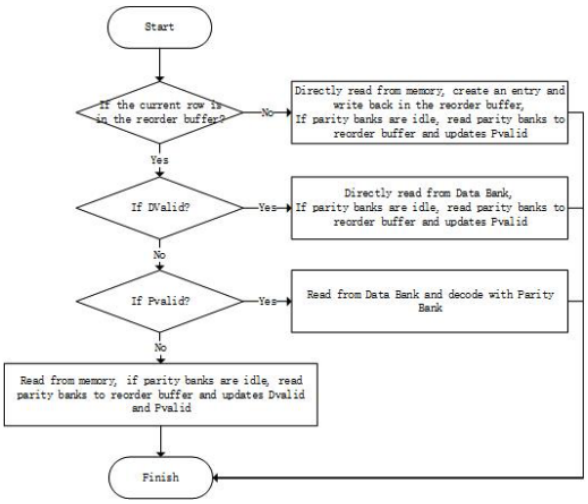

Figure 14: Read Algorithm Flowchart

4.2.4 *Write Algorithm*

Figure **??** shows the write algorithm flow chart. For each write request to a new row, the scheduler creates an entry in the store buffer and moves the tail pointer to one block down. For each write request to an existing row, check Dvalid and Pvalid, If part of DB are valid, update corresponding PB and Pvalid. If all DB are valid, update PB. If all DB and PB are valid, set Done, ready to commit. Special case: If only one DB is invalid and the bank is idle, read from memory.

4.2.5 *Writeback Algorithm*

Since the size of the reorder buffer is limited, every time the buffer becomes full, data in the buffer should be written back to the banks. The reorder buffer acts like a double-pointer fifo, we have a head pointer and a tail pointer indicate the beginning and the end of the current buffer. When the fifo becomes full and a new entry is to be allocated, we evict the old entry pointed by the tail pointer and move the tail pointer one location backforward and the head pointer one location forward. The write-back steps are more complicated when the evicted entry has been updated by write instructions (we have a "R/W" value in each entry to indicate if the entry has be written).

5. EXPERIMENT METHODOLOGY

In this section, we present our experiment setup. We run the simulation with SPEC2006 benchmark and the LTE, UTMS application traces. Generally, SPEC2006 is a single-core benchmark, thus we look at the results with the simulator running with a single-core processor. For the application

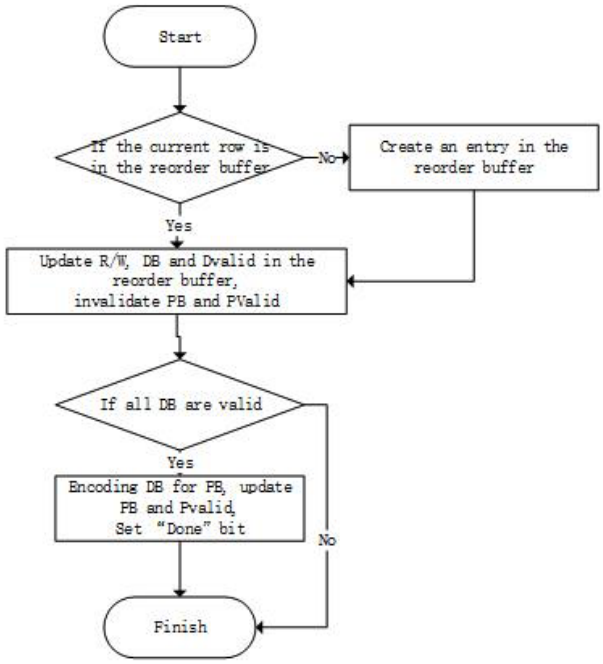

Figure 15: Write Algorithm Flowchart

traces, we want to look at the performance of our architecture with a multi-core processor. Thus, we run the application traces with a 6-core processor.

We run the simulation with the baseline model and the modified model where I have added the reorder buffer as well as the coding structure. In this way, we can compare the cycle number and find out how much performance improvement we can get with our innovative model.

We present the architecture diagram of our model, one the the single-core processor model and the other one is the multi-core processor model.

5.1 Architecture Diagram with single-core processor

The SPEC2006 benchmark is running with a single-core processor as shown in the figure **??**. The instruction trace are read by the processor and generate corresponding memory requests. The memory requests are pushed into request queue which is connected with the input of the reordered buffer of our memory system. After the processing steps described in the algorithm above, the bank requests are generated and sent to the memory controller resides in the HBM memory. The data are sent back to the reorder buffer or the CPU via the data bus.

5.2 Architecture Diagram with multi-core processor

For the application traces, we run the simulator with a multi-core processor, specifically 6 cores, shown in figure **??**. The memory traces are read by the simulator and then pushed into the request queue. For each item we store the corresponding cycle number, read or write access and address. Since it's a cycle accurate simulator, at the beginning

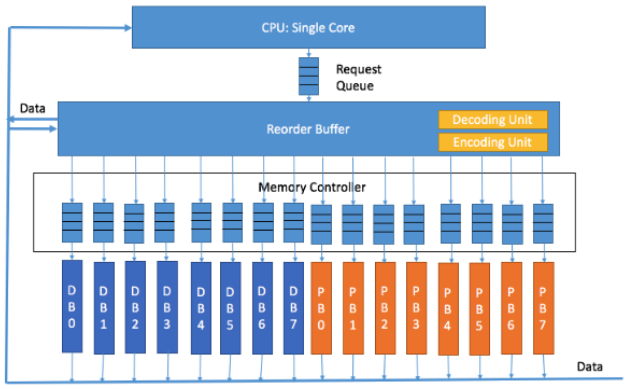

Figure 16: Single-core CPU Simulation

Depth	8	16	32	64	128
Size	8KB	16KB	32.2KB	64.5KB	129KB

Table 1: Reorder Buffer Size under Different Depth

of each cycle, the simulator will pop multiple requests from the request queue, all the requests are issued in the same cycle.

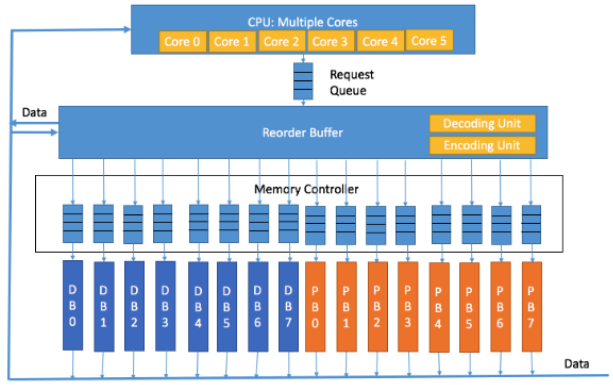

Figure 17: Multi-core CPU Simulation

6. RESULTS

Our code design implemented on Ramulator platform provides us a platform to evaluate the performance improvement in the memory access when compared to the original uncoded memory layout. The Ramulator platform allows for excitation of the memory scheduler module with standard benchmarks that represent differentiated and distributed computational and memory access pattern. Figure **??** - Figure **??** captures result of comparison of the performance improvements across different benchmarks.

6.1 Cost Analysis

Table 1 shows the reorder buffer size under different depth.

6.2 Simulation Cycles

In this section, we present the simulation cycles of the baseline model and the modified model. The modified model is running multiple times with different reorder buffer size, specifically, the size varies between infinite, 128, 64, 32, 16,8. We can see that the performance improvement is higher when the reorder buffer size is larger. This is the design trade-off between the memory overhead and the performance.

The mcf benchmark has the most significant improvements with the use of our coding scheme because of the reason that most of its instructions are memory requests and most of the memory requests are read requests. Since we donâĂŹt include the delays of the write back of parity banks and the encoding and the decoding steps, the improvements for each benchmark is the result of the reducing the latency of the read requests with the coded data from parity banks. On the other hand, the mcf trace hit its performance bottleneck when the reorder buffer size goes beyond 64. This means that memory pattern of mcf is clustered and all the memory requests could be stored in the reorder buffer when the reorder buffer is large enough. In order to further improve the performance, we need to come up with other alternatives in addition to increasing the reorder buffer size.

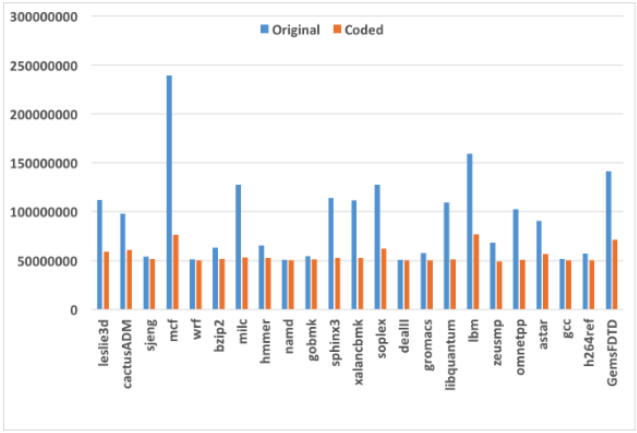

Figure 18: Simulated Number of CPU cycles: Baseline HBM versus Coded HBM across different benchmarks (with infinite reorder buffer size)

For the application traces, the performance improvement is suboptimum compared to the SPEC2006 benchmark and increasing the reorder buffer size will not improve the performance. The reason for this is because the memory pattern for the application traces are clustered which makes the reorder buffer hit its bottleneck when the size is only 10. Thus in order to further improve the performance we need to come up with other alternatives in the next phase.

7. CONCLUSIONS

In this paper, we explore the use of coding theory in DDR and high bandwidth memory (HBM) system. Conventional DDR systems use timing optimization techniques around DDR protocol to improve the efficiency of accesses. In this work we propose increasing redundancy to help distribute access

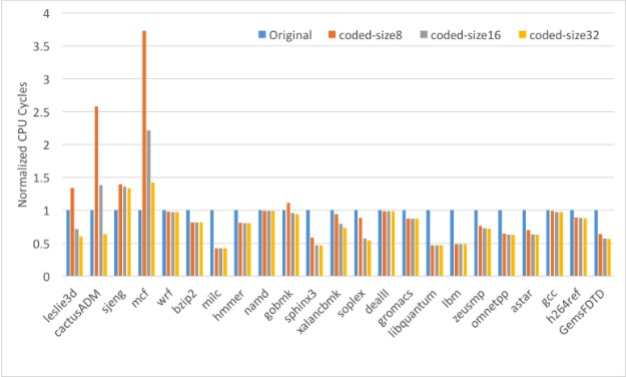

Figure 19: Simulated Number of CPU cycles: Baseline HBM versus Coded HBM across different benchmarks (with reorder buffer size of 8, 16, 32)

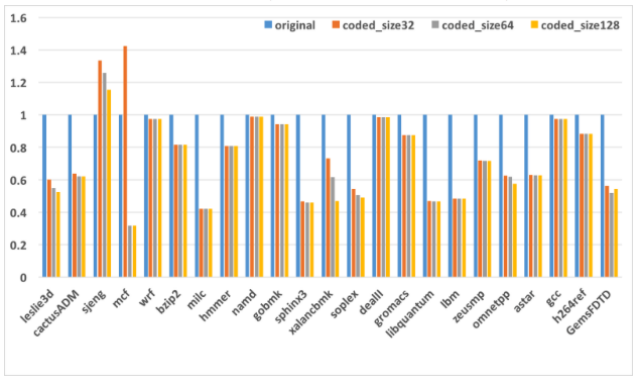

Figure 20: Simulated Number of CPU cycles: Baseline HBM versus Coded HBM across different benchmarks (with reorder buffer size of 32, 64, 128)

across dram pages. This will help in efficient retrieval of data. These schemes will be evaluated and optimized with several iterations. The aim is to find the best performing code design scheme given the performance requirement compared to alternatives. Our proposed scheme results in a decrease in the of the number of contended memory accesses between cores, therefore reducing the overall latency of the system. The reduction in the latency can be seen directly as an increase in the overall system performance. We show that with a memory overhead of 15%, we can enable 4 extra access to a bank while remaining within the given design parameters. We also included emphasis on the code design and optimization of the access scheduler and made significant process in improving the design emulation on the Ramulator platform.

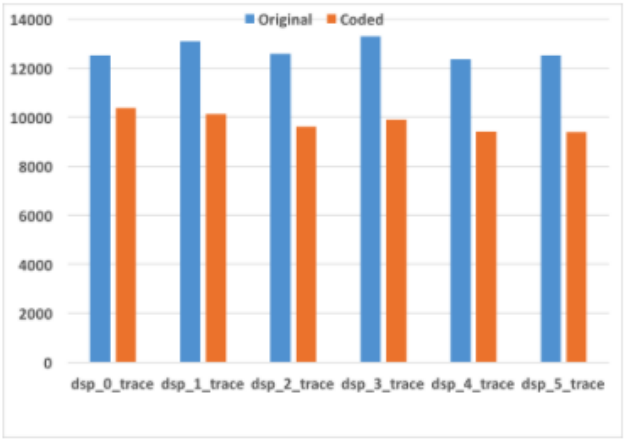

Figure 21: Simulated Number of active dram cycles: Baseline HBM versus Coded HBM across application-driven LTE traces (with reorder buffer size of 10)

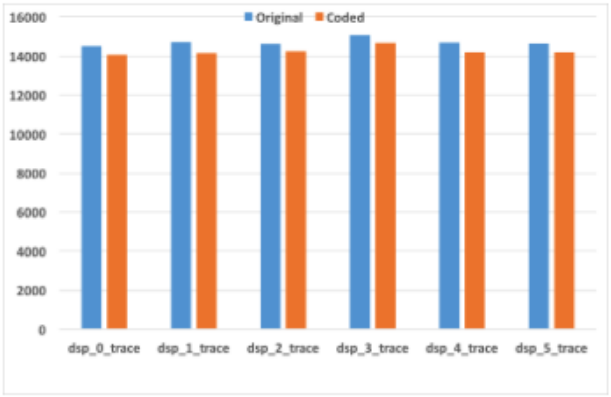

Figure 22: Simulated Number of active dram cycles: Baseline HBM versus Coded HBM across application-driven UTMS traces (with reorder buffer size of 10)