# 

[**1 Introduction**](#_it9v2j1xrrr) **1**

[**2 Summary of Phase I**](#_80idj1ugdo00) **2**

[**3 Technical Discussion**](#_m0hdtc4y1for) **2**

[3.1 Code Design for HBM](#_xulbd83cgtzi) 2

[3.1.1 Overview of HBM architecture](#_6acdtnn0gn0b) 2

[3.1.2 Coding algorithm](#_9qcxlp57e27l) 3

[3.1.3 Combine Coding Algorithm with HBM](#_uke4i75i8mba) 4

[**4 Progress in Phase II**](#_5oq5uhhx2zok) **4**

[4.1 Optimize Reed-Solomon coding algorithm](#_dk8yxjagl0y) 4

[4.2 Implement coding algorithm with HBM](#_tufl4ljpkbg9) 6

[4.3 Scheduler algorithm design](#_5asfuz1zuxvx) 7

[4.3.1 Read algorithm:](#_ee48ye4rcm5m) 8

[4.3.2 Write algorithm:](#_y7pbgmxyw3t3) 9

[4.3.3 Refresh algorithm:](#_uf5uhnvjqwj) 9

[4.4 Code design with Ramulator](#_16k29lywiwex) 9

[4.5 Improvement over default Ramulator controller](#_vtcqn9j9yamo) 10

[**5 Key Results**](#_lh5zbbcaphij) **10**

[**6 Summary and plan for next phase**](#_59ujy3ceiaoq) **11**

[**7 References**](#_fkee596p2ure) **11**

# 1 Introduction

In the era of disruptive development in the fields of data science and machine learning, computing systems with high processing capacity are in huge demand. A system’s processing power is determined primarily 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 that uses redundancy in data storage to provide parallel memory access with low latency.

# 2 Summary of Phase I

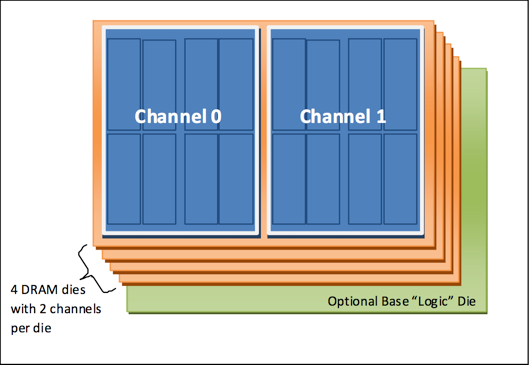
In Phase I of this project, we defined the problems and governing parameters for coding architecture for dense memory systems. We also devised the evaluation criteria for coding schemes for memory architecture and carried out detailed study of the next generation HBM and HBC memory layout, access protocol and controller design. Finally, we developed a systemC model to support evaluation of proposed memory coding scheme.

# 3 Technical Discussion

In this phase, we have made several improvements on the proposed coding schemes. These include optimizing the decoder, improving the write algorithm, and characterizing performance via Ramulator, an event-driven DRAM simulator.

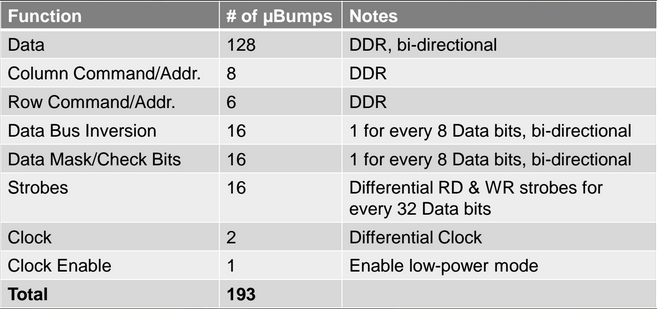
## 3.1 Code Design for HBM

### 3.1.1 Overview of HBM architecture

HBM stands for High Bandwidth Memory which enables systems with extremely high bandwidth requirements like future high-performance GPUs. The HBM standard defines a stack of memories as shown in the Figure 1. Each stack contains 2 channels with 4 DRAM dies. This memory design enables a 3D scaling of memory, which packs more data in the same space as 2D scaling and reduces access latency. The architecture also helps the computing cores to parallelize their access channels, meeting the high data access requirements while consuming less power.

The HBM standard defines a stack with features such as footprint, signaling interface, commands, protocols and ECC support. However, it does not specify the stack’s internal architecture or the precise DRAM timing parameters. This flexible architecture enables system designers to design various memories and storage patterns that enable higher access rates.

Each HBM channel provides access to an independent set of DRAM banks. Requests from one channel may not access data attached to a different channel. Channels are independently clocked, and need not be synchronous. Each channel is expected to provide a 128-bit data interface that supports up to 1-2 Gbps of data rate per signal equivalent to 500-1000MHz of DDR memory. This is equivalent to 16-32 GB/sec of bandwidth per channel. 8 channels stacked together as in Figure 1, which provides 128-256 GB/sec of bandwidth per stack. The following table summarizes key parameters of HBM memory:



**Table 1** Tabular description of parameters of HBM memory

Since each channel is independent, our coding scheme of this project will be based on a single channel with HBM DRAM operating in Pseudo Channel mode. Pseudo Channel mode divides a channel into two individual sub-channels of 64 bit I/O each, providing 128 bit pre-fetch memory Read and Write access for each Pseudo Channel. Both Pseudo channels operate semi independently: they share the channel’s row and column command bus as well as CK and CKE inputs, but decode and execute commands individually. Address BA4 is used to direct commands to either to Pseudo Channel 0 (BA=0) or Pseudo Channel 1 (BA=1).

### 3.1.2 Coding algorithm

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 [6]. 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 [3], BAMBOO [2], and ChipKill [4], 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 “hardcoded” 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 [1,5-6]. In fact, the (14,10)-RS code is in production at Facebook in their HDFS-RAID system [5]. This makes it a natural starting point for our dense memory storage architecture.

### *3.1.3 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.

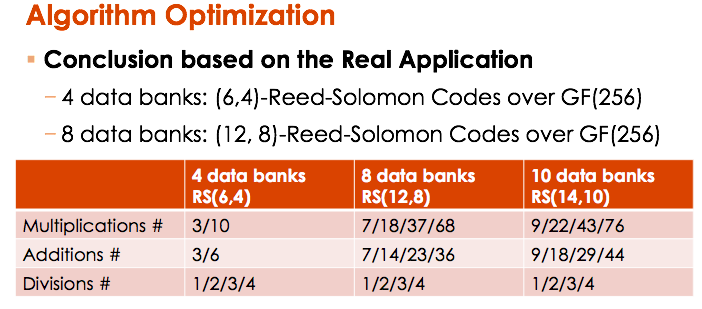
# 4 Progress in Phase II

During the Phase II of this project, we have looked at five main aspects described below:

## 4.1 Optimize Reed-Solomon coding algorithm

In the last phase we proposed a RS(14, 10) coding scheme which provides parallel accessibility to the data banks while minimizing the parity data storage overhead. This scheme, however, requires at least 10 data banks to be open at the same time, which leads to higher power consumption. Our analysis suggested that using RS(6, 4), we can get similar parallel access benefits while minimizing the power and overhead costs. Moreover, the RS(14,10) code demands larger computational cost for decoding, and requires a large lookup table. Both of these can increase the access latency and memory overhead. In this phase, we have examined augmenting the basic Reed Solomon decoding algorithm by adding an improved Gaussian Elimination algorithm.

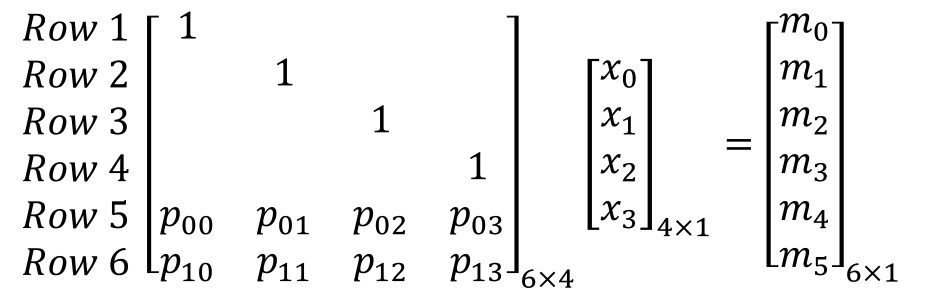
The figure below compares RS(14,10) with RS(6,4):



**Figure 2** *Comparison of RS(14,10) with RS(6,4) based on the decoding algorithm complexity requirements*

In standard Gaussian elimination decoding, a matrix inversion is required for finding the location and correction value. The matrix inversion implementation is computationally expensive and requires more memory. In the improved Gaussian elimination, we use iterative subtraction and elimination to compute the missing data element from the code.

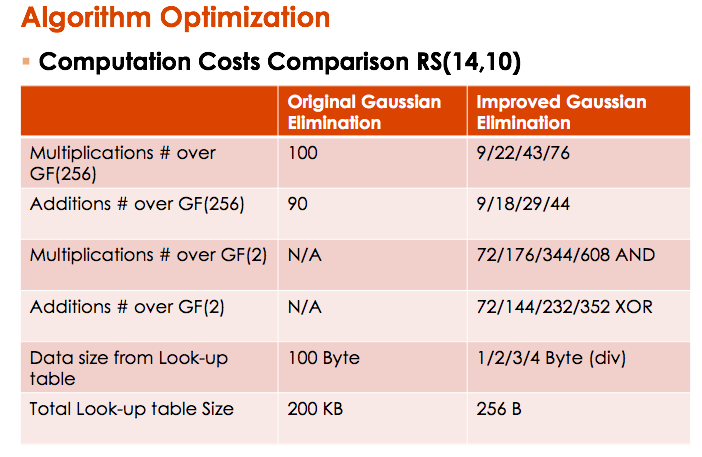
According to the generation function shown below, m0, m1, m2, and m3 are from data banks, while m4 and m5 are from parity banks. In this example, assume we receive m0, m1, m4, m5, and must solve for m2 and m3.



We have the following decoding steps given two data from data banks and two data from parity banks:

1. Multiply Row 1 by p00 and subtract by row 5 and row 6, get A1
2. Multiply Row 2 by p01 and subtract by row 5 and row 6, get A2
3. Multiply A1 by p02, multiply A2 by p12, then subtract, get A3
4. Divide A3 by (p03\*p12 - p13\*p02), get m3
5. Subtract A2 by m3\*p12, then divide by p13 get m4

The figure below summarizes the results of the optimization over decoding algorithm (RS(14,10)):



**Figure 3** *Computational cost improvement with optimized gaussian elimination algorithm. The improvement comes with reduction in the multiply and accumulate functions along with significant reduction in lookup.*

We have improved this algorithm and customized it for our implementation to reduce the computational cost.

## 4.2 Implement coding algorithm with HBM

In the previous phase report, we discussed the coding architecture and code distribution for HBM memories. During this phase, we have used this coding along with the Pseudo Channel Mode of HBM to store the data and parity more efficiently. This mode divides the channels into two logical parts: data and parity. We use addressing structure of the Pseudo Channel Mode to access the data and parity banks.

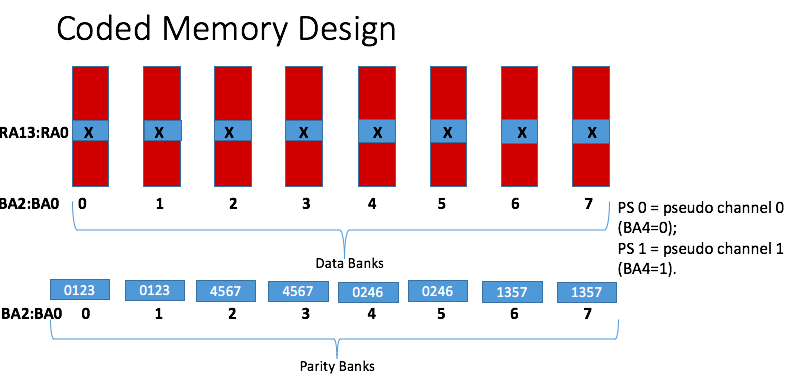
The Figure 4 below describes the address mapping for the channels. The blue colored least significant bits of the address signifies the byte level addressing of the data. The higher six LSBs in green indicate column address, the next 14 bits in red indicate row address, and the remaining 3 bits decide the bank.



**Figure 4 :** *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.

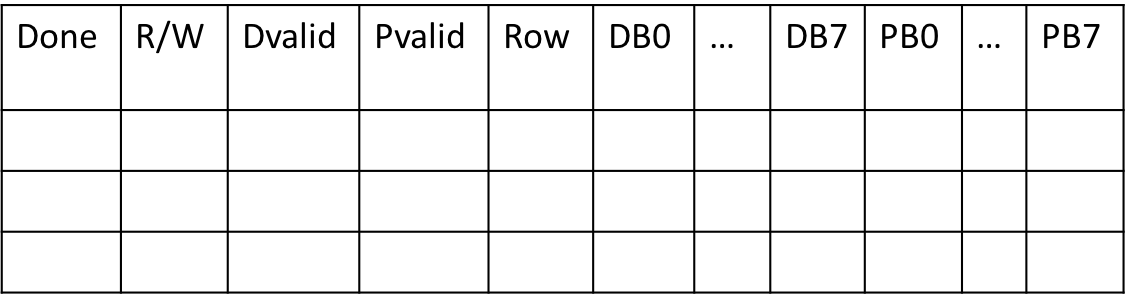


**Figure 5** *Data storage layout for coded memory system*

## 4.3 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 6 shows the structure of the reorder buffer,



1

**Figure 6** *Structure for 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.

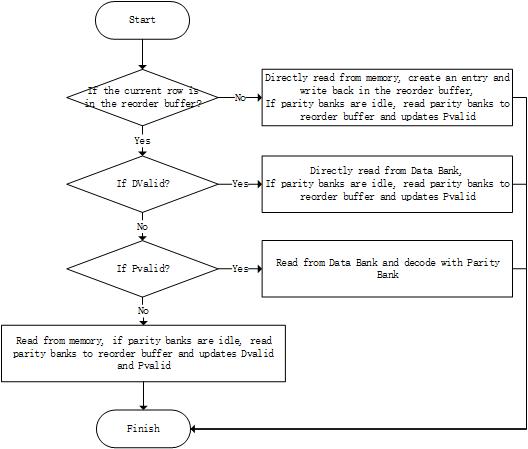
“PB0-PB7” stores the data of the parity bank.

A head and a tail pointer are used to maintain the validity of the entry in the reorder buffer.

### 4.3.1 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 7 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 7** *Read algorithm flowchart*

### 4.3.2 Write algorithm:

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.

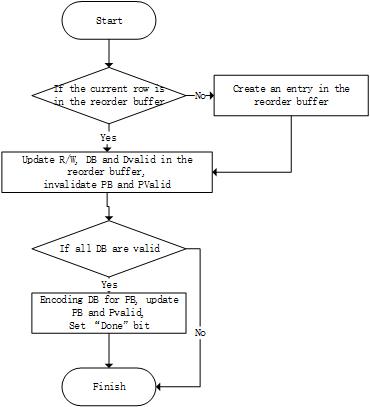
For each write request to an existing row, check Dvalid and Pvalid,

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



**Figure 8** *Write algorithm flowchart*

### 4.3.3 Mixed read and write requests

### 4.3.3.1 Write-after-Read

Here we consider the scenario when a write request is issued after a read request to the same location. During the read request, the scheduler first allocates/finds an entry in the reorder buffer according to the read algorithm, which ensures that the corresponding row resides in the reorder buffer after the read request is issued, unless the entry is retired. Thus, when the write request comes, it directly updates the entry in the reorder buffer and then updates the DValid, PValid and parity banks according to the write algorithm. In case of the entry is retired, the write request can be issued in the same way as described in the write algorithm.

### 4.3.3.2 Read-after-Write

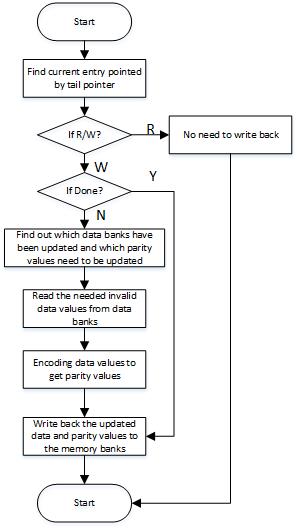
Here we consider the scenario when a read request is issued after a write request to the same location. During the write request, the scheduler allocates/finds an entry in the reorder buffer and updates the corresponding parity banks according to the write algorithm. Thus, when the read request is issued, it could find its entry in the reorder buffer and directly read from the entry. In this case, the read request could be issued in a very short time.

### 4.3.3.3 Write-after-Write

Here we consider the scenario when another write request is issued after a write request to the same location. In general cases, the parity bank value is invalidated during each write request, there’s no need to calculate the parity bank values. The worst case happens when all the other data banks values are valid in the reorder buffer, and each time the write request need to recalculate the parity bank values.

4.4.4 Write-back 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). The following flowchart shows the steps of a write-back procedure.



### 4.3.5 Refresh algorithm

Schedule refresh for every period of time or when the buffer is full, for invalid DB entry, read data from data banks and update PB and then commit.

## *4.4 Code design with Ramulator*

Ramulator is an event driven and cycle accurate simulator that is generally used in industry and academia to simulate DRAM behaviour. This extensively flexible simulator uses a configuration file that can be used to simulate HBM memory accesses. For example, the number of channels, ranks, speed, and size can be specified in the configuration file. We use a trace file from the SPEC2006 benchmark that contains read and write requests to a specific address as the input to the simulator.

We add the reorder buffer to the memory controller: every time a request is issued, it is first fed into the reorder buffer and rescheduled and then it generates additional parity bank access requests. The regenerated requests are then fed into the HBM memory module.

We assume the open page policy during simulation. We also use GEM5 simulator together with the reorder buffer, so that the memory trace could be scheduled more easily.

## 4.5 Improvement over default Ramulator controller

In the process of evaluating the code design using the Ramulator platform, we have changed the controller design to improve the access procedure performance of the coded memory. The change in the design comes with great benefits and some caveats as documented below:

Pros:

* Utilize coded data storage architecture to reschedule the memory requests to reduce bank conflicts.
* Add memory reorder buffer to bypass read requests.
* Schedule the refresh instructions to the parity banks.

Cons:

* Add memory overhead to store the reorder buffer.
* It takes longer time to dispatch memory requests.
* Cannot increase throughput when the memory requests are distributed among different rows.

4.6 Experiment Methodology

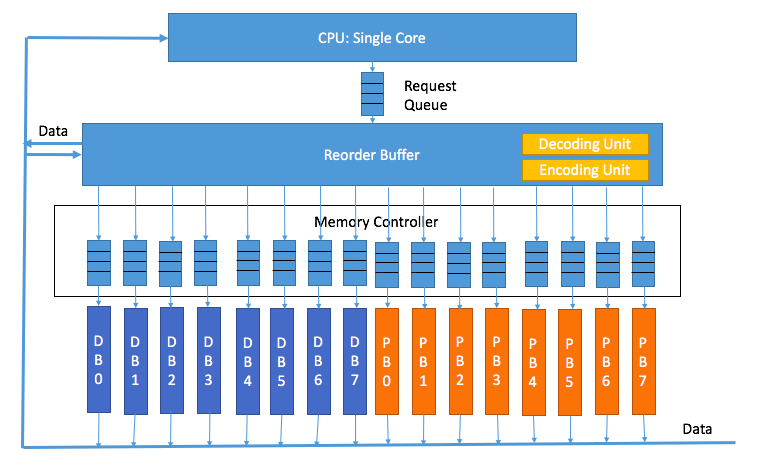
In this section, we present our experiment setup. We run the simulation with SPEC2006 benchmark and the LTE, UTMS application traces. Generally, SPEC2006 is a single-core benchmark, thus we look at the results with the simulator running with a single-core processor. For the application traces, we want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

In this section, 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.

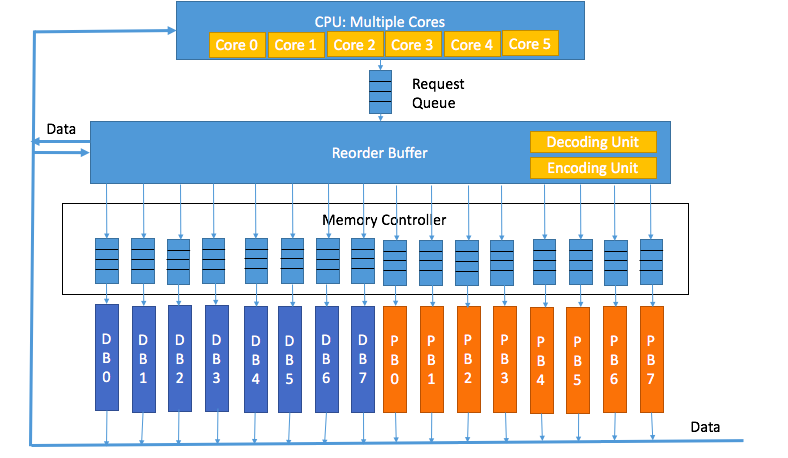
4.6.1 Architecture Diagram with single-core processor

The SPEC2006 benchmark is running with a single-core processor as shown in the figure below. 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.



4.6.2 Architecture Diagram with multi-core processor

For the application traces, we run the simulator with a multi-core processor, specifically 6 cores. 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 of each cycle, the simulator will pop multiple requests from the request queue, all the requests are issued in the same cycle.



# *5 Key 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 9 - Figure 11 captures result of comparison of the performance improvements across different benchmarks.

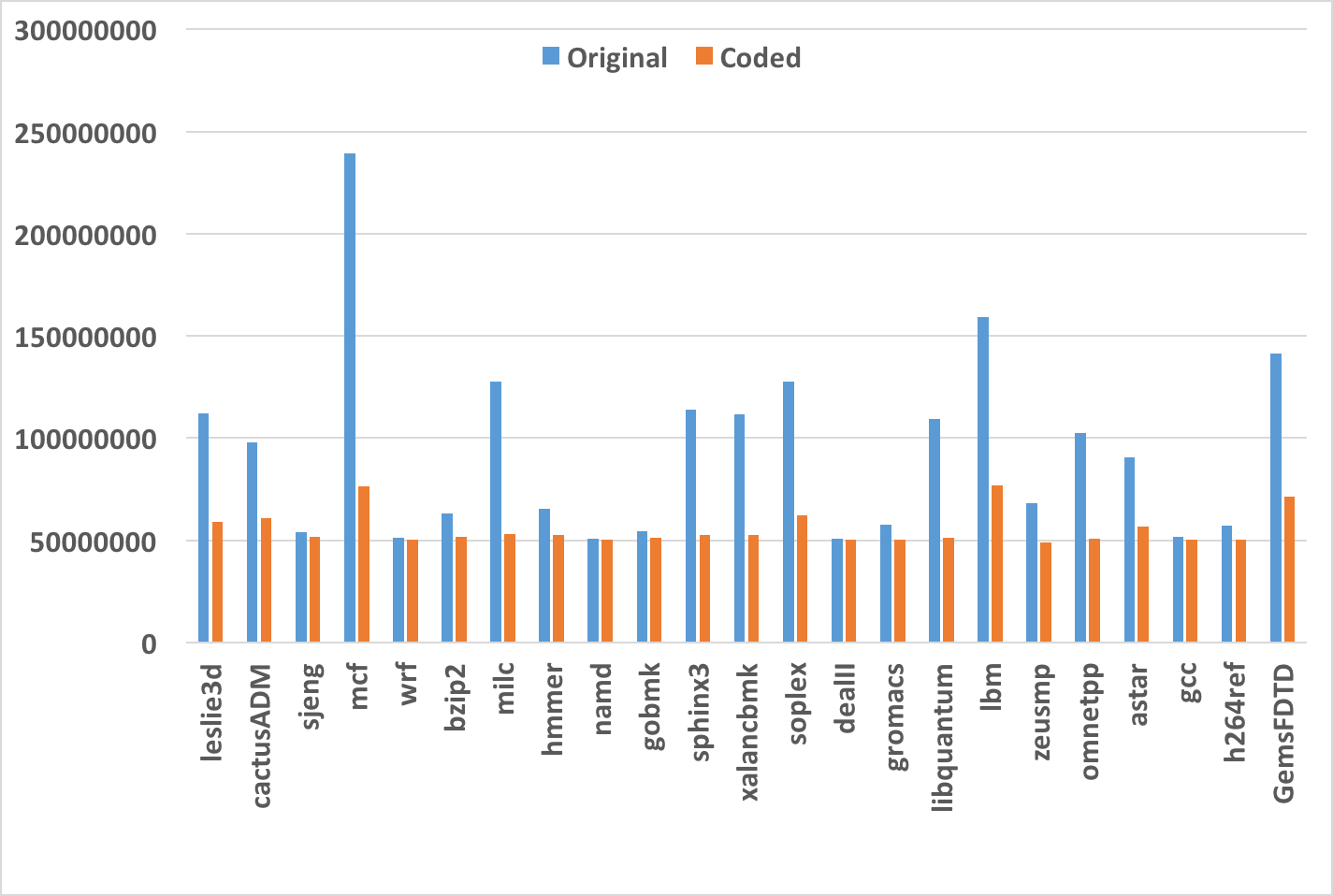
5.1 Simulation Assumptions

1. Assuming decoding and encoding delay of 0 cycle
2. Reorder buffer could be accessed at any time, the delay to find the accessed row in the buffer is 0 cycle.
3. Assuming no improvements for write requests.
4. Write-back to memory from reorder buffer is done when the corresponding row is evicted.

5.2 Simulation Cycles

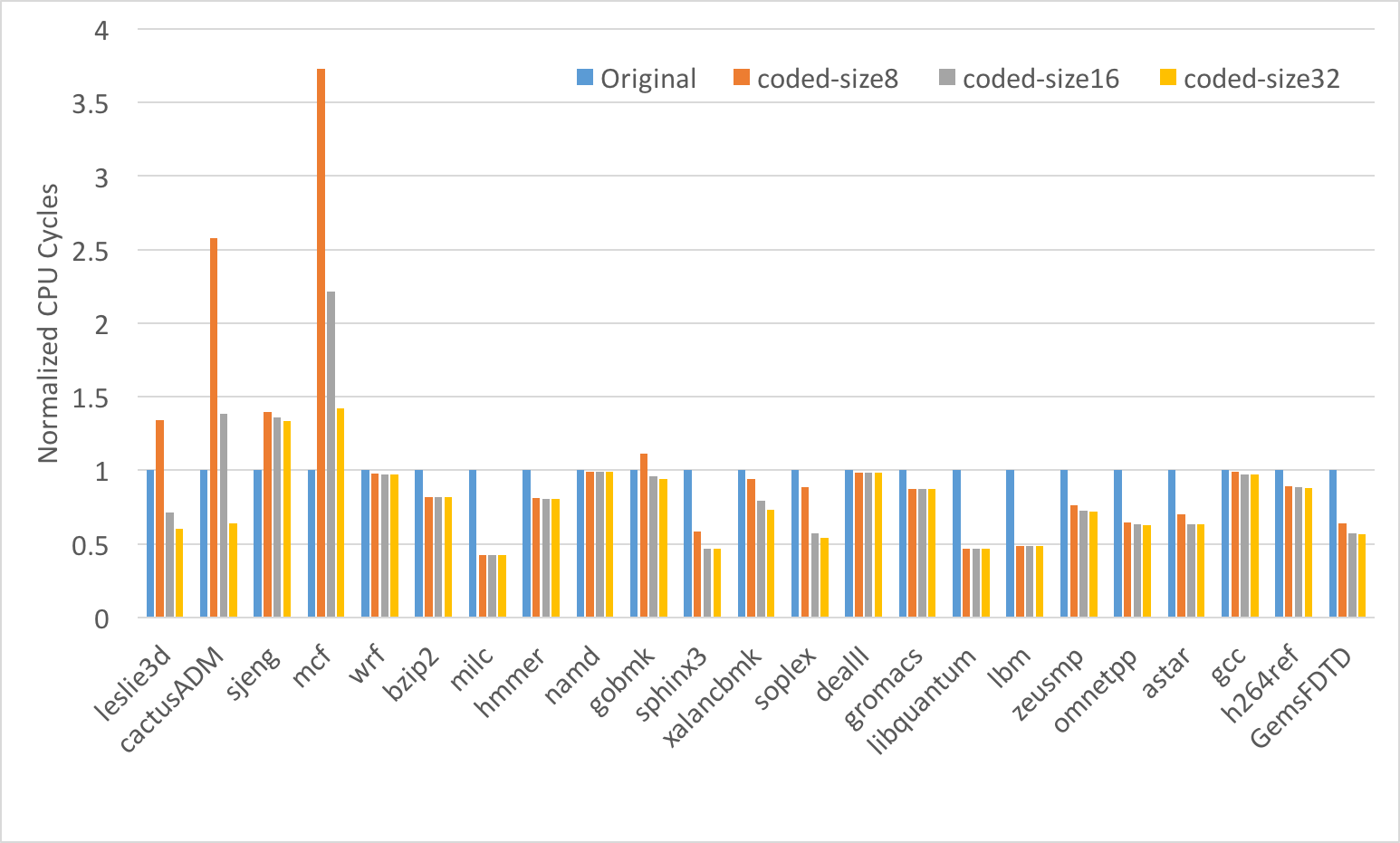
In this section, we present the simulation cycles of the baseline model and the modified model. The modified model is running multiple times with different reorder buffer size, specifically, the size varies between infinite, 128, 64, 32, 16,8.

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.



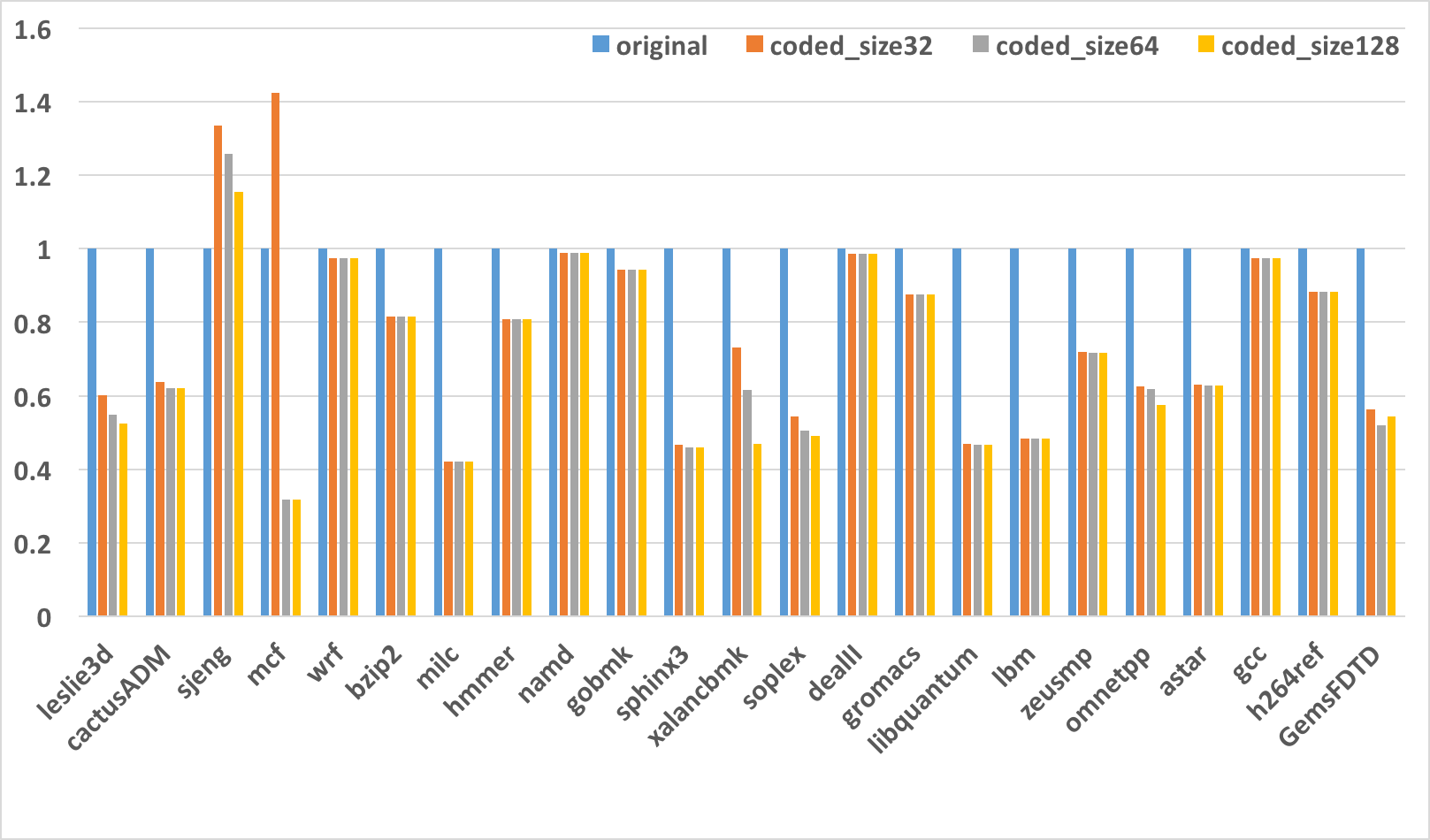
**Figure 9 :** Simulated Number of CPU cycles: Baseline HBM versus Coded HBM across

different benchmarks (with infinite reorder buffer size)



**Figure 10:** Simulated Number of CPU cycles: Baseline HBM versus Coded HBM across

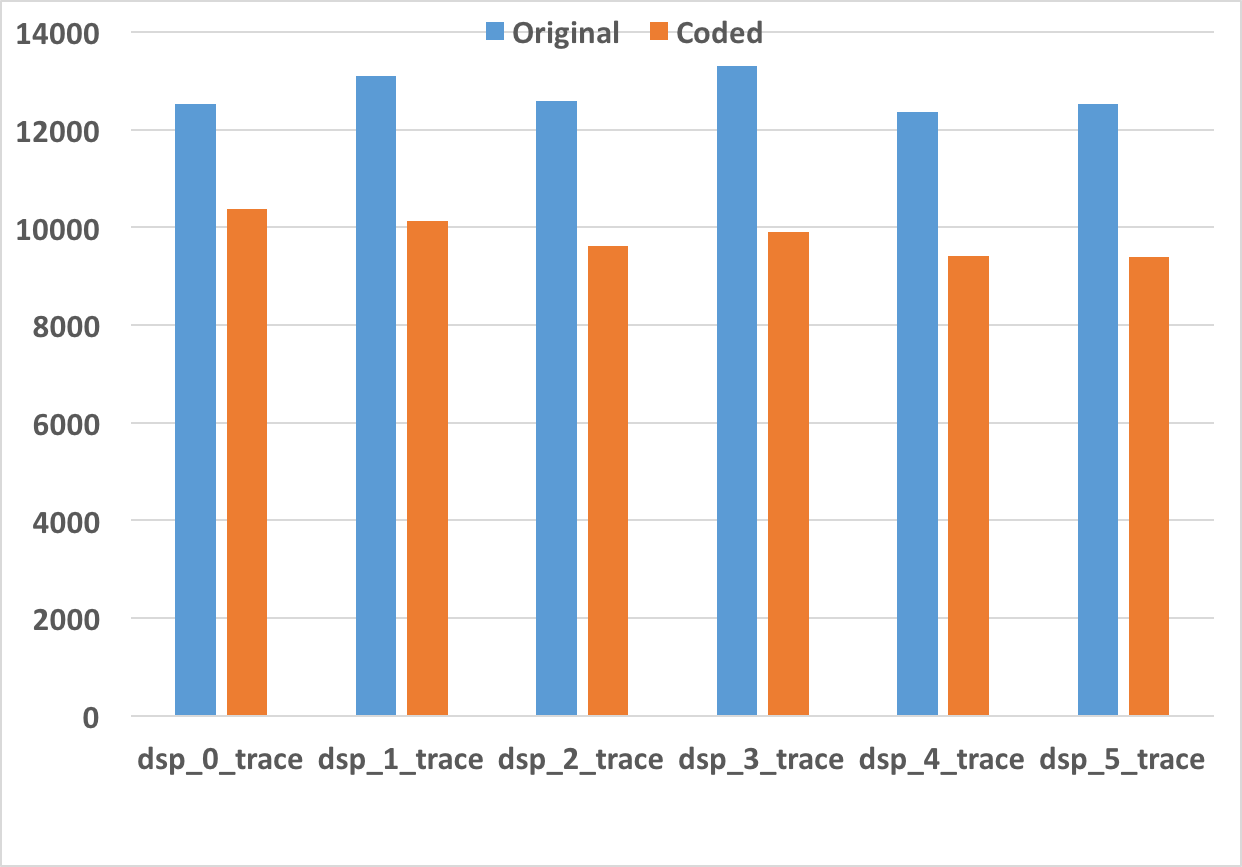
different benchmarks (with reorder buffer size of 8, 16, 32)



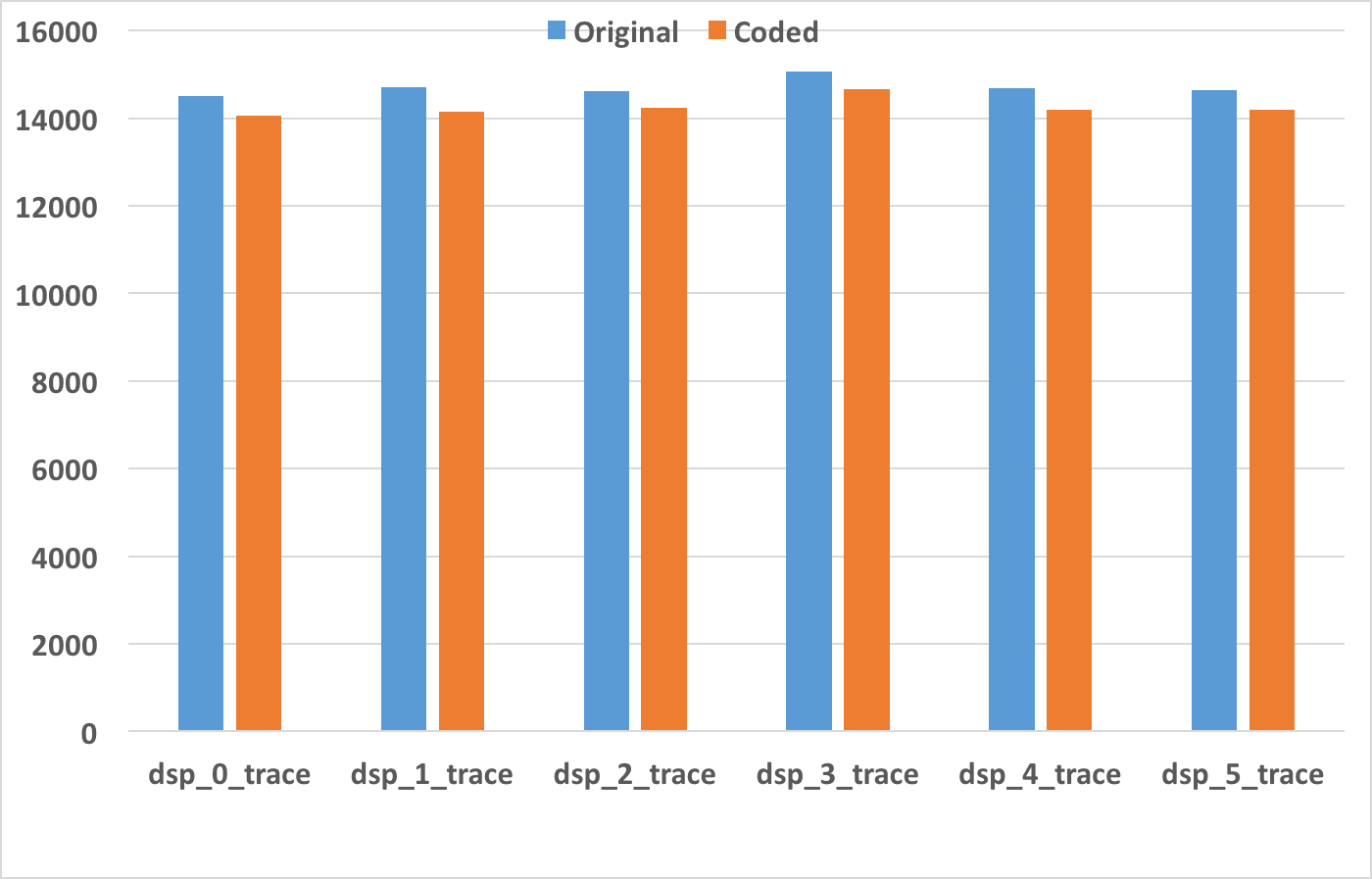
**Figure 11:** Simulated Number of CPU cycles: Baseline HBM versus Coded HBM across

different benchmarks (with reorder buffer size of 32, 64, 128)

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 don’t 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 12 :** Simulated Number of active dram cycles: Baseline HBM versus Coded HBM across application-driven LTE traces (with reorder buffer size of 10)



**Figure 13 :** Simulated Number of active dram cycles: Baseline HBM versus Coded HBM across application-driven UTMS traces (with reorder buffer size of 10)

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.

5.3 Cost Analysis

The chart below shows the reorder buffer size under different depth.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Depth | 8 | 16 | 32 | 64 | 128 |
| Size | 8KB | 16KB | 32.25KB | 64.5KB | 129KB |

# *6 Summary and plan for next phase*

Our focus during this phase included emphasis on the code design and optimization of the access scheduler. We also made significant process in improving the design emulation on the Ramulator platform. Our immediate next goals are towards optimization of size of the reorder buffer, algorithm development to do error correction and out-of-order data return. This will also be accompanied by deeper design analysis and optimization of the code design to meet the performance and evaluation criterias that were devised in Phase II and reduce the cost of the solution.

# 7 References

[1] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, “Erasure Coding in Windows Azure Storage,” *Atc*, p. 12, 2012.

[2] J. Kim, M. Sullivan and M. Erez, "Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory," *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Burlingame, CA, USA, 2015, pp. 101-112.

[3] Intel Corp., “Intel Xeon Processor E7 Family: Reliability, Availability, and Serviceability,” 2011.

[4] International Business Machines Corp. (IBM), “Chipkill Memory,” http://www-05.ibm.com/hu/termekismertetok/xseries/dn/chipkill.pdf, Tech. Rep., 2012

[5] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, “XORing Elephants: Novel Erasure Codes for Big Data,” *39th Int. Conf. Very Large Data Bases, VLDB*, vol. 6, no. 5, pp. 325–336, 2013.

[6] K. Shanmugam, D. S. Papailiopoulos, A. G. Dimakis, and G. Caire, “A repair framework for scalar MDS codes,” *IEEE J. Sel. Areas Commun.*, vol. 32, no. 5, pp. 998–1007, 2014.