**Dynamic Coding for Improved Performance of Memories**

PI: Sriram Vishwanath

Lead PhD: Hardik Jain

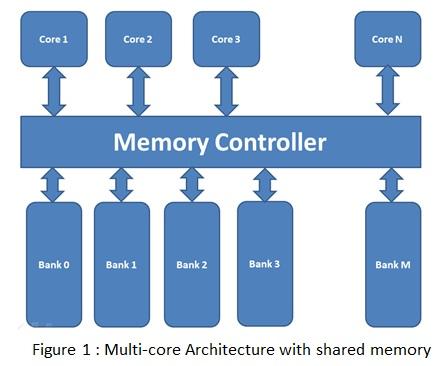
Lead Masters: Casen Hunger

Consultant: Ankit Rawat

**Abstract:**

This phase of the project explores use of coding techniques to the memory systems. Various code designs for the system are explored. We propose techniques of dynamically coding a localized high frequency access region and prefetching the data / codes to increase the efficiency of the system. For given design parameters, we achieve 4 consecutive accesses per bank per cycle.

**1. Introduction**

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 1 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.

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 report, 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. 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.

**2. Main Results**

**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.

**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 – 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.

**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.

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 αa0+βb0 for any integer value of α and β.The least complex design for the decoder would be for taking α = 1 and β = 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 2 captures the effect that coding has on both of these parameters for varying numbers of cores. Figure *2a* and *2b* 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 *2c* and *2d* 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.

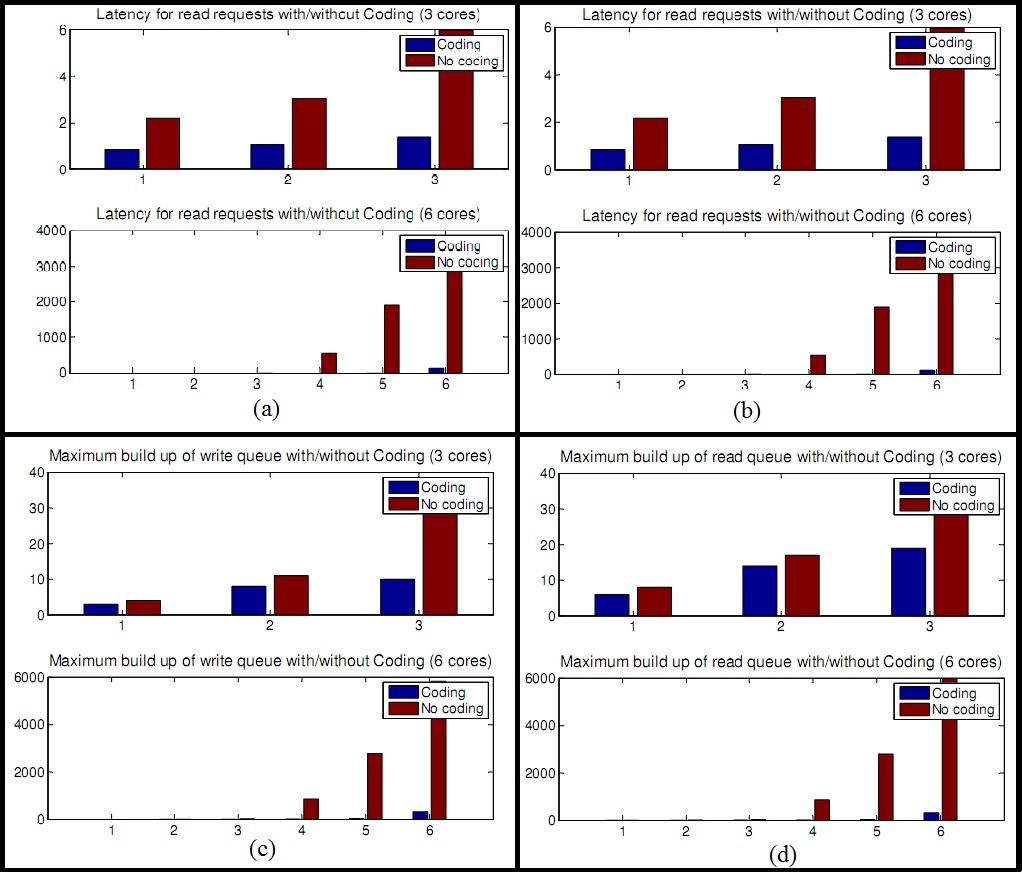


Figure 2

Design II

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

· Read access : 4 per bank in one cycle

· Write access : 2 per bank in one cycle

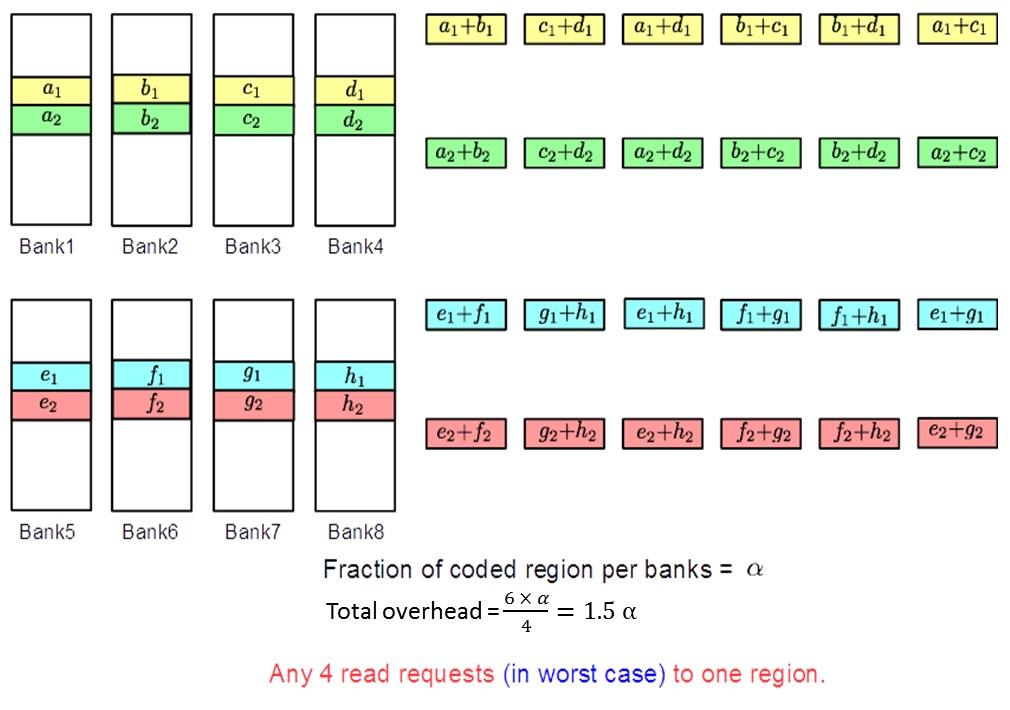
· Memory size 8 kB – 256 kB

· Number of Banks : 8

· Memory overhead : 15%

· Parity banks : 5 or 6 shallow banks for code storage

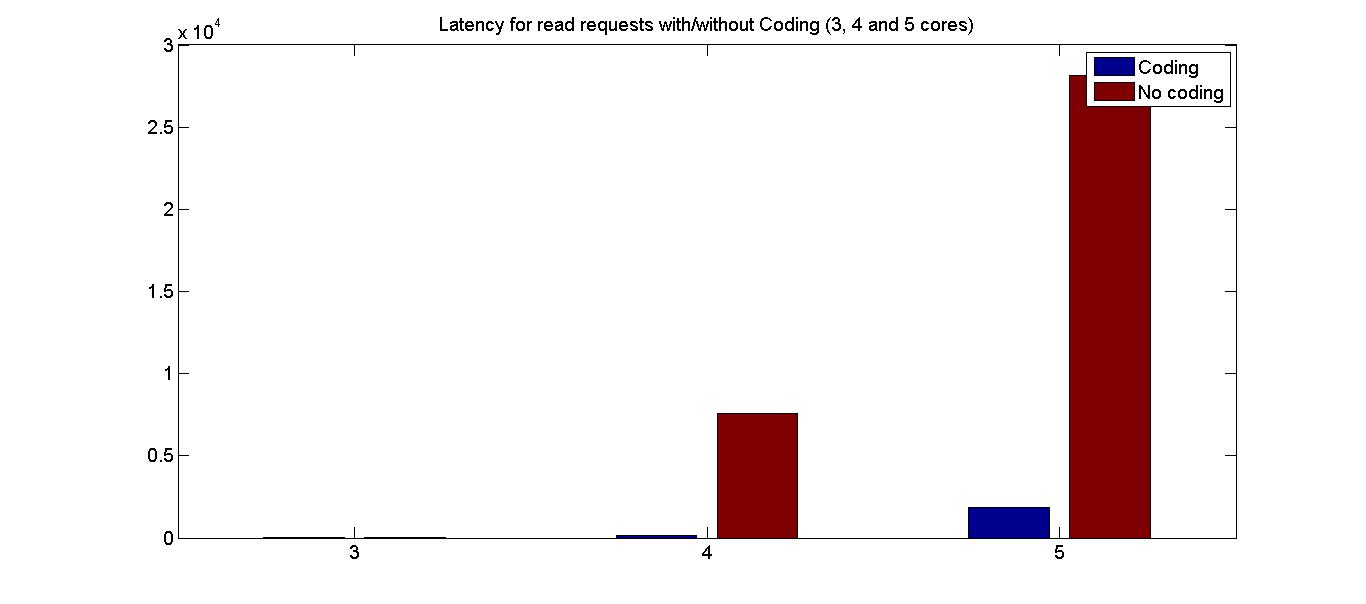
Using the above parameters, we design coding scheme to efficiently store the codes and achieve the objective. We use the concept of batch codes to code specific areas within each of the banks. This allows us to serve multiple accesses to the coded region using the parity banks. With this scheme, we guarantee 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 1-8 are the coded region. These region are assumed to be of *\alpha* fraction of the memory.



**Figure 3**

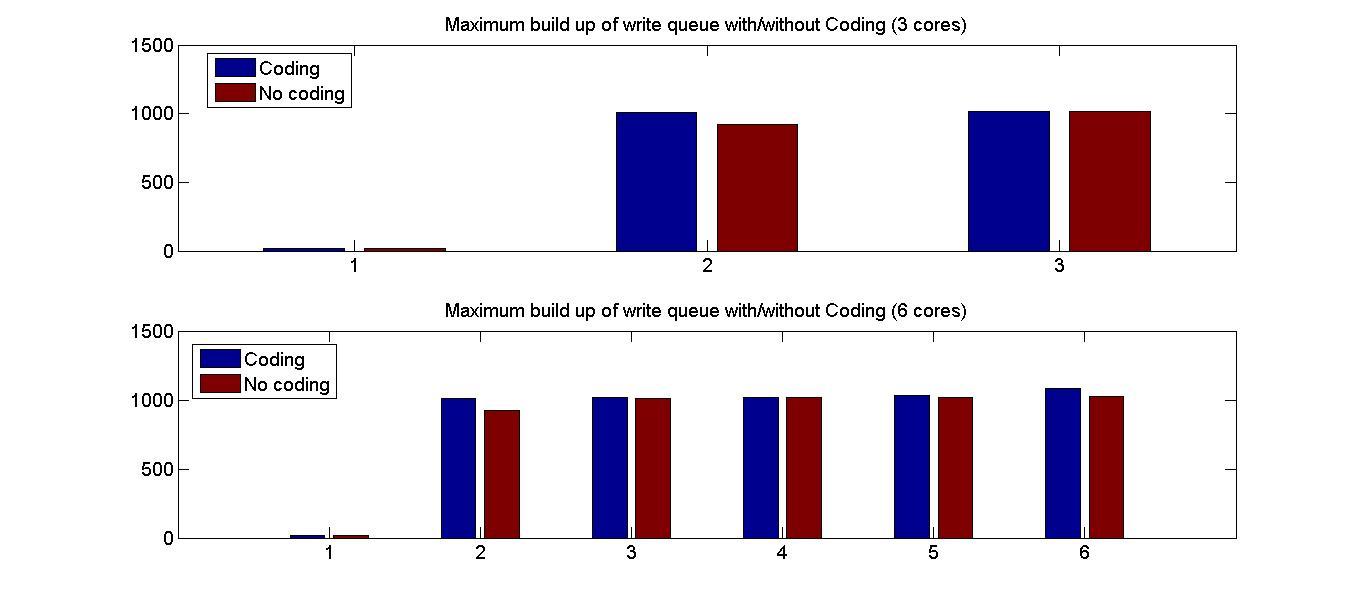
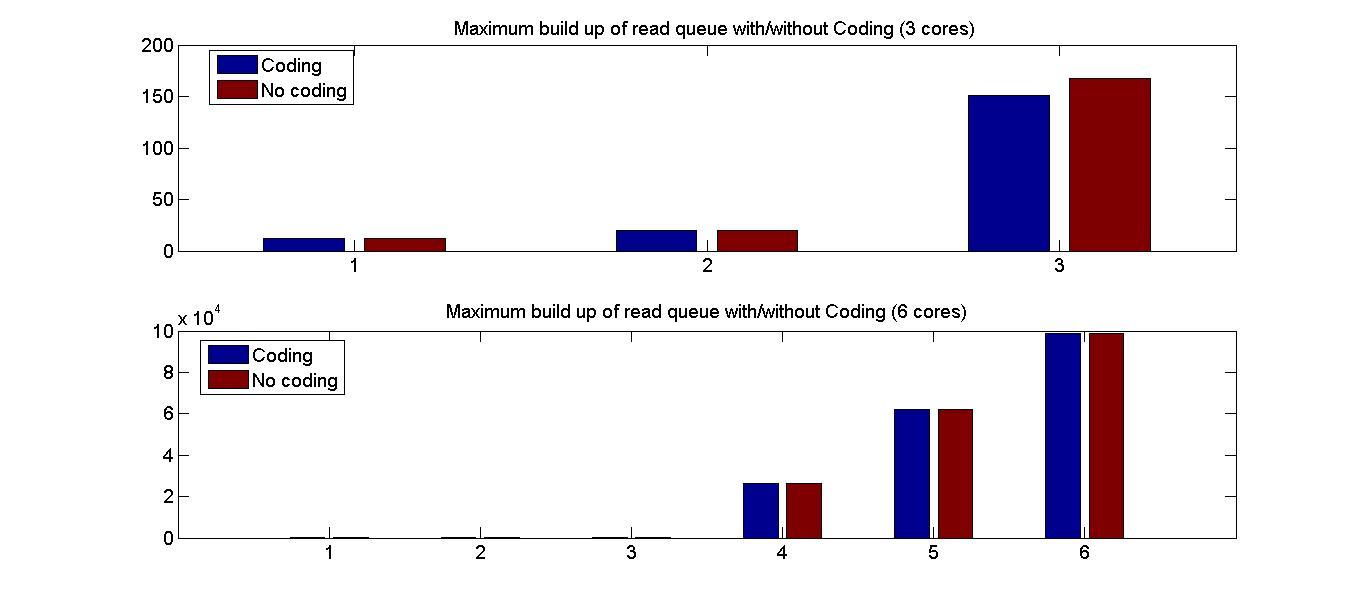
Figure 3 presents storage scheme for this design is presented.

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 10 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,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.



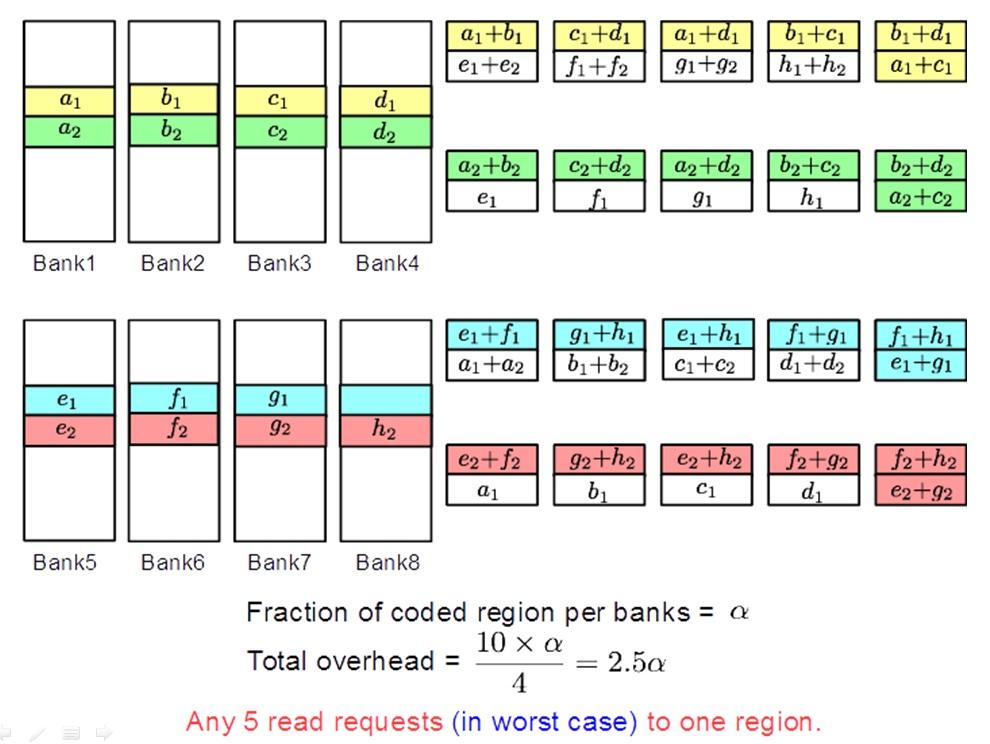
Worst Case analysis : The code scheme falls off to 4 access in a cycle when there are non-sequential and non-consecutive access to the memory banks. For example when the access pattern is to a1, b8,c9,d15. Since we do not code this combination, it does not get benefit of extra access. However in this 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 4, 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.



**Figure 4**

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 region’s parity banks are in use. Figure 5 shows the storage pattern of the codes in the bank. The overall overhead in this system is 2.5 \alpha.



**Figure 5**

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 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.

Coding Performance Enhancement

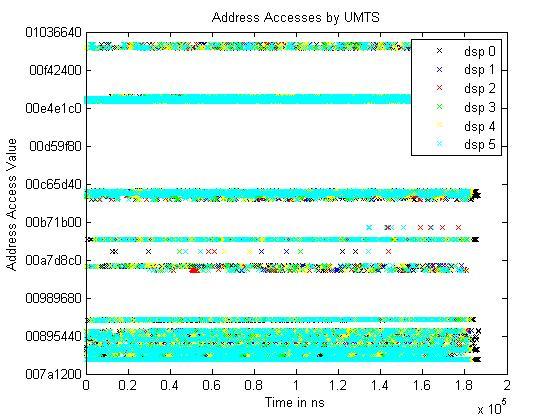
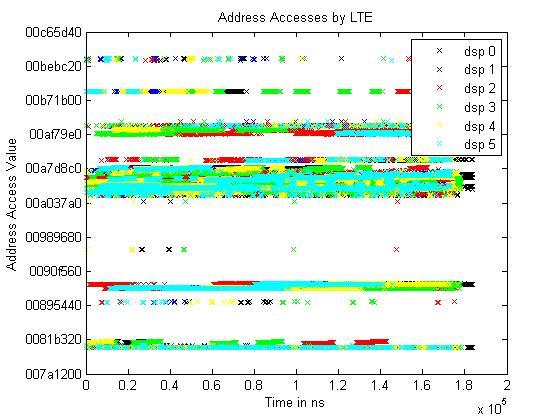
In this section, we explore the technique of dynamic coding in order to reduce the memory and access overhead associated with the parity banks. We first discuss the scheme of dynamic coding and follow it by discussing the potential benefits of prefetching the codes.

*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 6 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 –axis). 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 X 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.

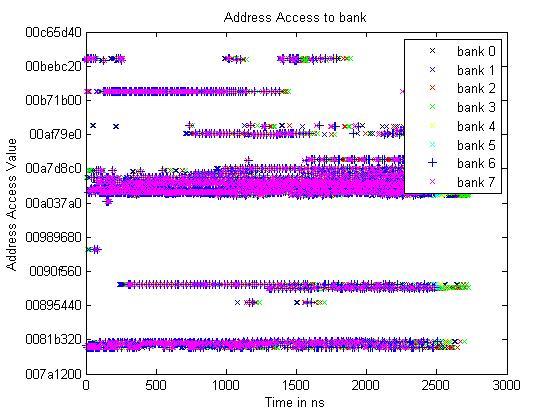


**Figure 6**

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 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.

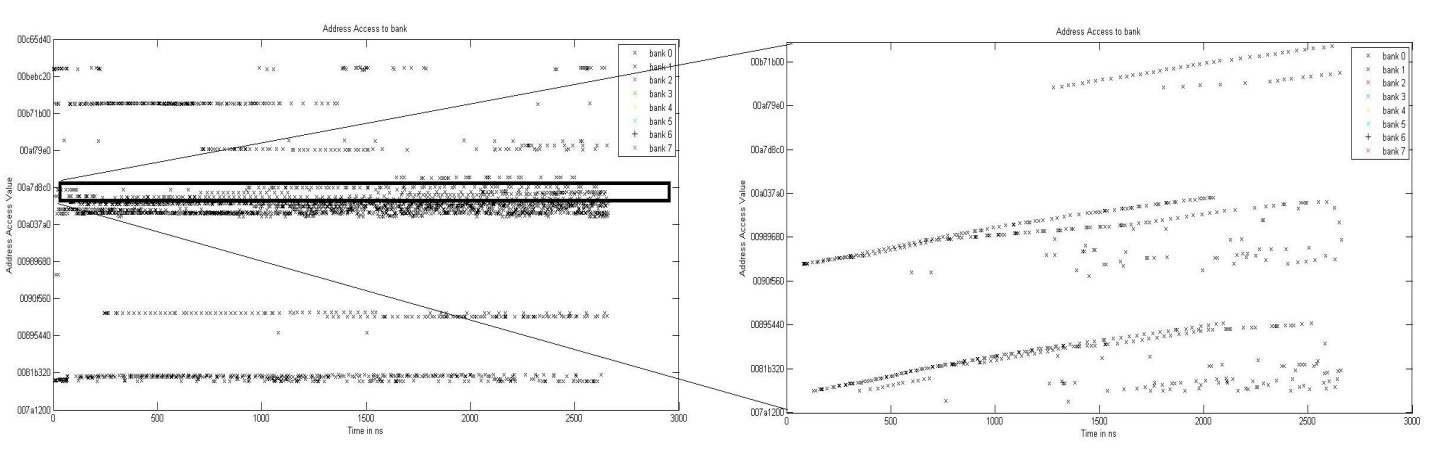
Dynamic coding requires the system to divide the memory into subregions and to keep track of accesses in these sub-regions. Once the number of accesses to a sub-region reaches a given threshold, it must then make this region the currently coded area. We propose this mechanism based on window concept. 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.



**Figure 7** shows the access to banks

*Prefetching Codes****:***

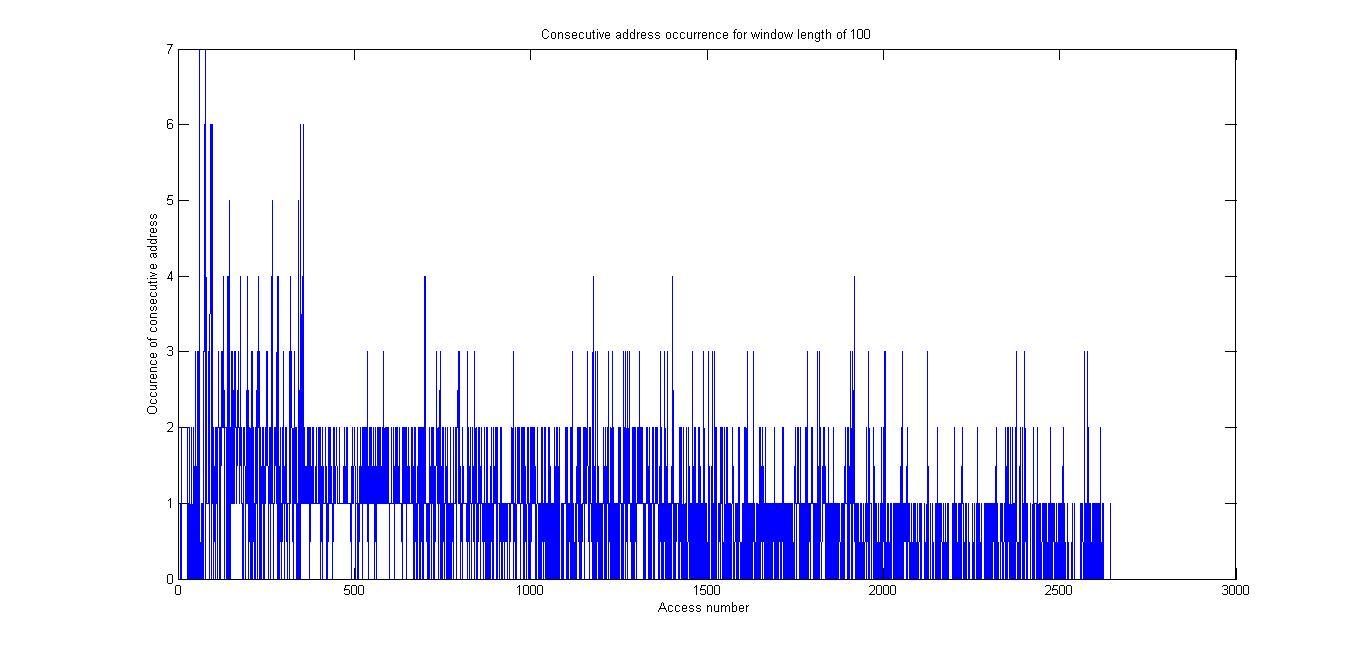
The technique of dynamic coding reduces the memory overhead by exploiting the localized nature of memory accesses from the cores. In this section, we explore prefetching the coded data to reduce the access overhead caused for fetching the codes. This is done by exploiting the gaps in the memory access 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.



**Figure 8**

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 8 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 9**

In figure 9, 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.

**3. Next Steps:**

The next steps of this effort are described in greater detail within the proposal to delve deeper into a proof of concept.