Dynamic Coding for Improved performance of Memories

Abstract

Memory systems work hard to keep up with access requests from cores. Growing computer sizes, heterogenous systems and increasing level of integration has increased more. Performance focussed systems use enhancements like multi-port memories to increase the access capacity. However, they come with a cost in terms of area, complexity and cost of redesign and rebuilding a system. In this paper, we explore a mathematical solution to the problem where we explore an efficient memory storage and reterival mechanism for efficent access. We first analyze the request pattern of general memory controller and a application-specific memory controller.
We then provide a mathematical approach to storing the data in specific way to achieve higher access rate. We call this specific way of storing the memory as Algorithmic Memory. We discuss methods to design codes and provide example designs for 8 bank memory systems.
At last, we analyze and compare the improvement of coded memory with general memory. We present a significant improvement in critical word read and write latency with coded memory. We also provide intuitions derived from this analysis which can help the system designers to efficiently use Algorithmic memory implementation.

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. Figure 1 shows the difference in the performance of processor and memory. This difference is increasing as we move along the Moore's law. The performance of a program is a combination of performance of processor and memory. With processors becoming faster and multi-core, this difference is further increasing. Slower memory slows down the whole system. It is for this reason most beneficial to keep memory accesses latency as short as possible. Long-latency memory accesses occur when there is a miss in the last level cache(LLC). This leads to an access to shared memory. The processor has to wait for the shared memory to return after the access. This wait creates a stall in the processor.

In case of multi-core processor architecture, the access latency to the shared memory increases due to contention from requests issued by other cores. This results in formation of large access request queues waiting to be served by the slow shared memory. Figure 2 shows a general multi-core architecture where N processor cores share a memory consisting of M banks. Requests from each core is sent to the memory controller first, which then arbitrates and in turn issues requests to the memory. Because the memory controller has parallel access to all M banks, a bank queue is used for each individual bank request. These bank queues are served every memory clock cycle and the acknowledgement with data (in the case of a read) is sent back to the processor.

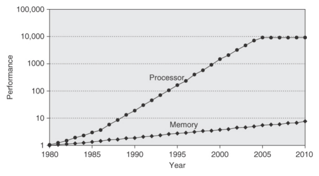


Figure 1: The gap in performance, measured as thedifference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over time. Taken from Computer Architecture: Quantitative Approach [1]

In the scenario where multiple cores request access to the memory locations which belong to the same bank, the memory controller puts these request in the respective bank queues. This contention between cores to access from the same bank is known as a bank conflict. The bank conflicts mean that the requests will be served sequentially and some of the cores will have to wait longer for their request to execute. As the number of bank conflicts increase, the latency for memory accesses to the bank increase, thus an 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 solution is to use 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. The additional bank accesses results in a decrease in 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 present various design to store the redundancy across the parity banks and evaluate

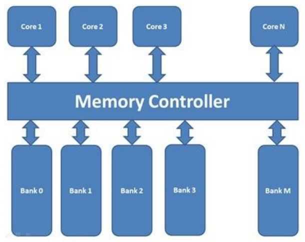


Figure 2: General Multi-core Architecture with shared memory

each of them for their cost. We also implement these designs using systemC and regress it through memory traces from real multi-core system.

2. Background and Related Work

3. Code Design

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 correction in the transmission of data to increased compression for data storage. We aim to extend the benefits of coding theory to improve the efficiency of 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 access 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 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 each of the three code designs explored in this project.

3.1. Design Parameters

In this section, we discuss various parameters that we consider to design and simulate the efficient code storage in this project.
Memory overhead: The crucial cost in coded memory system is to store the compressed redundancy or the codes. The extra memory space used to store these codes should limit to 15% of the overall memory.
Memory size: The memory size and the parity storage size decide the code function to be used to essentially compress the redundant data. This design considers a portion of memory to be coded.
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 row location within the bank.
Cache line size: The memory accesses are bundled in a burst as a cache line is evicted 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. This parameter is not used in the design of the coding 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 1.54 ns as the access rate. This would mean that the clock rate to memory would be at 650MHz. This parameter is required to simulate the performance for the shared traces.

3.2. Designs

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.
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 its 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 a_0 and b_0 could be stored as $\alpha a_0 + \beta b_0$ for integer values of α and β which belong to any Galois Field. 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 a_0 to a_n, b_0 to b_n , c_0 to c_n and d_0 to d_n

elements respectively. The possible codes for these memories could be:

$$a_i + b_i, b_i + c_i, c_i + d_i \text{ and } c_i + a_i \text{ for i = 0 to n} \quad (1)$$

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:

$$a_i + b_i + c_i + d_i \text{ for i = 0 to n} \quad (2)$$

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 banks using elements from different banks. We call this type of coding as Interbank Coding. We also explore the orthogonal way of coding, i.e. intra-bank coding where we use the memory elements from the same bank to generate codes.

Following are the objectives used in code design:

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

In the following sub-sections, we discuss 3 designs for storing coded data. Table 1 compares these designs for various parameters and associated costs.

3.2.1. Code design I This design uses the concept of batch codes to code specific rows of memory 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 read 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 regions are assumed to be of α fraction of the memory.

Best case scenario 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 $a_1, b_1, c_1, d_1, a_2, b_2, c_2, d_2, a_3, b_3, c_3, d_3$. Here, we can serve a_1, b_1, c_1, d_1 using a_1 with the parity banks $a_1 + b_1, b_1 + c_1, c_1 + d_1$ and serve a_2,c_2,d_2 using b_2 from data bank and b_2+c_2,b_2+d_2,a_2+c_2 from parity banks. Lastly, we can serve c_3 and d_3 using data bank 3 and data bank 4.

Worst Case scenario 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

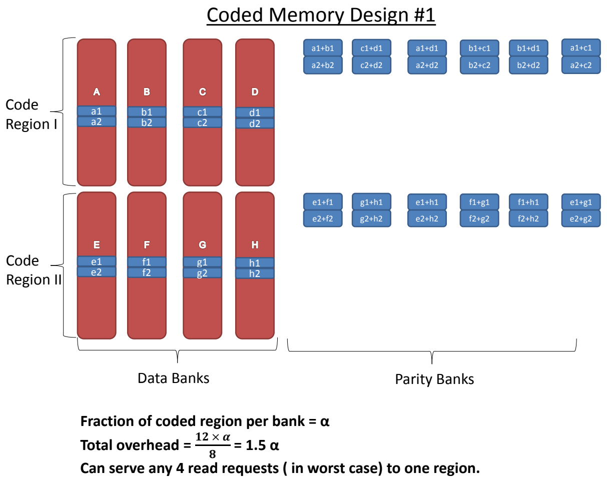


Figure 3: Code Design I

the access is requested for a_1, b8, c9, d15. Since a code for this combination is not present in parity banks, the benefit of parallel access is unused. 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. We discuss this mechanism further in section 4.4. Figure 6 shows the storage pattern of the codes in the bank. The overall overhead in this system is 2.5α.

3.2.2. Code Design II The second design, presented in figure 4, improves over first design by allowing 5 read accesses per bank per cycle. This design also divides banks into two regions. The first region is Bank 1 to Bank 4 and 5 corresponding Parity banks. The two regions in figure 4 are upper 9 banks forming one region and lower 9 banks forming another. This design allows intermix storage of parity among regions. The design uses 5 parity banks per region. The data in this scheme is coded for both inter bank and intra-bank. The intra-bank codes are stored in the alternate parity bank region. This allows usage of parity banks from other region if they are available.

Best case scenario 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 $a_1, b_1, c_1, d_1, a_2, b_2, c_2, d_2, a_3, b_3, c_3$. Here, we can serve a_1, b_1, c_1, d_1 using a_1 with the parity banks $a_1 + b_1, b_1 + c_1, c_1 + d_1$ and serve a_2, b_2, d_2 using b_2 with the parity banks $a_2 + d_2$ and $b_2 + d_2$. Lastly, we can serve c_2 and d_3 using bank 3 and bank 4.

Worst Case scenario analysis: The code scheme can do 5 accesses 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 $a_1, a_6, a_9, a_{15}, a_{20}$, we can perform these 5 reads with the help of coded banks. We can use the prefetching mechanisms discussed later to look ahead in the queue and pre-fetch codes

Design	Max Read per bank	Max Write per bank	Locality	Rate	Memory Over-head	Logical Com-plexity
I	4	2	2	$2/5$	1.5 α	Low
II	5	2	2	$2/5$	2.5 α	Medium
III	4	2	3	$1/2$	α	Medium

Table 1: Comparison of design with respect to the performance parameters and associated cost

from parity banks for further accesses.

3.2.3. Code Design III The two designs discussed above achieve a rate of $2/5$. Here, we explore a code design which achieves a rate of $1/2$. This design requires 9 data banks and 9 parity banks as shown in figure 5. It also has a comparatively higher locality of 3. That is, it requires the memory controller to "know" two out of three data elements to decode the third. This also helps us achieve 4 simultaneous read access per bank per cycle and 2 simultaneous write access per bank per cycle. The memory overhead here is less (just α) compared to the previous designs. However, it possesses higher logical complexity because of increased locality. Example cases for this design are described below :

- 4 reads for a_0: 1 read from a_0, 1 read from (a_1, a_2, $a_0 + a_1 + a_2$), 1 read from (a_3, a_6, $a_0 + a_3 + a_6$), and the 4th read from (a_4, a_8, $a_0 + a_4 + a_8$).
- 3 reads for a_0: 1 read from a_0, 1 read from (a_3, a_6, $a_0 + a_3 + a_6$), and the 3rd read from (a_4, a_8, $a_0 + a_4 + a_8$).
  1 read for a_1: 1 read from a_1.
- 2 reads for a_0: 1 read from a_0 and the 2nd read from (a_3, a_6, $a_0 + a_3 + a_6$).
  2 reads for a_1: 1 read from a_1 and the 2nd read from (a_4, a_7, $a_1 + a_4 + a_7$).
- 2 reads for a_0: 1 read from a_0 and the 2nd read from (a_3, a_6, $a_0 + a_3 + a_6$).
  1 read for a_1: 1 read from a_1.
  1 read for a_2: 1 read from a_2.

Figure 7 describes the system with 9 data banks and 9 parity bank. Since most systems are implemented with number of banks as 2^n for some n. We present an example of the code with 8 data banks in figure 8. For using 8 data banks, we drop the bank I. We also ignore the data from Bank I for constructing parity. So, three of the parity banks have the locality of 2, while the rest of the parity banks have locality of 3. The new scheme for 8 data banks has 9 parity banks. Since the locality is 3 here in this design, i.e. , each parity is made up of combination of 3 data banks, we need to make sure that all three requests are in one line to be able to use the parity bank. For example parity bank 0 contains A+B+C. So, the following scenarios arise:

- *Scenario I*: 1st request of A and 1st request of B are in same row. Then, we can search for a request in the same row for bank C by doing a look ahead.
- *Scenario II*: 1st request of A and 1st request of C are in same row. Then, we can search for a request in the same row for bank B by doing a look ahead.
- *Scenario III*: 1st request of B and 1st request of C are in same row. Then, we can search for a request in the same row for bank A by doing a look ahead.

So, the simple pseudo code for doing this would be :

```
for each data bank
    for each auxiliary bank1 of data bank
            Look ahead in auxiliary bank2 and check if 3
        end
    end
```

Example: -
For **data bank** to be **A**
auxiliary bank1 goes from [B C D G A E]
auxiliary bank2 goes from [C B G D E A]
The element A is just there in **auxiliary bank1** and **auxiliary bank2** to maintain the symmetry because A + E has locality of 2.

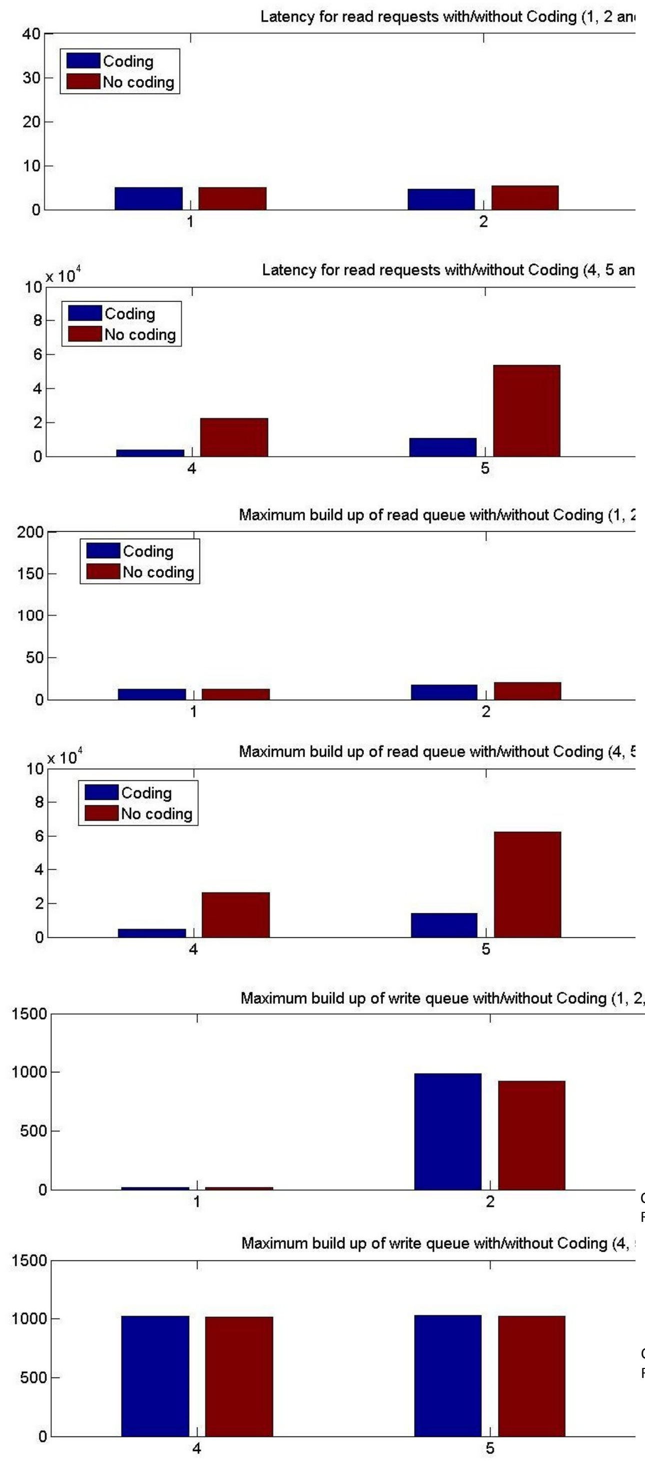


Figure 4

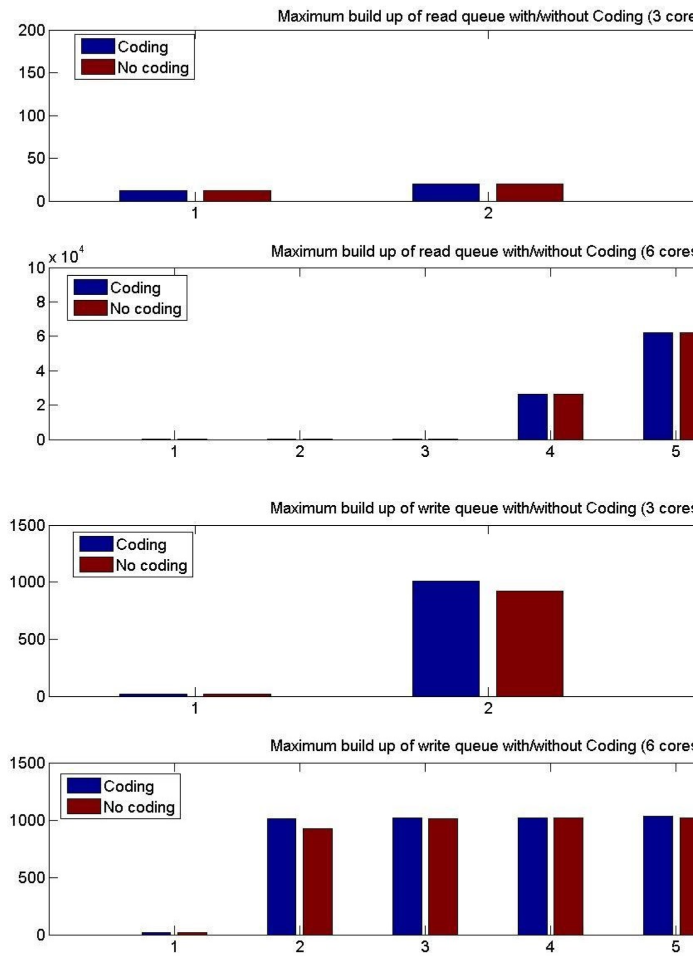
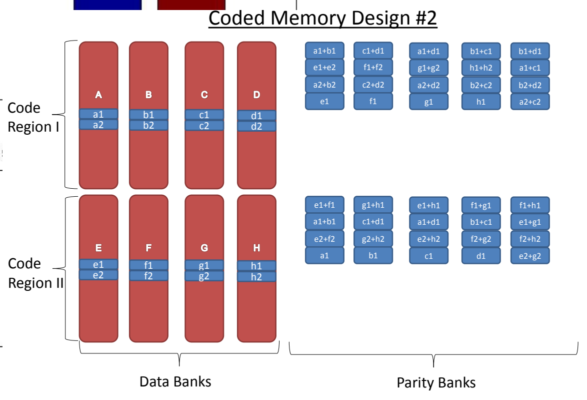


Figure 5: Comparison of Design II with No coding case

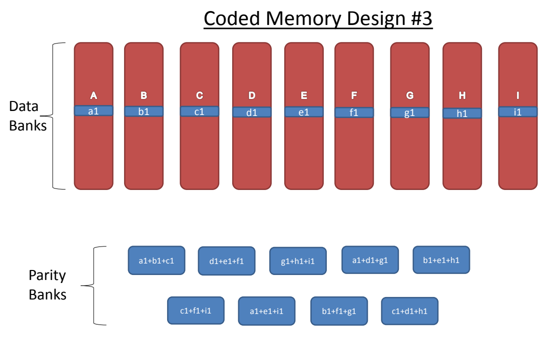


Fraction of coded region per bank = α

Total overhead = $\frac{10 \times 2\alpha}{8}$ = 2.5 α

Can serve any 5 read requests (in worst case) to one region.

Figure 6: Code Design II

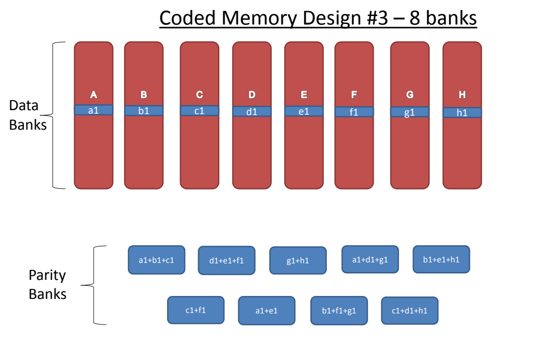


Fraction of coded region per bank = α

Total overhead = $\frac{9 \times \alpha}{9}$ = α

Can serve any 4 read requests (in worst case) to one region.

Figure 7: Code Design III



Fraction of coded region per bank = α

Total overhead = $\frac{9 \times \alpha}{8}$ = 1.125α

Can serve any 4 read requests (in worst case) to one region.

Figure 8: Code Design III with 8 data banks

4. Memory Controller Design

The coding schemes discussed in previous section is implemented using systemC. This section describes the architectural detail of how the schemes are implemented using optimized algorithms.
A general Memory controller consists of three main levels of processing.
The first level is *Core Arbiter* , the unit responsible for handling requests from cores. The second level is *Bank Arbiter* responsible for arbitrating requests to banks. The third level is *Access Scheduler* which schedules most efficient access for each cycle.
The *Core arbiter* receives request from all the cores at every clock cycle. It maintains a queue for each core. The depth of each queue is *8 requests*. That is, it can hold 8 requests from each core for service. When a queue is filled up, the controller signals the respective core as busy. So, the core cannot issue a memory request and hence stalls until the controller's status gets back to normal. This unit is also responsible for arbitration among the request for cores. The arbitrator goes through the last element of each queue and determines the priority order. It uses a two-step priority order mechanism : *first* it arranges the request in order of QoS with high QoS at high priority. *Second*, for request having same priority, it arranges them using round robin.

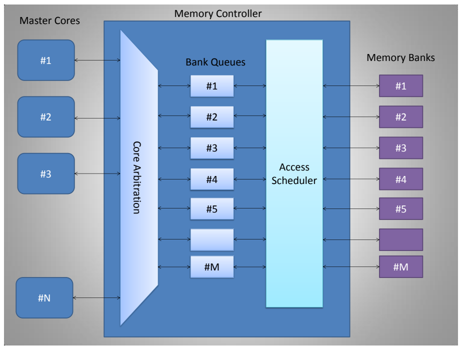


Figure 9: Architecture of Memory Controller

The *Bank arbiter* is a unit in the memory controller which contains read and write queues for each bank. The core arbitrator arbitrates requests from cores, expands them and fills them in the bank queues. The depth of bank queues in current implementation is *10*. This means that the depth of each read and write queue is 10. These queues are for each bank. In an eight bank system, there are total of eight read queues and eight write queues. This requires a total of 16x10 or 160 requests to be stored in these queues.
The *Access Scheduler* is responsible for scheduling access to the DRAM memory banks at every cycle. This logic looks at the read and write queues for each bank and forms an access pattern which is most efficient. In doing so, it goes through the requests in the bank and makes the pattern. It prioritizes the requests at the bottom of the queue however, in case of coding

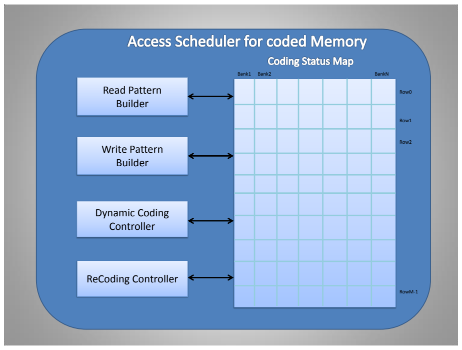


Figure 10: Access Scheduler for coded memory

00	Codes are up-to-date
01	Codes are outdated. Fresh data in Data bank
10	Codes are outdated. Fresh data in Parity bank
11	Reserved

Table 2: Code Status Map

memory systems (discussed in 4.2) it looks ahead in the queue to maximize the accesses per cycle. The write queues hold the writes until the writes are full. The reads are given priority over writes until the write queue gets filled.

4.1. Code Status Table

This part of access scheduler is a memory which holds the status of code for each row in the memory banks. As depicted in figure 10, this block's rows correspond to rows of the memory banks. The columns are equivalent of each memory bank. Each cell in this memory stores two bits. These two bits denote the status of the code for each row. Table 2 maps the value of bits to their meaning. This memory is present in the memory controller and is assumed to be accessible at a very fast rate.

4.2. Read Algorithm for Coded Memory system

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. Figure 11 describes the algorithm.
Figure 12 describes an access pattern scenario. The reads are maximized by using the parity banks. The key here is to be able to use the parity bank to serve the reads to other banks. The controller first marks *a1* to be read from data bank A. It then iterates through bank B,C and D to find out if any read can be served from the parity bank using the read of *a1*. Thus, it finds out that *b1*,*c1* and *d1* can be served using the parity banks. The controller marks the corresponding parity banks to be busy. Then it moves to bank B and serves *b2* from the data bank and searches if it can serve any request from A, C

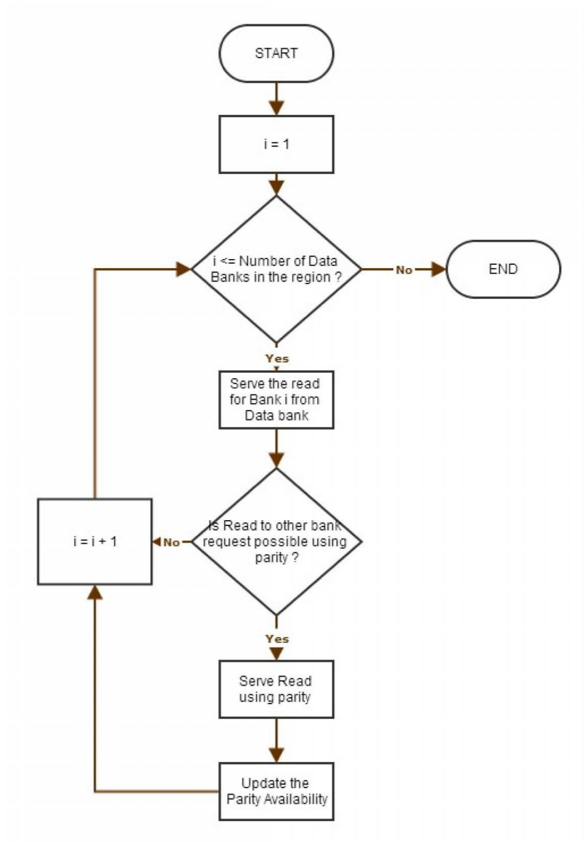

START
i = 1
i <= Number of Data Banks in the region ?
No
END
Yes
Serve the read for Bank i from Data bank
Is Read to other bank request possible using parity ?
No
i = i + 1
Yes
Serve Read using parity
Update the Parity Availability

Figure 11: Flowchart of Read Algorithm

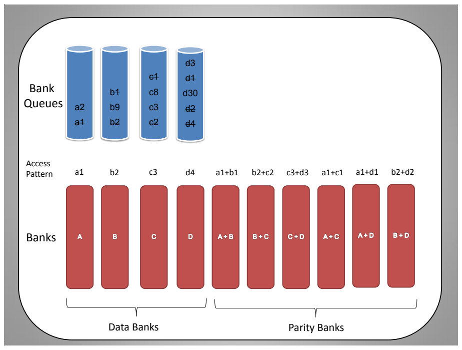


Figure 12: Figure describing access pattern formation for a region

and D using the parity banks. It serves *c2* and *d2* from the parity banks since they are available. Similarly, it goes through bank C and D to form the complete access pattern. As can be observed from figure 12, the controller serves 4 requests from bank D and 1 request from bank A. The banks are thus accessed in order of their queue size with the least queue size accessed first for forming the access pattern. However, this rule does not guarantee that there would be one access to the first bank and 4 access to the last bank. This is dependent on the accesses in the queue.
The example in the above paragraph highlights three different aspect of the read algorithm. *First*, the access scheduler tries to form the most optimum pattern for read. The accesses from the other banks are searched. The controller maintains a bitmap of availability of parity banks. It marks them busy when it assigns a access to be served from the respective bank. *Second*, the controller has the ability to search in the whole queue in order to find a request which can be served from the parity banks.
Third, the controller accesses bank in order of their queue size. This enables it to maximize the chance of last bank being able to serve 4 read requests.
we access $a1 + b1$ from the parity bank, we need to have either *a1* or *b1* in order to decode the other. Now it is quite common the case when the access across the banks are not to the same row. For example, if the accesses are to *a1* and *b5*, the parity bank $a + b$ cannot be used. However, if there was a request in the queue for *a* for *a5* or in *b* for *a1*, it can serve all the three requests in one cycle. That is, it can serve *a1* , *b5* and either *b1* or *a5* in one cycle. This approach helps increase number of accesses per cycle. In case of the described coding system, for a best case scenario of each region, we can serve 10 accesses per cycle since we have 10 banks. This in simple terms is a direct **2.5** times more than the baseline system for each region.
The above scheme of optimizing the accesses per cycle does come with a cost. It increases the chances of having out-of-order execution of memory access requests. This does not pose a problem in the case when the memory requests go out of order for different cores. However, the logic needs to take care of in-order execution of requests from each cores.

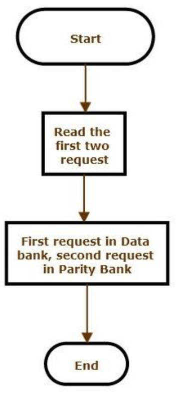

Start
Read the first two request
First request in Data bank, second request in Parity Bank
End

Figure 13: Flowchart of Write Algorithm

- Write about how we solve the out of order look ahead problem. If we solve it at all ????

4.3. Write Algorithm for Coded Memory system

The write requests are first filled in the write queues for each bank. The queue holds the requests until it is full. The controller schedules write to the memory only when the queue is full. This ensures that reads get priority over writes.

The specification of the coded system is to perform **2** writes per bank per cycle. This is achieved by using the parity banks in the coded memory system. When a write is scheduled for a particular bank, the scheduler picks up **two** requests from the head of the queue. It writes the first request to the corresponding data bank. The second write is committed to the parity bank of the respective row. Figure 13 describes the write algorithm as a flowchart.
For example, as shown in figure 14, the write queue is full for all the four banks. The controller picks up the two write requests and schedules them to be written to the respective data bank and the parity bank. The controller also updates the code status table map with the appropriate status. It updates the writes committed to data bank with '01' and writes committed to parity bank with '10'.
A note for the above example about all banks being full at the same time. This is not a general case. The write or read operation is selected for each bank. The controller is designed to schedule read and write for each bank depending on the queue size and availability of the parity of banks.
Another exception to the write logic is when the accesses are linear in nature for the four banks. For example, if the write is to *a1*,*b1*,*c1* and *d1*. In such a case, the controller uses the parity banks to write the parities *a1* + *b1*,*b1* + *c1*,*c1* + *d1*,*d1* + *a1*,*d1* + *b1* and *a1* + *c1*. This ensures that the parity banks remain updated and the memory controller no longer need to update the parity bank. The reason for this exception is to avoid the extra cost involved in recoding the whole row. Though with this, we fall back to one write per bank while saving the cost of recoding.
Figure 14 describes the writes algorithm with an example. The write is selected for a bank when the write queue is full. The scheduler picks up two elements from each bank for writes. It schedules the first element to be written to the data bank and the second one to be written to the memory bank. The memory controller updates the code status map according to table 2. For elements stored to the data bank, i.e., the first element, it updates it to **01** and for elements stored in parity bank to **10**. writes are scheduled to address *a2* and *a9*. The write to *a2* goes to *a2*, however, the write to *a9* goes to the data bank *a9* + *b9*. This helps us do two writes in one cycle. However, the databank of *a9* still contains the stale value. The scheduler updates the *coding status* of each row after committing a write.

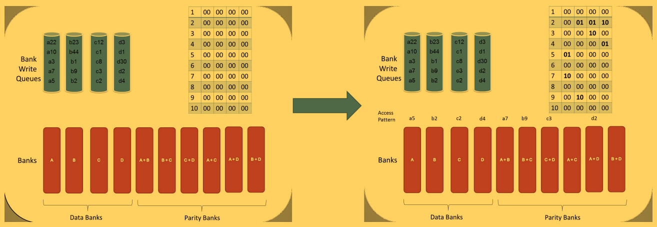


Figure 14: Figure describing write algorithm access pattern

4.4. Dynamic Coding

The *dynamic coding* block in the access scheduler is responsible for maintaining codes for heavily accessed memory sub regions. This block primarily helps in the reduction of code storage for unused memory. This algorithm finds out the current heavily accessed region and indicates to the controller to code that region. With this, we require only a fraction of the whole memory to be coded.
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 15 shows the access pattern of the LTE cores 0 to 5. 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 16 for UMTS. The access pattern for UMTS is more concentrated. Here again, all of the access for a duration of approximately 0.2 ms is in the address range of 0x007a1200 to 0x01036640.
Figure 17 shows a view of memory accesses from the bank's side. It shows the access request pattern for each memory bank. The concentration of accesses to a region is observed across memory banks. This makes us conclude that the memory banks can be parallelly coded for a particular region as shown as green and yellow color in figure 3.
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 in the memory overhead can be used to reduce the complexity of the decoder by using simple coding functions (e.g. xor) and for denser 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

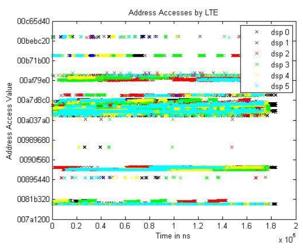


Figure 15: Memory Access by LTE Cores

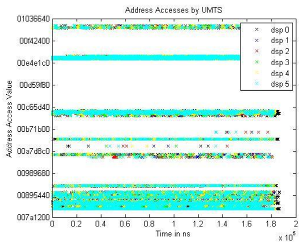


Figure 16: Memory Access by UMTS Cores

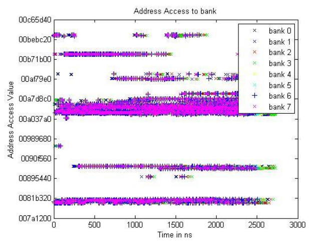


Figure 17: Memory Access per bank for LTE

region is not very frequent. *Dynamic coding* requires the system to divide the memory into sub-regions 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 Least Recently Used (LRU) policy similar to cache.

The block implements a simple logic to determine heavy access to a particular region. It divides the whole memory in to subregions. The memory can be divided dynamically with the provision of the following window parameters $\{$StartAddress,Length$\}$. The controller can have multiple window parameters with the constraint that the total length should be less than the available memory for code storage. This would allow the system designer to have small chunks of distributed memory to be coded. It is important to note here that the codes described are obtained by linear combination of data elements of the same row in various memory bank. So, essentially the window parameter for address signifies the row start.

The dynamic coding controller resets the count of access to the subregions at the switch of the subregion. The new counts determine the next change of the coded subregion.

4.5. 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 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 a_0 and b_0, are likely to be accessed in the near future. It reads $a_0 + b_0$ from the parity bank for future use. Next, access to location a_0 and b_0 are issued to the memory. Now, instead of reading both a_0 and b_0, 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

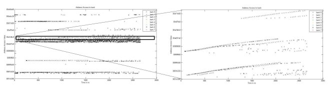


Figure 18

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 18 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. In figure 19, we simulate the prefetching of the code by using a window of length ??. That is, we look ahead to ?? requests 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.

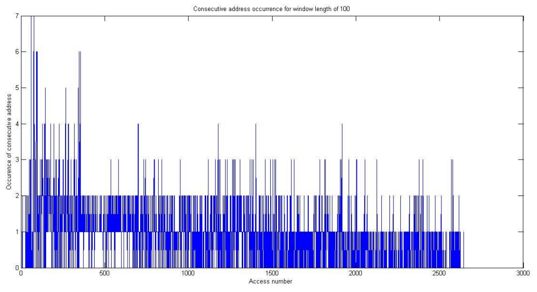


Figure 19

4.6. ReCoding controller

Each write in the coded system destroys the code for that row. The writes are therefore destructive to codes. It then implies a requirement to the memory controller to restore the code back with the updated values. The reCoding unit is responsible for recoding the updated rows at a later time. The reCoding unit comprises of a queue which holds the row number of the updated code. Each row in the queue is associated with the cycle number or a counter which serves as a timing notion of how old is the request in the queue. The controller can be designed to recode a row within X number of cycle after its update.

5. Introduction

This document provides the formatting instructions for submissions to the 20th Annual IEEE International Symposium on High Performance Computer Architecture, 2014. In an effort to respect the efforts of reviewers and in the interest of fairness to all prospective authors, we request that all submissions to HPCA 20 follow the formatting and submission rules detailed below. Submissions that violate these instructions may not be reviewed, at the discretion of the program chair, in order to maintain a review process that is fair to all potential authors. This is a generous format, with plenty of space – there should be no need to tweak it in any significant way.

An example submission (formatted using the HPCA-20 submission format) that contains the submission and formatting guidelines can be downloaded from the location cited below. Complete details can be found on the conference website at *hpcaconf.org*.

Changes to last year's format:

- Papers are limited to 11 pages not including references.
- Each reference must specify all authors (no *et al.*).

Acceptance philosophy

Every effort will be made to judge a paper on its own merits. There will be no target acceptance rate. We expect to accept a wide range of papers with appropriate expectations for evaluation. While papers that build on significant past work with strong evaluations are valuable, papers that open new areas with less rigorous evaluation are even more so.

All questions regarding paper formatting and submission should be directed to the program chair.

6. Preparation Instructions

6.1. Paper Formatting

All submissions should contain a maximum of 11 pages of single-spaced two-column text and figures excluding references. You can use an unlimited number of extra pages for references. If you are using LaTeX [2] to typeset your paper, then we strongly suggest that you use the template available at `http://www.eecg.toronto.edu/~enright/hpca20template.tar.gz` – this document was prepared with that template. This document can also be downloaded at `http://www.eecg.toronto.edu/~enright/hpca20sample.pdf`. If you are using a different software package to typeset your paper, then please adhere to the guidelines given in Table 3.

Please ensure that you include page numbers with your submission. This makes it easier for the reviewers to refer to different parts of your paper when they provide comments.

Also, please ensure that your submission has a banner at the top of the title page, similar to this one, which contains the submission number and the notice of confidentiality. If using the template, just replace XXX in the template with the submission number you receive from the submission website.

6.2. Content

~~**Author List.**~~ All submissions are double blind. Therefore, please do not include any author names in the submission. You must also ensure that the metadata included in the PDF does not give away the authors. If you are improving upon your prior work, refer to your prior work in the third person.

Figures and Tables. Ensure that the figures and tables are legible. Please also ensure that you refer to your figures in the main text. The proceedings will be printed in gray-scale, and

Field	**Value**
Page limit	**11** pages, **not including references**
Paper size	US Letter 8.5in × 11in
Top margin	1in
Bottom margin	1in
Left margin	0.75in
Right margin	0.75in
Separation between columns	0.25in
Body font	10pt
Abstract font	10pt, italicized
Section heading font	12pt, bold
Subsection heading font	10pt, bold
Caption font	9pt, bold
References	8pt, no page limit list all authors' names

Table 3: Formatting guidelines for submission.

many reviewers print the papers in gray-scale. Therefore, if you must use colors for your figures, ensure that the different colors are highly distinguishable in gray-scale. If a figure is not easily understandable in gray-scale, then assume it will not be understood by the reviewers. In many cases, it is better to just prepare your documents without color.

Main Body. Avoid bad page or column breaks in your main text, i.e., last line of a paragraph at the top of a column or first line of a paragraph at the end of a column. If you begin a new section or sub-section near the end of a column, ensure that you have at least 2 lines of body text on the same column.

References. There is no length limit for references. **Each reference must explicitly list all authors of the paper** (no *et al.*). Author of NSF proposals should be familiar with this requirement. Knowing all authors of related work will help find the best reviewers.

7. Submission Instructions

7.1. Paper Authors

Declare all the authors of the paper upfront. Addition/removal of authors once the paper is accepted will have to be approved by the program chair. The paper selection process is carefully run in a way that maximizes fairness by seeking to eliminate all conflicts of interest. Late changes to author lists can invalidate that process.

7.2. Declaring Conflicts of Interest

The authors must register all their conflicts into the paper submission site. Conflicts are needed to resolve assignment of reviewers. Please get the conflicts right. You have several days between the registration of the paper and final submission – there is no need to do the conflicts in a rush at the last second. If a paper is found to have an undeclared conflict that causes a problem, the paper may be rejected.

Please declare a conflict of interest with the following for any author of a paper:

1. PhD advisor
2. Other past or current advisors
3. Current or past students
4. People whom you have collaborated in the last 5 years. This collaboration can consist of a joint research or development project, a joint paper, or when there is direct funding from the potential reviewer (as opposed to company funding) to an author of the paper. Co-participation in professional activities, such as tutorials or studies, is not cause for conflict. When in doubt, the author should check with the PC chair.
5. People with the same current affiliation or who were in the same institution in the last 5 years.
6. Between people whose relationship prevents the reviewer from being objective in his/her assessment. Please be reasonable. For example, just because a reviewer works on similar topics as the paper you are submitting is on, you cannot declare a conflict of interest with them.

All conflicts must be justified. You will have to declare all conflicts with PC members as well as non-PC members with whom you have a conflict of interest. When in doubt, contact the program chair.

7.3. Concurrent Submissions and Resubmissions of Already Published Papers

By submitting a manuscript to HPCA-20, the authors guarantee that the manuscript has not been previously published or accepted for publication in a substantially similar form in any conference or journal. The authors also guarantee that no paper which contains significant overlap with the contributions of the submitted paper is under review to any other conference or journal or workshop, or will be submitted to one of them during the HPCA-20 review period. Violation of any of these conditions will lead to rejection.

Extended versions of papers accepted to IEEE Computer Architecture Letters can be submitted to HPCA-20. If you are in doubt, contact the program chair.

7.4. Submission Site

The submission site is located at `http://hpca20.eecg.toronto.edu/conf`.

References

[1] John L. Hennessy and David A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[2] Leslie Lamport, *LaTeX: A Document Preparation System*, 2nd ed. Reading, Massachusetts: Addison-Wesley, 1994.