Achieving Multi-Port Memory Performance on Single-Port Memory with Coding Techniques

ABSTRACT

Many performance critical systems today must rely on performance enhancements, such as multi-port memories, to keep up with the increasing demand of memory-access capacity. However, the large area footprints and complexity of existing multi-port memory designs limit their applicability in practice. This paper explores a coding theoretic framework to address this problem. In particular, this paper introduces a framework to encode information across multiple single-port memory banks in order to *algorithmically* realize the functionality of multi-port memory.

This paper proposes three code designs which have significantly less storage overhead as compared to existing replication based emulations of multi-port memories. To achieve optimal performance, we also demonstrate a memory controller design that utilizes redundancy across coded memory banks to most efficiently schedule read and write requests sent across multiple cores. Furthermore, guided by real-life traces, the paper explores two potential directions to improve the efficiency of the coding based memory design: 1) *Dynamic coding*, and 2) *Prefetching*. This paper then shows, through simulations, significant performance improvements in critical word read and write latency in the proposed coded-memory design when compared to a traditional uncoded-memory design.

1. INTRODUCTION

Memory accesses are one of the most vital parts of any program. A program is intrinsically made up of loads and stores to the memory. As illustrated in Figure 1, we have witnessed an increasing difference between the performance of processors and memory systems as we move along the Moore's law [1]. Even with the saturation/demise of the Moore's law [2, 3], processing power is expected to grow with the increased reliance on multi-core architectures[4]. Since an end-to-end performance of a program heavily depends on both the performance of processor and memory, slower memory becomes a bottleneck and slows down the whole system. This motivates computer architects to explore various ingenious ways to keep memory access latency as short as possible, including sustained efforts towards enhancing the memory hierarchy[5]. Despite these continuous efforts, long-latency memory accesses do occur when there is a miss in the last level cache (LLC). This triggers an access to shared memory, where the processor has to wait for the shared memory to return the requested information. Such waits create stall in the processors.

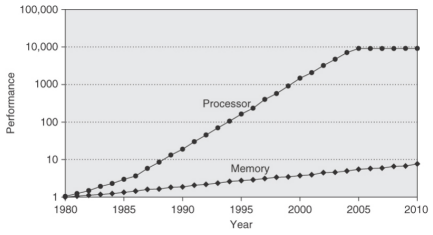


Figure 1: The gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over a 30 year span [6].

In case of multi-core processor architecture, the problem of stalls gets exacerbated even more as access latency to the shared memory increases due to contention among requests issued by different cores. This results in formation of large access request queues waiting to be served by the slow shared memory. Figure 2 illustrates a general multi-core architecture where N processor cores share a memory consisting of M banks. The access requests from each core is sent to the memory controller first, which then arbitrates and in turn issues (schedules) 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 corresponding processor.

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*. In a typical memory system, the bank conflicts lead to the contentious request being served in a sequential manner. Consequently, some of the cores have to wait longer for their request to be fulfilled. As the number of bank conflicts increase, the latency for memory accesses to the bank grows and results into the slow down of the entire

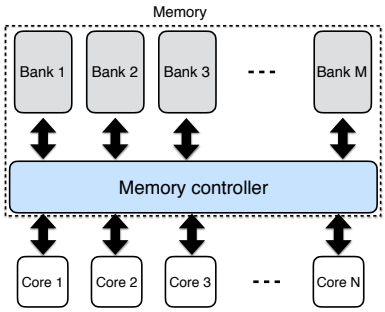


Figure 2: General multi-core architecture with a shared memory.

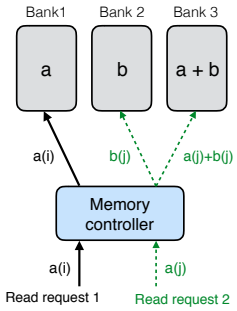


Figure 3: Enabling multiple read accesses to a bank by coding. Given two read requests $\{a(i), a(j)\}$ directed to Bank 1, we can deal with bank conflict in the following manner: 1) First request for $a(i)$ can be directly served by Bank 1 itself, and 2) The read request for $a(j)$ can be served by downloading $b(j)$ and $a(j)+b(j)$ from Bank 2 and Bank 3, respectively. Another case where two read request corresponding to two different banks, e.g., $\{a(i), b(j)\}$, can be simultaneously served from their respective banks without utilizing Bank 3.

system.

In this paper, we aim to address the issue of increased latency due to bank conflicts. The key idea behind our memory design is to distribute the accesses intended for the information stored in a particular bank across multiple banks in the memory. In order to achieve this goal, we rely on coding theoretic techniques to create redundancy across memory banks. That is, we store the information in the memory banks in such a manner that it is possible to recover the information stored on a particular bank by utilizing the information stored in other memory banks. This allows us to simultaneously serve multiple read requests intended for a particular bank, one read request by directly accessing the bank and other requests by querying other banks in the systems. In Figure 3, we illustrate this with the help of an example. The setup in Figure 3 comprises 3 memory banks where the third bank being redundant as its content is function of the content stored on the first two memory banks. Such redundant banks are also referred to as *parity banks*. Assume that the information is arranged in L rows in two first two banks, represented by $[a(1), \ldots, a(L)]$ and $[b(1), \ldots, b(L)]$, respectively. Then the third bank store L rows containing $[a(1)+b(1), \ldots, a(L)+b(L)]$, where $+$ denotes the XOR (modulo 2 addition) operation. As illustrated in Figure 3, this design allows us to simultaneously serve any two read requests (irrespective of their association with first two banks) in a single memory clock cycle. Here, we assume the memory controller to be capable of performing necessary decoding operations, e.g., recovering $a(j)$ from $b(j)$ and $a(j)+b(j)$.

Besides illustration of the avoidance of bank conflicts arising due to read requests, the memory design in Figure 3 helps us recognize various other key issues that are relevant to a hybrid memory: 1) Serving write requests (with or without bank conflicts), 2) Effective utilization of storage space, and 2) Arbitration/scheduling of the accesses. A successful memory system should be able to enable efficient write accesses and maintain the consistency among various read and write requests. Therefore, a memory design should take both read and write requests into account. In the design shown in Figure 3, any write request for an information element, say $a(i)$, should be committed to both Bank 1 and Bank 3. As far as the utilization of the storage space is concerned, the design in Figure 3 uses 3 banks to store the 2 banks' worth of information. This correspond to the information rate[1] of $\frac{2}{3}$. Moreover, as it will become clear later, additional storage space is needed to store the pointers and queues/buffers that further increase the redundancy in any memory system. Finally, in a multi-core setup where multiple cores are sending access requests to hybrid memory systems some designs do not allow for all these requests to be met in a single memory clock cycle. This would require queueing of access requests and subsequent mapping of these requests to the memory banks in a manner such that the overall performance of the entire system is optimized. This objective is referred to as arbitration or scheduling.

Main contributions and organization: In this paper we systematically address all these key issues pertaining to a shared memory system that can simultaneously multiple access requests in a multi-core setup. We present all the necessary background on realization of multi-port memories using single-port memory banks along with an account of relevant prior work in Section 2. We then present the main contributions of the paper which we summarize below.

- We focus on the design of the storage space (array of memory banks) in Section 3. In particular, we employ three specific coding schemes to redundantly store the information in memory banks. These coding schemes, which are based on the literature on distributed storage systems [7, 8, 9, 10], allow us to realize the functionality of multi-port memories from a single port memories while efficiently utilizing the

[1]The information rate is a standard term to measure the redundancy of a coding scheme. The information rate of 1 correspond to the presence of zero redundancy and the most efficient utilization of the storage space.

storage space. Moreover, these coding schemes have low complexity encoding and decoding processes that require only simple XOR operation.

- We present a memory controller architecture for the proposed coding based memory system in Section 4. Among other issues, the memory controller design involves devising scheduling schemes for both read and write requests. In our setup, these scheduling schemes need to take the underlying coding scheme into account in order to utilize the redundancy present in the array of memory banks in the best possible manner. Furthermore, we also address the issue of keeping track of the validity of the information stored in various banks. Note that, due to unserved previous write requests, some of the stored data might have become outdate as far as a particular read request is concerned.

- Focusing on specific application where memory traces might exhibit favorable access patterns, we explore two ways to improve the efficiency of our coding based memory design in Sections 4.6 and 4.7. First, we propose a dynamic coding scheme which is based on continuous detection of heavily accessed regions on memory banks. The dynamic coding scheme only encode these heavily access regions at a particular time instance. As different (uncoded) regions begin receiving more accesses, the dynamic coding scheme updates the content of parity (redundant) memory banks by encoding these regions. Second solutions involves predicting the patterns of memory addresses in different access requests. Based on this prediction, the data from free bank is prefetched to serve subsequent request for information with the help of the prefetched data. This creates the opportunities to serve a large number of access requests in a given memory clock cycle. We note that the design of such prefacing schemes crucially depends on the underlying coding scheme.

- Finally, we conduct a detailed evaluation of the proposed designs of shared memory systems in Section 7. We implement our memory designs using system C and evaluation the overall performance of these designs by regressing their system C implementation through memory traces from real multi-core systems. In addition, we also analyze the performance of our purposed designs with the help of extensive simulation on Ramulator, a DRAM simulator designed by Kim et al. [11].

2. BACKGROUND AND RELATED WORK

2.1 Emulating multi-port memories

The multi-port memories are essential to provide seamless memory accesses in a multi-core setup as these memories can support simultaneous accesses to data elements (which are potentially stored on the same memory bank) by multiple cores. However, designing a true multi-port comes at a large cost. Besides complex circuit implementation, the area requirements for multi-port bit-cells is significantly higher than that for single-port bit-cells [12, 13]. This motivates the exploration of algorithmic and system level designs to emulate multi-port memories using simple and area efficient single-ported memory banks [14, 15, 16, 17, 18].

[ANKIT: This patent by Chappell, Chappell, Ebcioglu and Schuster[19] arguing against both multi-port RAMs and their emulation using single-port RAMs. But the emulation is replication based so we can make a case here for coding based emulation.....]

In the interest of space constraints, we do not present an extensive account of these designs with all possible request patterns. Since these designs are simple enough, we invite the readers to verify that these designs can indeed handle all possible access patterns.

2.1.1 *Supporting only read requests*

Replication-based designs are the most prevalent candidates in this design space. Assuming that a memory design is required to support only read request, say r read request per memory clock cycle, one can simply store r copies of each data element on r different single-port memory banks. In every memory clock cycle, the r read requests can be served in a straightforward manner by mapping all read request to distinct memory banks (cf. Figure 4). This way, the r-replication-based design completely avoids bank conflicts for up to r read request in a memory clock cycle.

REMARK 1. *If we compare the memory design in Figure 4 with that in Figure 3, we notice that both designs can simultaneously serve* 2 *read requests without causing any bank conflicts. Note that the design in Figure 3 consumes smaller storage space as it needs only* 3 *single-port memory banks while the design in Figure 4 requires* 4 *single-port memory banks. However, for the design in Figure 4, the access process involves some computation (Xoring operations). This observation indeed generalizes to the conclusion that the sophisticated coding schemes allow for better storage efficient designs as compare to the replication based design [20]. However, this storage efficiency usually comes at the cost of increased computational requirement. Therefore, it is important to employ those coding schemes that enable storage efficiency with as small computational overhead as possible.*

2.1.2 *Supporting both read and write requests*

[ANKIT: Mainly describing the results from the work of Auerbach, Chen, and Paul[14].] It is evident from the discussion so far that we can indeed emulate the behavior of a multi-port memory on read requests by storing data on single-port memory banks in a redundant manner. The redundancy mechanism can vary from simple replication-based strategy to more sophisticated coding schemes. However, a successful memory design necessarily need to address the issues

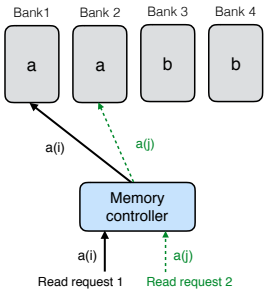


Figure 4: 2-replication based design to support multiple 2 read requests in the same memory clock cycle. The two banks' worth of data $\mathbf{a} = [a(1), \ldots, a(L)]$ and $\mathbf{b} = [b(1), \ldots, b(L)]$, all the data elements are stored on two distinct memory banks. Note that any 2 read requests to distinct memory banks. For example, the figure considers the scenario with 2 read requests for elements $\{a(i), a(j)\}$. Since both $a(i)$ and $a(j)$ are stored on 2 banks, one of those banks can be used to serve each request without causing any bank conflicts. It's straightforward to verify that this memory design avoids bank conflicts for any other set of 2 read requests.

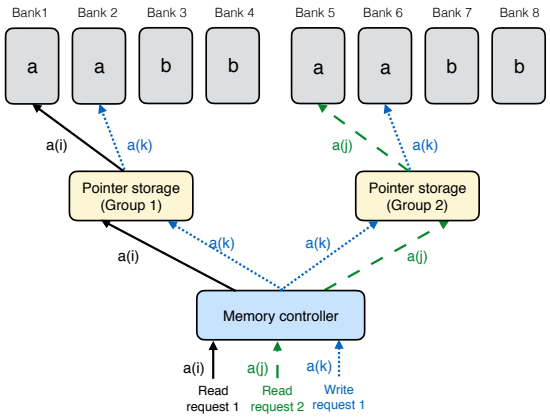


Figure 5: 4-replication based design to support $r = 2$ read requests and $w = 1$ write request in one memory clock cycle. Both collections of information elements $\mathbf{a} = [a(1), \ldots, a(L)]$ and $\mathbf{b} = [b(1), \ldots, b(L)]$ are replicated on $r \cdot (w + 1) = 4$ different single-port memory banks. These banks are then partitioned into $r = 2$ disjoint groups. We utilize each group to serve one read request. In a given memory clock cycle, we focus on the specific access pattern with the read requests for $\{a(i), a(j)\}$ and the write request for $\{a(k)\}$. Assuming that Bank 1 (from Group 1) and Bank 5 (from Group 2) have the updated versions of the data elements $a(i)$ and $a(j)$, respectively, we serve the read requests for $a(i)$ and $a(j)$ from Bank 1 and Bank 5, respectively. As for the write request for the data element $a(k)$, we need to perform this write request in at least one memory bank in each of the two groups. This will enable both groups to continue serving any possible set of $r = 2$ read requests during future accesses. Since we have one memory bank storing $a(k)$ in each of the groups that is not busy serving write request, we write the updated $a(k)$ in these non-busy banks (Bank 2 and Bank 3 in this case). During the writing process, we also need to modify the pointer storage accordingly to keep track of the banks in each group that are storing the most updated values of different data elements.

of (potentially multiple) write requests as well. A challenge that arises in the presence of write requests is that one also need to ensure the consistency across different requests. This requires managing multiple versions of the same information across all the memory banks and making sure that stale information is not supplied in response to a particular read request.

Restricting ourselves to replication-based designs, a multi-port memory that simultaneously supports r read requests and w write requests in a memory cycle can be emulated by using a $r \cdot (w + 1)$ replication scheme, where $r \cdot (w + 1)$ copies of each data element are stores on $r \cdot (w + 1)$ different single-port memory banks. We illustrate this scheme for $r = 2$ and $w = 1$ in Figure 5. According to all of our previous illustrations, we assume that we have two symbols' worth of information $\mathbf{a} = [a(1), \ldots, a(L)]$ and $\mathbf{b} = [b(1), \ldots, b(L)]$. We store $r \cdot (w + 1) = 4$ copies of each data element and partition all the memory banks that store a data element into $r = 2$ disjoint groups with each group containing $(w+1) = 2$ memory banks. In Figure 5, Banks 1 - 4 and Banks 5 - 8 correspond to Group 1 and Group 2, respectively. For the underlying replication-based scheme, we also require additional storage space to keep track of the versions of different copies of the information elements. This space is referred to as the pointer storage. In Figure 5, we illustrate how this design serves a particular set of 2 read requests and 1 write request.

Additional cost to support write requests: Let's look at the additional cost associated with the requirement of being able to support write requests. Recall that the r-replication enables us to serve any set of r read requests in a memory clock cycle. Demanding that we also support w write requests, the required replication factor of the replication-based design jumps to $r \cdot (w + 1)$. This follows as we use r different groups of banks to serve r different read requests. In order to avoid bank conflicts this requires that, for every data element, there should be at least one memory bank that store the most update version of that data element at the beginning of every memory clock cycle. Thus, we should be able to perform w write requests in each of the groups of memory banks. Since we have w write and 1 read operations to perform in every group, we require at least $(w + 1)$ memory banks in each of the r groups. This amounts to the replication factor of $r \cdot (w+1)$. Furthermore, the memory design also requires additional storage space to keep track of the locations of the updated versions of each of the data element. This storage space is referred to as pointed storage in Figure 5. Note that in order to ensure that the pointer storage space is small, we need to continuously update all the replicas of each data element. For data elements that do not have

an entry corresponding to them in the pointer storage, we assume that all of their replicas are storing their current version. This process of synchronization across the replicas of a data element is opportunistically performed on the different banks storing the replicas when these banks are not busy serving access requests from the cores. Therefore, we have two components of the additional cost for the ability to support write requests: 1) Storage space for more replicas and pointer storage and 2) Continuous background maintenance task of synchronizing all the replicas of a data element with its current version.

REMARK 2. *As illustrated above, an r-replication based design to serve r read request can be modified to an $r \cdot (w+1)$-replication based design to support r read and w write requests. If we focus on memory design that supports multiple read requests by using sophisticated coding schemes (e.g., the design in Figure 3), we can modify it to support both read and write requests as well. A generic approach that can be used to support r read requests and w write requests is as follows[2]. Take a coding scheme based memory design that can serve any set of r read requests. Now replication this whole design $(r+w)$ times. These $(r+w)$ copies of the original design are considered as $(r+w)$ different groups. Now, given r read requests we look for minimum number of groups that store the most updated version of the data elements associated with these read requests and serve all the read requests. In the worst case this would require using r different groups. For the w write requests, we commit these w requests to w different groups that are not used to serve read requests. Note that there are at least w such groups. While performing a write request inside a group, we update all the memory banks of the group according to the write request. Similar to the $r \cdot (w+1)$-replication based design, this design also requires additional storage space to store pointers to keep track of the groups storing the most updated version of the data elements. Furthermore, In order to keep this storage space small, we again need to opportunistically synchronize all the banks with the most recent version of the data elements.*

2.2 Storage-efficient emulation of multi-port memories

As described in Section 2.1, by utilizing various ways to introduce redundancy (ranging from simple replication to more sophisticated coding schemes) it's possible to design a memory based on only single-port memory banks that emulate the behavior of a multi-port memory. In a setup where only read requests need to be served (cf. Section 2.1.1) such an emulation is less costly to achieve, both in terms of storage and computational cost. In particular, by careful selection of the underlying coding scheme, it's possible to serve multiple read requests by incurring both small storage and computational overhead (cf. Remark 1).

[2]We note that depending on the specific coding scheme, one can present a more storage-efficient design. Here, we present a universal scheme that works for any coding scheme.

However, the emulation become much more costlier in the scenario when write requests also needs to be performed (cf. Section 2.1.2). Besides the increment in the number of single-port memory banks, the ability to serve write requests also requires the installation of pointer storage to keep track of the various versions of the data elements present in the memory banks. As highlighted in Section 2.1.2, it's necessary to continuously synchronize all the memory banks storing a particular data element in order to keep the pointer storage space small. Furthermore, the presence of varying version in the banks also complicates the process of arbitration, i.e., mapping access requests to memory banks, as read requests need to be served by the bank storing the current version of the data element. Since most of the programs in a multi-core would involve significant amount of write requests, any design to emulate multi-port memory using single-port memory needs to take these overheads into account.

We believe that various tasks that arise in the presence of write requests and contribute to computational overhead of the memory design, including synchronization among memory banks and complicated arbitration, can be better managed at the algorithmic level. Note that these tasks are performed at memory controller. It's possible to reduce the effect of these tasks on the overall performance of memory system by relying on the increasing available computational resources while designing the memory controller. On the other hand, we believe that the storage overhead is a more fundamental issue that needs to be addressed for the emulation of the multi-port memories to be viable and appealing. In particular, the large replication factor in a naive design (cf. Remark 2) limits the applicability of the obtained memory in practice due to large storage overhead and the associated large area requirement resulting from this.

In order to reduce the storage overhead, we avoid the two step (naive) memory design process highlighted in Remark 2: 1) First, employ a coding scheme that can serve multiple read requests, and 2) Then replicated the obtained memory bank arrangement multiple times in order to support write requests as well. As opposed to this two step process, we simply encode the data elements using specific coding schemes which create parity banks by encoding over a multiple data banks and have reasonably high rate. We select the underlying coding scheme to support multiple read requests in the worst case. Instead of replicating the obtained design, we exploit the fact we do not always encounter worst case pattern for read request and the obtained design can potentially serve access patterns with much larger number of read requests. In other words, for many access patterns with a given number of read requests, there are many memory banks that remain unused. These unused banks are generally available to perform (part of) pending write requests. Therefore, if one aims at performing arbitration among access requests arising over a slightly longer duration as opposed to focusing on requests arriving at each memory clock cycle, all the requests can

be served without good latency. In this way, instead of designing various components of the memory system, e.g., bank array and memory controller, independently, taking the holistic view of the entire memory system allows us to not commit unnecessary storage space in terms of large number of bank which only provide small amount of utility in terms of performance of the system. We recognize that this approach leads to increased complexity at the memory controller. *However, we show that the increment in the complexity can be kept within the acceptable level while insuring storage-efficient emulation of multi-port memories with the help of better algorithmic design.*

2.3 Related work

Coding theory is one of the well studied field which deals with mitigating the adversarial effects of the underlying medium in an information processing system [20, 21]. In particular, the developments in the field have enabled both reliable communication across noisy channel and storage over fault-prone storage units in resource efficient manners. Recently, we have witnessed intensive efforts towards the applications of coding theoretic ideas to design large scale distributed storage systems (see e.g., [22, 23, 24]). In this domain of coding for distributed storage systems, the issue of access efficiency has also received attention, especially the ability to support multiple simultaneous read accesses with small storage overhead [9, 10, 25, 26] and references therein. In this paper, we rely on the coding techniques developed under in this domain to realize emulation of multi-port memories using single-port memory banks. However, we note that the existing work on batch codes [9] only focuses on the read requests. On the other hand, the successful emulation of multi-port memory also requires handling write requests in an efficient manner. Furthermore, the design presented in this paper also needs to address the entire memory system which also involves memory controller design as opposed to just focusing on the storage array.

Here, we note that the issue of designing coding schemes that have low update complexity, i.e., that can be modified with low overhead as the information gets updated, have also received some attention in the literature (see e.g., [27, 28]). However, this treatment is extensive enough to address the update issues that arise in the context of our memory systems, where write requests may be very frequent and a large portion of the bank array needs to get updated. Again, the key issue that distinguish our work from the majority of the literature on coding for distributed storage is that we need to take the interplay among read and write requests and its affect on the overall performance (latency) into account. Furthermore, we are not allowed to encode across a very large number of storage units (memory banks in our case), which is very much feasible in today's large scale cloud storage systems.

In this paper, we also explore the idea of proactively prefetching the information from memory banks to improve the access efficiency of our memory design. The idea of prefetching in realizing fast data transfer between processors and memory has been previously explored in the literature [29, 30, 31, 32] and references therein. However, our work addresses the issue of data prefetching in the context of coded memory system which is not addressed earlier in the literature. This does share some similarity with the recent line of work on coded caching [33] which aims to reduce the data downloaded from servers in a communication network by utilizing the cache available at the end users. Here, we would like to point out that there are many key differences in the our setup with coded memory banks with that considered in [33]. In particular, our setup has data stored in an encoded form stored across memory banks and caching is enabled by the memory controller, which is a centralized units. Whereas the setup of coded caching have a centralized storage system (referred to as a server) and cache units which store encoded information are distributed across users.

The work which is closest to our solution for emulating a multi-port memory is by Iyer and Chuang [17, 18], where they also employ XORing based coding schemes to redundantly store information in an array of single-port memory banks. However, we note that our work significantly differers from [17, 18] as we specifically rely on different coding schemes arising under the framework of batch codes [9]. Additionally, due to the employment of distinct coding techniques, the design of memory controller in our work also differs from that in [17, 18].

[ANKIT: Also cite the work by Rivest et al. [16] and Endo, Matsumura and Yamada [15].]

3. CODES TO IMPROVE ACCESSES

As a key idea behind the memory designs proposed in this paper, we plan to utilize coding schemes to introduce the redundancy into the storage space comprising an array of single-port memory banks in order to support multiple accesses to the array. For this approach to be successful, it's important to work with the right coding schemes which are designed by keeping the access-efficiency as the central criterion. In this section, we describe three such coding schemes that we employ for the task of emulation of multi-port memories using single-port memory banks. Before presenting these coding schemes, we first introduce some basic concepts and terminology that we use throughout the rest of this paper.

3.1 Coding for memory banks

A coding scheme is defined by its encoding process which maps a given collection of information elements to a larger and redundant collection of elements known as codeword. In the context of memory banks, we begin with *data banks* which refer to the memory bank storing the original information. The encoding process of the underlying coding scheme then utilize the content of these data banks to generate elements to be stored in the additional memory banks such the overall array of

memory banks stores the information in a redundant manner. The additional memory banks are termed as *parity banks*. The encoding operations used to generate parity elements in this paper can be classified into two categories.

1. **Inter-bank encoding:** A parity element depends on at most one element for each of the data banks.
2. **Intra-bank encoding:** A parity element is generated using multiple elements from a data bank.

Furthermore, we restrict ourselves to linear coding schemes which perform only linear operations (over a finite field) during the encoding process. In fact, in order to keep the computation complexity small, we only consider coding schemes over the binary field. This implies that the linear operations are nothing but XOR operations. The following example further clarifies these concepts along with some necessary notation with help of a coding scheme that employs only inter-bank encoding.

EXAMPLE 1. *Let's consider a setup with two data banks* $\mathbf{a}$ *and* $\mathbf{b}$. *We assume that each of banks store* LW *binary data elements*[3] *which are arranges in an* $L \times W$ *array. In particular, for* $i \in [L] \triangleq \{1, \ldots, L\}$, $a(i)$ *and* $b(i)$ *denote the* i*-th row of the bank* a *and bank* $\mathbf{b}$, *respectively. Moreover, for* $i \in [L]$ *and* $j \in [W] \triangleq \{1, \ldots, W\}$, *we use* $a_{i,j}$ *and* $b_{i,j}$ *to denote the* j*-th element in the rows* $a(i)$ *and* $b(i)$, *respectively. Therefore, for* $i \in [L]$, *we have*

$$a(i) = \big(a_{i,1}, a_{i,2}, \ldots, a_{i,W}\big) \in \{0,1\}^W$$
$$b(i) = \big(b_{i,1}, b_{i,2}, \ldots, b_{i,W}\big) \in \{0,1\}^W.$$

Now, consider a linear coding scheme that produces a parity bank $\mathbf{p}$ *with* $L'W$ *bits arranged in an* $L' \times W$ *array such that for* $i \in [L'] \triangleq \{1, \ldots, L'\}$,

$$\begin{aligned} p(i) &= \big(p_{i,1}, \ldots, p_{i,W}\big) \\ &= a(i) + b(i) \\ &\triangleq \big(a_{i,1} \oplus b_{i,1}, a_{i,1} \oplus b_{i,1}, \ldots, a_{i,1} \oplus b_{i,1}\big). \end{aligned} \tag{1}$$

Figure 6a illustrates this coding scheme. Since the parity bank is based on those rows of the data banks that are indexed by the set $[L'] \subseteq [L]$, *we also use the following concise notation to represent the encoding of the parity bank.*

$$\mathbf{p} = \mathbf{a}([L']) + \mathbf{b}([L']).$$

In general, we can use any subset $\mathcal{S} = \{i_1, i_2, \ldots, i_{L'}\} \subseteq [L]$ *comprising* L' *rows of data banks to generate the parity bank* $\mathbf{p}$. *In this case, we have*

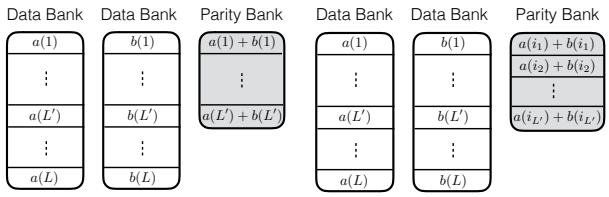
$$\mathbf{p} = \mathbf{a}(\mathcal{S}) + \mathbf{b}(\mathcal{S}),$$

or

$$p(l) = a(i_l) + b(i_l) \text{ for } l \in [L'].$$

Figure 6b illustrates the case with a generic set $\mathcal{S}$.

[3]It is possible to work with data elements over larger alphabets/finite fields. However, assuming data elements to be binary suffices for us as we only work with coding schemes defined over binary field in this paper.



(a) Parity. (b) Parity.

Figure 6: Design.

REMARK 3. *Note that we allow for the data banks and parity banks to have different sizes, i.e.,* $L \neq L'$. *This freedom in memory design can be utilized to reduce the storage overhead of parity banks based on the underlying application. The case when the size of a parity bank is smaller than a data bank, i.e.,* $L' < L$, *we say that the parity bank is a* shallow bank. *We note that it's perfectly reasonable to have the provisions for shallow banks, especially in proprietary designs of integrated memories in a system on a chip (SoC).*

3.1.1 Degraded reads and their locality

Now let's focus on the application of redundant (encoded) data stored on an array of banks to improves the accesses to the original data elements by avoiding bank conflicts. Let's consider the coding scheme illustrated in Figure 6a, where the parity bank $\mathbf{p} = \mathbf{a}([L']) + \mathbf{b}([L'])$. Now, given two read requests $\{a(1), a(L)\}$, which would have led to a bank conflict (at the data bank $\mathbf{a}$) without the parity banks, can now be simultaneously served without any bank conflicts: 1) Serve the request $a(L)$ from the data bank $\mathbf{a}$, and 2) Download $b(1)$ and $p(1) = a(1)+b(1)$ from the data bank $\mathbf{b}$ and the parity bank to serve $a(1)$ by reconstructing $a(1) = b(1) + p(1)$. A read request which is served with the help of parity banks (the request for $a(1)$ in this case) is termed as *degraded read*. Each degraded read has a parameter *locality* associated with it which corresponds to the total number of banks that need to be accessed to serve the degraded read. In this aforementioned case, the locality of the degraded read for $a(1)$ is 2 as we access two banks, the data bank $\mathbf{b}$ and the parity bank $\mathbf{p}$, to perform the degraded read.

In order to further illustrate the notion of locality, let's consider a setup where we generate a parity bank $\mathbf{p}$ by combining three data banks $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ as $\mathbf{p} = \mathbf{a} + \mathbf{b} + \mathbf{c}$. Now, a degraded read for $a(1)$ using the parity bank as

$$a(1) = b(1)+c(1)+p(1) = b(1)+c(1)+\big(a(1)+b(1)+c(1)\big)$$

has locality 3 as the degraded read is served using three memory banks.

3.2 Codes to emulate multi-port memory

We are now in position to describe the code designs employed in this work to carry out the emulation of multi-port memories. Among a large set of possible coding schemes, we focus on three specific coding schemes for this task. We believe that these three coding schemes

Design	Max reads (per bank)	Locality	Rate	Storage overhead	Logical complexity
I	4	2	2/5	1.5α	Low
II	5	2	2/5	2.5α	Medium
III	4	3	1/2	α	Medium

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

strike a good balance among various quantities of interests, including storage overhead, number of simultaneous read requests supported by the array of banks, and the locality associated with various degraded reads. Furthermore, these coding schemes respect the practical constraint of encoding across a small number of data banks. In particular, we focus on the setup with 8 memory banks. This is in contrast with the application of coding in communications, where encoding is performed over a much larger number of information symbols.

In the rest of this section, we present the three code designs and discuss the number of simultaneous read requests supported by these designs both in the worst and the best case. We also summarize all the relevant parameters associated with these designs in Table 1 which highlights the trade-offs among these parameters.

3.2.1 Code design I

This code design is motivated from the concept of batch codes [9] which enables parallel access to the content stored in a large scale distributed storage system. This design employs only inter-bank encoding across rows of data banks. The coding scheme is illustrated in Figure 7. The code design involves 8 data banks $\{\mathbf{a}, \mathbf{b}, \ldots, \mathbf{h}\}$ (each of size L) and 12 shallow banks (each of size $L' = \alpha L$ for $0 < \alpha \leq 1$). We partition the 8 data banks into two disjoint groups consisting of 4 banks each. The underlying coding scheme produces (shallow) parity banks by separately encoding data banks from the two groups. Thus, as shown in Figure 7, the entire array of banks, including data banks and parity banks, can be viewed as partitioned into two disjoint code region. For a specific choice of α, the storage overhead of this design is $12\alpha L$ which amounts to the rate of the coding scheme being

$$\frac{8L}{8L + 12\alpha L} = \frac{2}{2 + 3\alpha}.$$

REMARK 4. *Note that the size of shallow banks is a design choice which is controlled by the parameter $0 < \alpha \leq 1$. A small value of α corresponds to small storage overhead. However, $\alpha < 1$ ensures multiple access for only a subset of data elements (belonging to the encoded rows) from the data banks. The identity of the $L' = \alpha L$ rows from data banks that are encoded using the shallow parity banks depend on nature of access request being served by the memory system. In particular, in Section 4.6 we discuss a dynamic coding approach that dynamically change the encoded rows based on the detection of the regions of memory banks that receive a large number of access requests. For applications, where such sustained concentration of access patterns is not present, one can employ design with $\alpha = 1$, i.e., the parity banks have the same size as the data banks and all the elements of the data banks are stored in the encoded form in the array of banks.*

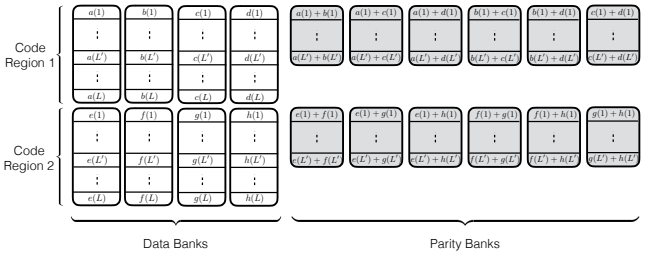


Figure 7: Code Design I

We now analyze the number of simultaneous read requests that can be supported by this code design.

Best case analysis: This code design achieves maximum performance when sequential accesses to the coded regions are issued. During the best case access, we can achieve up to 10 parallel accesses to a particular coded region in one cycle. Consider the scenario when we receive accesses to the following 10 rows.

$$\big\{a(1), b(1), c(1), d(1), a(2), b(2), c(2), d(2), \ldots \\ a(3), b(3), c(3), d(3)\big\}$$

Note that we can serve the read requests for the rows $\{a(1), b(1), c(1), d(1)\}$ using the data bank $\mathbf{a}$ and the three parity banks storing $\{a(1)+b(1), b(1)+c(1), c(1)+d(1)\}$. The requests for $\{a(2), c(2), d(2)\}$ can be served by downloading $b(2)$ from the data bank $\mathbf{b}$ and $\{b(2)+c(2), b(2)+d(2), a(2)+c(2)\}$ from the respective parity banks storing these. Lastly, in the same memory clock cycle, we can serve the requests for $\{c(3), d(3)\}$ using the data banks $\mathbf{c}$ and $\mathbf{d}$.

Worst case analysis: The code design under consideration (cf. Figure 7) falls off to 4 access in a single memory clock cycle when there are non-sequential and non-consecutive access to the memory banks. For example, when we receive read requests for the rows $\{a(1), b(8), c(9), d(15)\}$. Since parity elements combining the data elements from these requested rows are not present in the parity banks, the underlying coding scheme does not provide any benefits. However, as discussed in Section 4.7, we can use the prefetching mechanism to look ahead in the queue and prefetch codes from parity banks for the subsequent access in order to exploit the otherwise unused parity banks.

3.2.2 Code Design II

Figure 8 illustrates the second code design explored in this paper. Again, the 8 data banks $\{\mathbf{a}, \mathbf{b}, \ldots, \mathbf{h}\}$ are partitioned into two groups containing 4 data banks each. These two groups are then associated with two

code regions. This code design generates 9 parity banks for 8 data banks, where only the content stored on the data banks from the same region is combined to generate these parity elements. As evident from Figure 8, this design employs both inter-bank and intra-bank encoding in order to generate the content to be stored on the parity banks. For example, the parity element $a(1)+a(L'+1)$ results from an intra-bank encoding operation. This code design differs from the previous code design (cf. Figure 7) in terms of the size and arrangement parity banks. Even though $L' = \alpha L$ rows from each data bank are stored in a coded manner by generating parity elements, the parity banks are assumed to be storing $2\alpha L > L'$ rows. Furthermore, the parity-banks generated using intra-bank encoding from one code region are stored on the parity banks of another code region.

For a specific choice of α, the storage overhead of this design is $20\alpha L$ which amounts to the rate of the coding scheme being

$$\frac{8L}{8L + 20\alpha L} = \frac{2}{2 + 5\alpha}.$$

Note that this code design can support 5 read accesses per data bank in a single memory clock cycle as opposed to 4 read requests supported by the code design from Section 3.2.1. However, this is made possible at the cost of extra storage overhead. Next, we discuss the performance of this code design in terms of the number of simultaneous read request that can be served in the best and worst case.

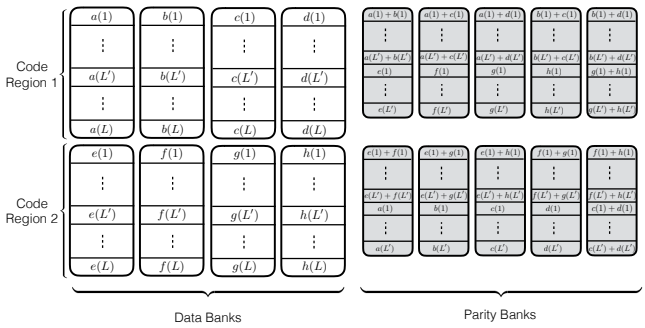


Figure 8: Code Design II

Best case analysis: This code design achieves the best access performance when sequential accesses to the data banks are issued. In particular, this design can support up to 9 read requests in a single memory clock cycle. Consider the scenario where we receive read requests for the following rows of the data banks.

$$\{a(1), b(1), c(1), d(1), a(2), b(2), c(2), d(2), a(3), b(3), c(3)\}$$

Here, we can serve the requests for the rows $\{a(1), b(1), c(1), d(1)\}$ using the data bank **a** with the parity banks storing the parity elements $\{a(1) + b(1), b(1) + c(1), c(1) + d(1)\}$. Similarly, we can serve the requests for the rows $\{a(2), b(2), d(2)\}$ using the data bank **b** with the parity banks storing the parity elements $\{a(2) + d(2), b(2) + d(2)\}$. Lastly, the request for the rows $c(2)$ and $d(3)$ is served using the data banks **c** and **d**.

Worst case analysis: The code scheme can enable 5 simultaneous accesses in a single memory clock cycle in the worst case. These are non-sequential and non-consecutive accesses to the memory banks. For example, when the access pattern corresponds to the rows $\{a(1), a(6), a(9), a(15), a(20)\}$, we can simultaneously serve these 5 read requests with the help of our coded memory. In order to better utilize the unused banks in this case, we can use the prefetching mechanisms (cf. Section 4.7) to look ahead in the queue and proactively download elements from the unused banks for future accesses.

3.2.3 Code Design III

The two code designs discussed so far have locality 2, i.e., we need to access 2 memory banks to serve a read request with the help of parity symbols. The next design that we consider has locality 3. This code design works with 9 data bank $\{\mathbf{a}, \mathbf{b}, \ldots, \mathbf{h}, \mathbf{z}\}$ and generates 9 (shallow) parity banks. Figure 9 describes this design. As on can verify that this design relies on only inter-bank encoding to generate parity symbols. The storage overhead of this design is $9\alpha L$ which corresponds to the rate of $\frac{1}{1+\alpha}$. We note that this design possesses higher logical complexity because of increased locality.

This design helps us support 4 simultaneous read access per bank per memory clock cycle. As an example, the requests for the rows $\{a(1), a(2), a(3), a(4)\}$ can be satisfied as follows. The request for $a(1)$ can be served bu the data bank **a**. The request for the row $\mathbf{a}(2)$ can be supported by using the data banks **b** and **c** along with the parity bank storing $a(2) + b(2) + c(2)$. Similarly, the data banks **d** and **g** along with parity bank storing $a(3) + d(3) + g(3)$ allows us to access $a(3)$. Lastly, we can serve the request for the row $a(4)$ by using the data banks **e** and **z** with the parity element $a(4)+e(4)+z(4)$.

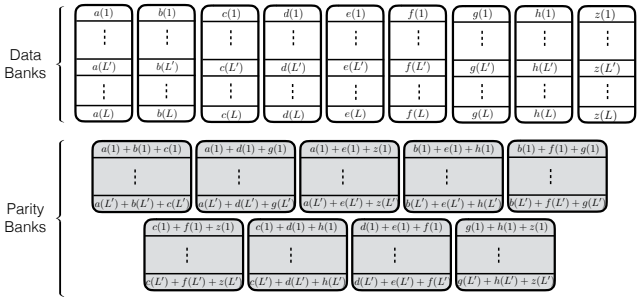


Figure 9: Code Design III

Remark 5. *Note that the coding scheme in Figure 9 describes a system with* 9 *data banks. However, we has set out to construct a memory system with* 8 *data banks. It is straightforward to modify this code design to work with* 8 *data banks* $\{\mathbf{a}, \mathbf{b}, \ldots, \mathbf{h}\}$*. In particular, we can assume the data elements store in the data bank to be*

all 0s and discard this bank. This results into the code design described in Figure 10. Since we assume the elements of the discarded data bank $\mathbf{z}$ *to be all 0s, the resulting 9 parity banks do not have any contribution from this bank. Note that the modified memory system has varying locality for degraded reads. Some degraded reads require accessing 3 banks (e.g., accessing* $a(1)$ *using* $b(1)$, $c(1)$ *and* $a(1)+b(1)+c(1)$*) while other degraded reads have locality 3 (e.g., accessing* $g(1)$ *using* $h(1)$ *and* $g(1)+h(1)$*). Note that the storage overhead of the modified design is the same as that of the design described in Figure 9.*

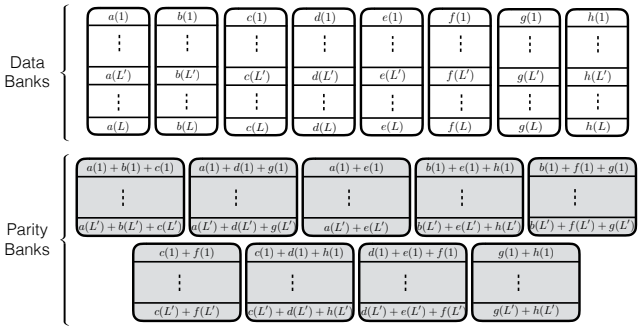


Figure 10: Code Design III with 8 data banks

[ANKIT: What about the best case vs worst case analysis for this design??]

4. MEMORY CONTROLLER DESIGN

The previous section describes various coding schemes to design the storage space of the proposed memory systems. In this section, we discuss the architecture of memory controller which constitutes the second main component of the memory system. Given that our memory system redundantly stores the data in memory banks, the design of the memory controller involves many novel ideas that enable it to exploit this redundancy in order to provide improved parallel access to the memory banks. This section presents these key architectural ideas behind the memory controller design along with various implementation related deails.

4.1 Main units at memory controller

A general memory controller consists of three main levels of processing (cf. Figure 11). The first level deals with handling the access requests from cores which is performed by *core arbiter*. The second level of processing is conducted by *bank arbiter* which is responsible for tracking requests for different data banks. The third main unit *access scheduler* deals with the final level of processing by scheduling the most efficient access patters for each memory clock cycle from the set of accesses assigned by the bank aribiter. Next, we discuss all of these three units and their functions in a greater detail.

- **Core arbiter:** 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: 1) First, it arranges the request in order of QoS with high QoS at high priority; and 2) Second, for request having same priority, it arranges them using round robin.

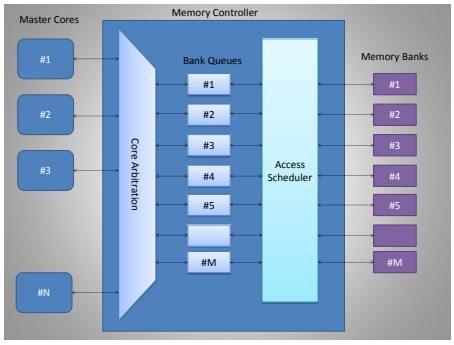


Figure 11: Architecture of Memory Controller

- **Bank arbiter:** This unit contains read and write queues for each data 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 160 requests to be stored in these queues.

- **Access scheduler:** The access scheduler is responsible for scheduling access to various memory banks in the storage space at every clock cycle. This logic looks at the read and write queues for each data 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 memory systems (discussed in 4.3) 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.

We note that the first two components of the memory controller, i.e., core arbiter and bank arbiter, in the setting with coded memory banks should not differ much from those in the traditional setup with uncoded storage space. It's the third component, access scheduler, that directly interacts with the array of memory banks and it should take the underlying coding scheme into account while scheduling both read and write requests.

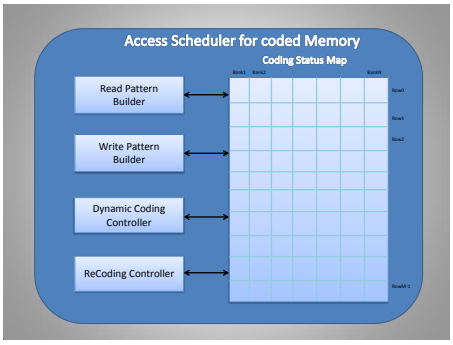


Figure 12: Access scheduler for coded memory

In the rest of this section, we discuss various parts of the access scheduler in detail.

4.2 Code Status Table

[ANKIT: Isn't this going to be too much storage? May be we should just keep a fixed sized pointer storage keeping the addresses of only those elements that are not up to date]
[MATT: The overhead from the map is $\frac{1}{4} bytes \times < num\ data\ banks > \times < num\ rows\ in\ map >$ bytes. For 8 data banks, we see 2 bytes needed per row. I believe a single row of HBM memory contains 1024, so the increase in storage is relatively small.] Due to write requests, the information stored in the memory banks need to be constantly updated. However, the updating an information element requires more work in the setup with coded memory banks as all the parity elements that depend on the information element also need be updated. In order to keep track of the freshness of the data stored in the various memory banks, we assign a storage space, namely *code status table*, in the access scheduler. As depicted in Figure 12, this block's rows correspond to rows of the memory banks. The different columns of the status table are associated to different memory banks in our memory design. Each cell in the status table stores 2 bits, which are mapped to the status of the code for the corresponding row of the memory banks as per Table 2. We assume that the elements of the code status table are accessible at a very fast rate.

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

4.3 Read pattern builder

[ANKIT: After we are done building access pattern...we also need to take the status of the parity and data symbols into account? NO?]
[MATT: The access pattern builder takes the status of the symbols into account] The primary

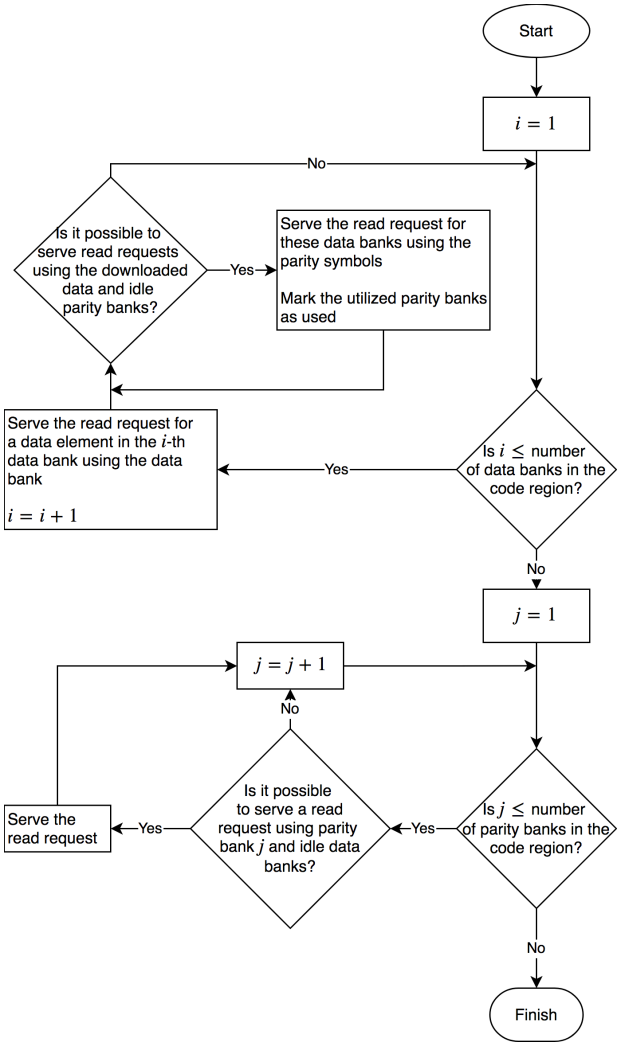


Figure 13: Description of the algorithm to build a read request pattern to be served in a given memory cycle.

objective of the access scheduler it to maximize the number of read requests served in a given memory cycle. This is achieved by the block *read pattern builder*. The main idea here is to exploit the redundancy in the parity banks in order to serve as many simultaneous read requests as possible. ~~However, reading elements from a parity bank is useful only if the corresponding accesses to data banks are also performed. For example, the parity element $a(1) + b(1)$ helps serve the read request for $a(1)$ only if we also read $b(1)$ for the associated data bank. Therefore, whenever the read pattern builder decides to serve a read request by accessing a data banks, it explores for the opportunities to serve additional read requests by downloading parity elements from the (unused) parity banks in the same memory clock cycle.~~

[MATT: We should include an example where using idle data banks is advantageous] We describe the algorithm employed by the read request pattern in Figure 13. We illustrate this algorithm with the help of an example, which is shown in Figure 14. The con-

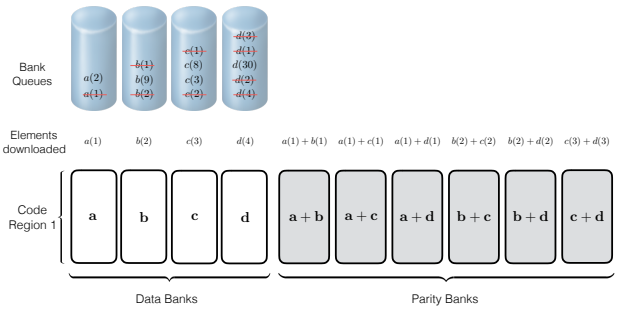


Figure 14: Illustration of the algorithm to build a read request pattern to be served in a given memory cycle. All the read requests associated with the strikethrough elements are scheduled to be served in a given memory cycle. The figure also shows the elements downloaded from all the memory banks in order to serve these read requests.

troller first marks *a(1)* to be read from the data bank **a**. It then iterates through the read requests fro the data banks **b**, **c**, and **d** to find out if any of those requests can be served by combining $a(1)$ with parity elements from the unused parity banks. As it so happens, the read requests for $b(1), c(1)$ and $d(1)$ can be served using $a(1)$ and the parity banks. The controller marks the corresponding parity banks to be used (or busy). Then it moves to the data bank **b** to serves the read request for $b(2)$ using this bank. It then searches if it can serve any request for the elements in the data banks **a**, **c**, and **d** by combining the downloaded data element $b(2)$ with the parity elements from the unused parity banks. As a result of this search, it decides to serve the read requests for $c(2)$ and $d(2)$ with the help of the (unused) parity banks storing the parity elements $b(2) + c(2)$ and $b(2) + d(2)$. Similarly, the read pattern builder goes through bank banks **c** and **d** to form the complete access pattern. As it can be observed from Figure 14, the controller serves $4, 3, 2$, and 1 read requests for the data banks **d**, **c**, **b**, and **a**, respectively. 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 always guarantee that there would be 1 read requests for the first bank and 4 access for the last bank. This is dependent on the actual read requests present in the queues. **[ANKIT: Also, we need to make sure that we are not jumping any write request while going through the queues to build a pattern, right? Or are we only doing FIFO processing of the read requests?] [MATT: The Core arbiter should be designed such that jumping write requests is impossible]**

The example considered in Figure 14 highlights three different aspect of the read algorithm (cf. Figure 13). First, the access scheduler iteratively tries to schedule the maximum number of read requests in a given memory cycle. The accesses from the other banks are searched. The controller maintains a bitmap of availability of parity banks. It marks a parity bank busy/used when it assigns a read request to be served with the help of this bank. Second, the access scheduler 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 access scheduler goes over the data bank in order of their queue sizes. This enables it to maximize the chance of serving 4 read requests for the last data bank in the considered ordering.

REMARK 6. *Here we note that the aforementioned approach of maximizing the number of read request being served 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, in order to prevent the out-of-order execution of the access requests arising from the same core, the logic needs to take care of in-order execution of requests from each cores.* ***[ANKIT: How does logic take care of this...Can you write 1-2 line on that? Would it affect the performance of the read algorithm?]***
[MATT: A solution to this problem is to only allow requests to enter the bank arbiter if they can be served immediately without risk of out-of-order execution.]

4.4 Write pattern builder

[MATT: Do we want a flowchart in here to parallel the one in section 4.3?] Next, we address the issue of scheduling write requests during the operation of the proposed memory system. Recall that the write requests are first filled in the write queues for each bank by the bank arbiter. The unit that resides in the access scheduler and commit these write requests to the memory banks is referred to as *write pattern builder*. The write pattern builder waits till a write queue gets full to schedule the write requests to be served. This approach is adopted in order to ensure that the read requests get priority over the write requests.

In particular, we aim to commit 2 write request per data bank in the given memory cycle. When the write queue for a particular bank gets full, the write pattern builder picks 2 write request from the head of the queue. It write the first write request to the corresponding data bank. For the other write request to be scheduled, the builder commits this request to one of the parity banks that depend on the data bank associated with the write request. In particular, the builder replaces the parity element that depends on the data element of the write request with the data element itself. Note that this destroys the parity elements and the coding scheme (in the corresponding row is deemed outdated). This requires us to update the code status table accordingly. We handle the outdated coding scheme using another block of the access scheduler, namely *ReCoding unit* (cf. Section 4.5).

We illustrate the aforementioned functioning of the write pattern builder with the help of an example shown in Figure 15. In this example, the write queue is full

for all the four banks. The controller picks up 2 write requests from each of these queues and schedules them to be written to the respective data bank and one 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. The write build pattern will only serve a write using parity banks if the request's row exists in the parity bank.[**ANKIT: ~~What do we don here in case of dynamic coding with $\alpha < 1$. Here, we may not have parity elements for some of the rows in the data bank. Which rows of the parity banks do those write requests get committed?~~**]

REMARK 7. *In the example described in Figure 15, the queues for all data banks become full at the same time. In general, this is not the case and the write queues for some of the data banks become full before it does so for the other data banks. In this case, the write requests for only some of the banks get scheduled in a given memory cycle. We note that the controller is designed to schedule both read and write for each bank depending on the queue size and availability of the parity of banks. Therefore, if a data bank is not utilized to serve a write request in a memory cycle, it can be potentially utilized to serve a read request in that cycle.* ***[MATT: The current simulations only schedule reads requests OR write requests in a single memory cycle. There are issues in Ramulator when scheduling reads and writes simultaneously...]***

REMARK 8. *An exception to the standard logic of the write pattern builder is invoked when the accesses are linear in nature for all* 4 *data banks. For example, if we have write requests for the elements* $a(1), b(1), c(1)$, *and* $d(1)$. *In such a case, the controller directly updates all the parity banks by writing the updated versions of the parity elements* $a(1)+b(1), a(1)+c(1), a(1)+d(1), b(1)+c(1), b(1)+d(1)$, *and* $c(1)+d(1)$. *This ensures that the parity banks remain updated and the memory controller no longer need to update the parity bank using ReCoding unit (cf. Section 4.5). The reason for invoking this exception is to avoid the extra cost involved in recoding the whole row. Though this saving in the cost of recoding come at a price as we fall back to* 1 *write request per data bank in the underlying memory cycle.*

4.5 ReCoding unit

As described in Section 4.4, committing write requests to the memory banks destroys the coding scheme for the corresponding row. Note that this information is available in the code status table. In order to maintain the original (coded) storage so that it can be utilized to reduce the latency experienced by the read requests, the memory controller needs to restore the coding scheme with the updated values of data elements. The ReCoding unit is responsible for recoding the updated rows at a later time. This unit comprises a queue which holds the row number of the updated code. Each row in the

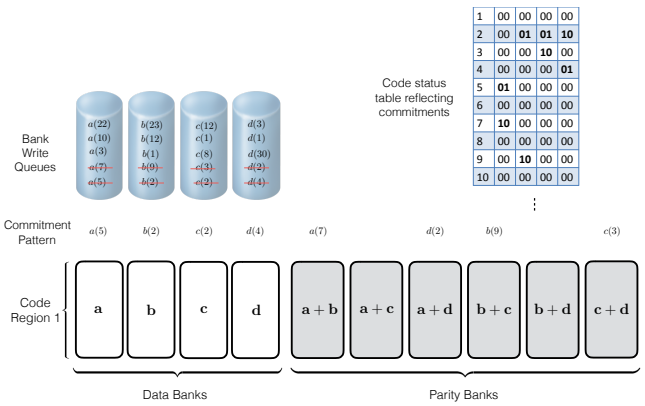


Figure 15: Figure describing write algorithm access pattern

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 XYZZZZZ number of cycle after its update.

The ReCoding unit restores the coding scheme by utilizing banks left idle by the read and write pattern builders. In the case of restoring data in a parity bank, the ReCoding unit downloads and stores from the data banks over one or more memory cycles and restores the parity code once all requisite data has been retrieved. For storing a write which has been stored in a parity bank, the ReCoding unit writes the new data to the parity bank and then begins to restore the parity code. If in the execution of the read and write pattern builders data which is useful to the ReCoding unit is accessed, the data is sent to the ReCoding unit.

4.6 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 16 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 17 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.

[MATT: We may need to use a different example if we don't end up using these traces. The parsec benchmarks only have 2 heavily accessed regions, and though this is simpler than the LTE/UMTS example, it is still demonstrates the idea behind the dynamic encoder] Figure 18 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 7.

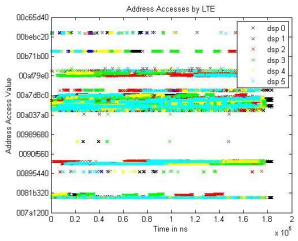


Figure 16: Memory Access by LTE Cores

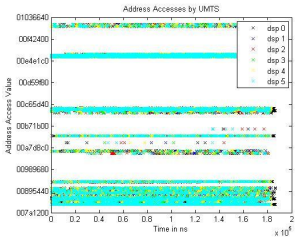


Figure 17: Memory Access by UMTS Cores

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

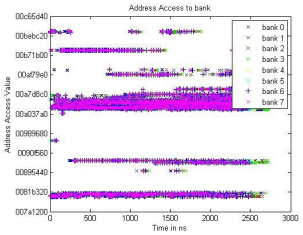


Figure 18: Memory Access per bank for LTE

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. *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.

[MATT: TODO: Discuss what condition triggers a switch. Also, to what degree does decreasing the length of the sub-region increase the complexity of the dynamic encoder? How small can the subregions be until the complexity is too great?]

4.7 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

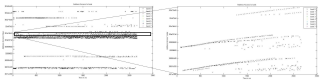


Figure 19

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 19 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 20, 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.

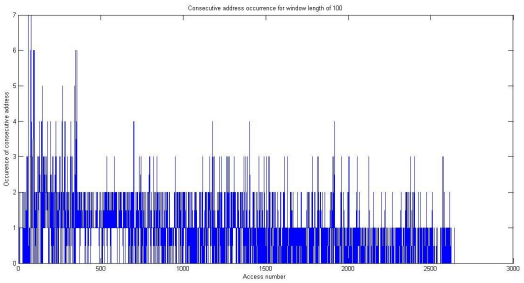


Figure 20

5. EXPERIMENTAL METHODOLOGY

6. SIMULATION RESULTS

6.1 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.

7. ACKNOWLEDGEMENTS

This document is derived from previous conferences, in particular HPCA 2017. We thank Daniel A. Jimenez, Elvira Teran for their inputs.

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

8. REFERENCES

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[2] M. M. Waldrop, "The chips are down for Moore's law." *Nature*, vol. 530, no. 7589, pp. 144–147, 2016.

[3] T. Simonite, "Moore's law is dead. Now what?" https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/, 2016, online; accessed 31 July 2017.

[4] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, May 2005.

[5] W.-F. Lin, S. K. Reinhardt, and D. Burger, "Reducing DRAM latencies with an integrated memory hierarchy design," in *Proceedings of 7th International Symposium on High-Performance Computer Architecture (HPCA)*, Jan 2001, pp. 301–312.

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

[7] A. G. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Transactions on Information Theory*, vol. 56, no. 9, pp. 4539–4551, Sept 2010.

[8] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin, "On the locality of codeword symbols," *IEEE Transactions on Information Theory*, vol. 58, no. 11, pp. 6925–6934, Nov 2012.

[9] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai, "Batch codes and their applications," in *Proc. of thirty-sixth annual ACM symposium on Theory of computing (STOC)*, 2004, pp. 262–271. [Online]. Available: http://doi.acm.org.ezproxy.lib.utexas.edu/10.1145/1007352.1007396

[10] A. S. Rawat, D. S. Papailiopoulos, A. G. Dimakis, and S. Vishwanath, "Locality and availability in distributed storage," *IEEE Transactions on Information Theory*, vol. 62, no. 8, pp. 4481–4493, Aug 2016.

[11] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible DRAM simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, Jan 2016.

[12] T. Suzuki, H. Yamauchi, Y. Yamagami, K. Satomi, and H. Akamatsu, "A stable 2-port sram cell design against simultaneously read/write-disturbed accesses," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 9, pp. 2109–2119, Sept 2008.

[13] D. P. Wang, H. J. Lin, C. T. Chuang, and W. Hwang, "Low-power multiport sram with cross-point write word-lines, shared write bit-lines, and shared write row-access transistors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 3, pp. 188–192, March 2014.

[14] D. J. Auerbach, T. C. Chen, and W. J. Paul, "High-performance multiple port memory," US Patent US 4 766 535, Aug 23, 1988. [Online]. Available: https://www.google.com/patents/US4766535

[15] K. Endo, T. Matsumura, and J. Yamada, "Pipelined, time-sharing access technique for an integrated multiport memory," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 4, pp. 549–554, Apr 1991.

[16] R. L. Rivest and L. A. Glasser, "A fast multiport memory based on single-port memory cells," Massachusetts Inst. of Tech. Cambridge Lab. for Computer Science, Tech. Rep. MIT/LCS/TM-455, 1991.

[17] S. Iyer and S.-T. Chuang, "System and method for storing multiple copies of data in a high speed memory system," US Patent US 8 935 507 B2, Jan 13, 2015.

[18] ——, "System and method for storing data in a virtualized high speed memory system with an integrated memory mapping table," US Patent US 8 504 796 B2, Aug 6, 2013.

[19] B. A. Chappell, T. I. Chappell, M. K. Ebcioglu, and S. E. Schuster, "Virtual multi-port ram," Apr 1993. [Online].

Available: https://www.google.com/patents/US5204841

[20] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*. Amsterdam: North-Holland, 1983.

[21] T. M. Cover and J. A. Thomas, *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, 2006.

[22] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Proc. of 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX, 2012, pp. 15–26.

[23] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing elephants: Novel erasure codes for big data," in *Proc. of 39th International Conference on Very Large Data Bases (VLDB)*, vol. 6, no. 5, June 2013, pp. 325–336.

[24] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 331–342, Aug. 2014.

[25] A. S. Rawat, Z. Song, A. G. Dimakis, and A. Gal, "Batch codes through dense graphs without short cycles," *IEEE Transactions on Information Theory*, vol. 62, no. 4, pp. 1592–1604, April 2016.

[26] Z. Wang, H. M. Kiah, Y. Cassuto, and J. Bruck, "Switch codes: Codes for fully parallel reconstruction," *IEEE Transactions on Information Theory*, vol. 63, no. 4, pp. 2061–2075, Apr. 2017.

[27] N. P. Anthapadmanabhan, E. Soljanin, and S. Vishwanath, "Update-efficient codes for erasure correction," in *Proc. of 48th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, Sept 2010, pp. 376–382.

[28] A. Mazumdar, V. Chandar, and G. W. Wornell, "Update-efficiency and local repairability limits for capacity approaching codes," *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 5, pp. 976–988, May 2014.

[29] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–12.

[30] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jimenez, "B-fetch: Branch prediction directed prefetching for chip-multiprocessors," in *Proceedings of 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014, pp. 623–634.

[31] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015, pp. 141–152.

[32] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proc. of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: ACM, 2013, pp. 247–259.

[33] M. A. Maddah-Ali and U. Niesen, "Coding for caching: fundamental limits and practical challenges," *IEEE Communications Magazine*, vol. 54, no. 8, pp. 23–29, Aug 2016.