Coded Access Architectures for Dense Memory Systems

ABSTRACT

We explore the use of coding theory in double data rate (DDR) and high bandwidth memory (HBM) systems. Modern DDR systems incur large latencies due to contention of multiple requests on the same memory bank. Our proposed memory design stores data across DRAM pages in a redundant manner using Reed-Solomon codes as a building block. A memory controller then assigns coded versions of the data to dedicated parity banks. This multi-bank coding scheme creates multiple ways to retrieve a given data element and allows for more efficient scheduling in multi-core memory architectures such as HBM. Our approach differs from conventional, uncoded systems which only optimize the timing of each incoming memory request.

We implement our proposed memory design on an HBM DRAM architecture via the Ramulator simulation platform. Experimental results show that multi-bank coding reduces the number of contended memory accesses, and thus the overall latency, for several standard benchmarks. Our design reduces the number of CPU cycles by nearly 70% compared to an uncoded baseline.

1. INTRODUCTION

In this era of disruptive development in the fields of data science and machine learning, computing systems with high processing capacity are in huge demand. The processing power of a system is mainly determined by its compute capability and memory operations. Over the past 50 years, computing systems have witnessed sustained growth, where according to Moore's law, the computational power of these computing systems has doubled every 18 months [**?**]. On the other hand, memory access speed has failed to match this growth in computational power [**?**]. Since memory accesses are an essential part of any program, the stark gap between memory access speed and CPU speed implies that the memory access is a major bottleneck towards increasing processing capabilities [**?**]. In order to reduce large data transfer times between the processor and the memory, researchers have improved data access capacity and speed via innovations in integrated circuit design and memory scheduling. However, these efforts fall short of keeping the memory access latency low enough to meet the growing demand for computation.

In general, recent efforts towards increasing compute capability have utilized multiple cores and high speed architectures. These approaches have also failed to deliver their intended benefits, mainly due to slow memory systems which cannot keep up with requests and therefore delay the overall processing speed [**?**, **?**, **?**, **?**]. Additionally, in a multi-core setup, the access requests from one core can interfere with requests from other cores. These contentions further increase access latency. For example, two cores issuing access requests for data elements stored on the same memory bank results in a *bank conflict*. Since the memory bank can serve only one access request per memory clock cycle, the second request must be queued for future clock cycles. As the number of cores increases, such bank conflicts become more frequent. This leads to many access requests being queued and waiting to be served by the slow shared memory.

In this paper we aim at improving the accesses of next-generation memory systems such as High Bandwidth Memory (HBM) and Hybrid Memory Cube (HMC). We propose a novel solution that not only improves access efficiency for memory reads and writes, but also mitigates bank conflicts which arise in a multi-core setup. In particular, we employ coding theoretic techniques to introduce redundancy to data storage. We then design a retrieval architecture that exploits this redundancy to provide parallel low latency memory access. Existing solutions to address latency-related issues in memory systems are mostly limited to improving command scheduling and clever data addressing. In contrast, our solution calls for completely redesigning the memory storage space. Our unique approach creates multiple ways to serve a read request for each data element. This allows us to design efficient retrieval mechanisms which spread the memory accesses intended for a particular memory bank across multiple other banks. As a result, the proposed memory design can simultaneously serve read requests that would cause bank conflicts and queues in existing systems. We demonstrate the utility of our proposed memory design by implementing it with DDR and High Bandwidth Memory (HBM) protocols, although our general framework is applicable to other memory architectures.

Organization: The rest of this paper is organized as follows. In Section 2 we introduce basics of a multi-core setup along with the necessary background on high bandwidth memory (HBM) protocols and coding techniques. We then discuss emulation of multi-port memories using single-port memory banks and present a detailed account of relevant prior work. In Section 3.1, we propose a coding-based framework to design the storage space of the memory. Specifically, we focus on a specific bank array design which utilizes Reed-Solomon

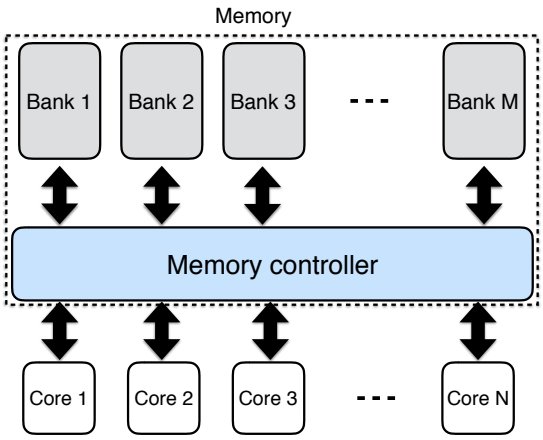


Figure 1: Multi-core Architecture with shared memory.

(RS) codes. Then in Section 3.2, we present a novel memory controller architecture that aims to exploit this coded bank array to improve read and write accesses to the memory. In Section 4, we evaluate the proposed RS code based memory design on an HBM architecture using the Ramulator DRAM simulation platform [**?**]. We conclude the paper in Section 5.

2. BACKGROUND

This section provides relevant background information multi-core memory architectures, and coding theory, and related work.

2.1 Multi-core Setup and Bank Conflicts

We consider the generic multi-core architecture illustrated in Figure 1, where N processor cores rely on a shared memory consisting of M memory banks. All the cores operate in a parallel manner and issue access requests to the shared memory in the event of last level cache misses. These access requests are received by memory controller, which then arbitrates and schedules the requests to be served by different memory banks based on the information stored on the memory banks. In particular, the memory controller maintains different queues for each of the M memory banks, which hold the read and write requests corresponding to the associated bank. These queues are then sequentially served every memory cycle and the acknowledgment with data (in the case of a read requests) is sent to the processor which issued the access request.

Assuming that the storage space of the shared memory consists of single port memory banks, i.e., it can support a single access request during a memory cycle, multiple access requests issued for content stored on the same bank cannot be served in a single memory cycle. This event is known as a *bank conflict* and leads to increased access latency. The effect of these bank conflicts on the overall access latency of the shared memory becomes even more pronounced as the number of cores increases. Thus, the memory controller needs to avoid such bank conflicts while mapping the access requests from different cores to the memory banks. One straightforward solution to mitigate bank conflicts is to employ multi-port memory banks which can support multiple access requests to the same bank in a single memory cycle [**?**]. However, the existing designs for multi-port memory banks suffer from increased circuit complexity and large area overhead [**?**]. Furthermore, in a multi-core setup with large number of cores, it is very likely for a bank to receive multiple simultaneous accesses that can exceed the number of ports present at the bank. Therefore, it's imperative to explore other solutions to the bank conflict problem beyond simply employing multi-port memory banks.

2.2 Coding Techniques: Preliminaries

Coding theory is the study of various approaches to transform available information into alternative representations with the aim of precisely controlling data redundancy, which enables reliable and efficient processing of the information. Coding has been successfully used in various fields of engineering and computer science, including communications, compression, cryptography, and data storage. The underlying transformation is referred to as a code, and its specific form depends on the application at hand.

2.2.1 Block Codes

In coding theory literature, block codes are one of the most studied classes of codes. An (n,k) block code transforms (encodes) k message symbols belonging to a finite alphabet Σ_1 to n code symbols belonging to another finite alphabet Σ_2. In many applications, these finite alphabets are finite fields which are equipped with addition and multiplication operations. The binary set $\{0,1\}$ with modulo 2 addition and usual multiplication is an example of a finite field. In addition, we assume that both alphabets are the same finite field $\mathbb{F}$. Under this setup, an (n,k) block code encodes vectors in $\mathbb{F}^k$ to vectors in $\mathbb{F}^n$. The vectors in $\mathbb{F}^n$ that are obtained through the encoding process are called *codewords*.

The quality of a code is characterized by two parameters: 1) *Rate*, which measures the amount of redundancy introduced during the transformation; and 2) *Minimum distance*, which measures the maximum number of code symbols that can fail without compromising the code's ability to recover the correct message symbols. The rate of a block code is defined as

$$\rho = \frac{k}{n}.$$

The minimum distance of the code $d_{\min}$ is defined as the minimum number of positions at which any two codewords differ. A block code with the minimum distance $d_{\min}$ can allow reconstruction of the original message even after the loss of any $d_{\min} - 1$ codeword symbols [**?**]. Ideally, one would like both the rate and the minimum distance of the underlying code to be as large as possible, however, there is a trade-off between these two parameters [**?**]. Most notably, the Singleton bound dictates that $d_{\min} \leq n - k + 1$. Maximum distance separable (MDS) codes attain this bound with equality which makes them optimally resource efficient (as these codes achieve the highest possible rate) for a given failure tolerance.

Reed-Solomon (RS) codes [**?**] are the most celebrated family of MDS codes that exist for all n and k. However, these codes typically require working with large field sizes that scale linearly with n. That said, we can map the arith-

metic over these large fields to the operation of binary field by viewing each element of the large field as a bit vector, e.g., see [**?**, **?**]. We will rely on vectorized RS codes in this paper. In contrast to other coding schemes that are employed in existing memory systems like SEC-DED [**?**], BAMBOO [**?**], and ChipKill [**?**], the RS based memory design in this paper allows for more intelligent memory controller design by improving data access.

2.2.2 Encoding Memory Banks

As mentioned above, a coding scheme is characterized by an encoding process that converts message to codewords. In the context of memory banks, we will restrict ourselves to systematic encodings, where message symbols appear as part of the codewords. In particular, we arrange the given information on some memory banks which are referred to as *data banks*. Then we generate *parity symbols* which are functions of the symbols stored in the data banks and store these parity symbols in new sets of memory banks, referred to as *parity banks*. Note that these additional banks, constitute the redundancy in the storage space. Furthermore, in this paper, we only consider encoding across the same row address, i.e., only the content stored on a given row address in the data banks is utilized to generate the parity symbols stored on the same row address in the parity banks.

The following example helps illustrate the key concepts and notation related to encoding memory banks to generate redundant storages space in memory systems.

EXAMPLE 1. *Consider the setup where information is arranged in two data banks* $\mathbf{a}$ *and* $\mathbf{b}$*. Each bank has L rows, each of which can store W bits. Therefore, each bank can be viewed as an* $L \times W$ *array. For* $i \in [L] \triangleq \{1,\ldots,L\}$*, let* $a(i)$ *and* $b(i)$ *denote the i-th row of the bank* $\mathbf{a}$ *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* $i \in [L]$*,*

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

Now, given a map $f : \{0,1\}^W \times \{0,1\}^W \to \{0,1\}^W$*, we generate a parity bank* $\mathbf{p} = f(\mathbf{a},\mathbf{b})$ *such that, for* $i \in [L]$*, we have*

$$p(i) = \big(p_{i,1},\ldots,p_{i,W}\big) = f\big(a(i),b(i)\big). \tag{1}$$

Among many choices, if we take the function f *to be a bit-wise XOR, then we get* $\mathbf{p} = \mathbf{a} \oplus \mathbf{b}$*, i.e., for* $i \in [L]$*,*

$$\begin{aligned} p(i) &= \big(p_{i,1},\ldots,p_{i,W}\big) = a(i) \oplus b(i) \\ &\triangleq \big(a_{i,1} \oplus b_{i,1},\ldots,a_{i,W} \oplus b_{i,W}\big). \end{aligned} \tag{2}$$

Figure 2 illustrates this coding scheme.

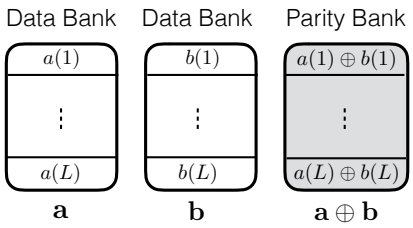


Figure 2: Parity bank obtained by performing bit-wise XOR on the two data banks $\mathbf{a}$ and $\mathbf{b}$.

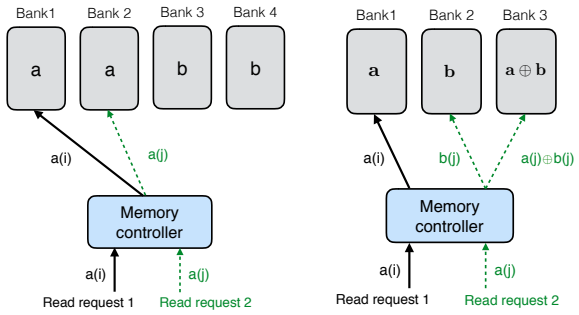
2.3 Emulating Multi-port Memories

A multi-port memory supports multiple simultaneous accesses to content stored in the same memory bank. As evident, such a memory reduces bank conflicts, where some access requests might get delayed due to a single access request accessing a particular memory bank. Therefore, multi-port memories form an essential component of a high performance multi-core setup. That said, designing true multi-port memory banks incurs large cost both in terms of circuit complexity and memory density as compared to single-port memory banks [**?**, **?**].

This has motivated various research efforts to explore algorithm and/or system level designs to emulate the functionality of a multi-port memory using a collection of single-port memory banks, e.g., see [**?**, **?**, **?**, **?**, **?**, **?**] and references therein. The key idea in most of the work in this direction is to employ single port memory banks to store the content in a redundant manner. As a result, when the memory system encounters concurrent accesses leading to bank conflict in a memory bank, it uses the redundancy in the memory banks to create ways to simultaneously serve these accesses using disjoint groups of memory banks. The design of redundant storage space using single-port memory banks is typically based on two general approaches: 1) replication [**?**]; and 2) erasure coding [**?**, **?**, **?**].

We briefly illustrate in Fig. 3 how such emulations of multi-port memories work. In particular, consider a setup where we need to store $2L$ rows of data $\mathbf{a} = [a(1),\ldots,a(L)]^T$ and $\mathbf{b} = [b(1),\ldots,b(L)]^T$. We are given single-port memory banks with each bank having capacity to store L rows of data. In order to simplify the exposition, we ignore the write requests for the moment. Our objective is to emulate a multi-port memory that can support 2 simultaneous read requests to $\mathbf{a}$ and $\mathbf{b}$. Fig. 3a describes a replication-based design to support 2 simultaneous read accesses. Similarly, Fig. 3b shows how one can generate a parity bank by taking element-wise XOR of two data banks to enable 2 simultaneous accesses. As it is evident from this example, in order to support the same number of read accesses, the replication-based scheme incurs higher storage cost as it requires more single-port memory banks to emulate a multi-port memory bank. This observation is consistent with similar trends observed in other applications, which usually provide opportunities for sophisticated coding schemes to replace replication schemes [**?**, **?**]. However, in order to make a coding-based solution viable, especially in the context of memory systems, it is essential to employ coding schemes with low computational complexity.

It is easy to generalize the replication scheme (cf. Fig. 3a) to support r simultaneous read accesses by storing r copies of each data element on r distinct single-port memory banks. Similarly, it is possible to employ more sophisticated coding schemes compared to the one in Fig. 3b to support multiple concurrent read accesses [**?**, **?**, **?**]. However, in order to translate this approach to a viable solution for a mem-



(a) 2-replication scheme. (b) Bit-wise XOR.

Figure 3: Supporting multiple simultaneous read accesses using single-port memory banks. Let's consider two concurrent requests for data $\{a(i), a(j)\}$ which cause bank conflict in a single memory banks. 1) For the replication-based design, 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. 2) In the coded memory system, as shown in Figure 3b, we can deal with bank conflict in the following manner: 1) First request for $a(i)$ can be directly served by Bank 1 , 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.

ory system, it is necessary to account for write requests as well. Note that the key challenge in employing a redundant storage space in the presence of write requests is to manage consistency across multiple copies of the data. In addition, one needs to ensure that the stale content, i.e., the content modified by a write request, is not supplied in response to a later read request.

Towards this, Auerbach et al. demonstrate a replication-based emulation of multi-port memory using single-port memory banks [**?**]. This emulation simultaneously supports r read accesses and w write accesses by storing $r \cdot (w+1)$ copies of each data element on $r \cdot (w+1)$ distinct single-port memory banks. In order to see this, assume that these banks are partitioned in r groups with each group containing $w+1$ copies of each data element. The r different groups of banks are used to serve r distinct read accesses. Assuming that each group has at least one bank that stores the updated (valid) copy of each data element, we can use that bank to serve the read request from the group. In parallel, for every group, we can perform w write requests on w unused banks in the group. Note that this process does not cause any bank conflicts and all the write requests do get performed inside each group.

Focusing on coding-based memory designs to support concurrent read accesses, we can modify the ideas from [**?**] to support write requests as well. We can take set of banks that support r simultaneous read accesses and replicate it $(r+w)$ times [1]. Each of these copies is referred to as a *group* of banks. Now, given r read requests we look for the minimum number of groups that store the most updated version of the data elements associated with these read requests and serve

[1]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.

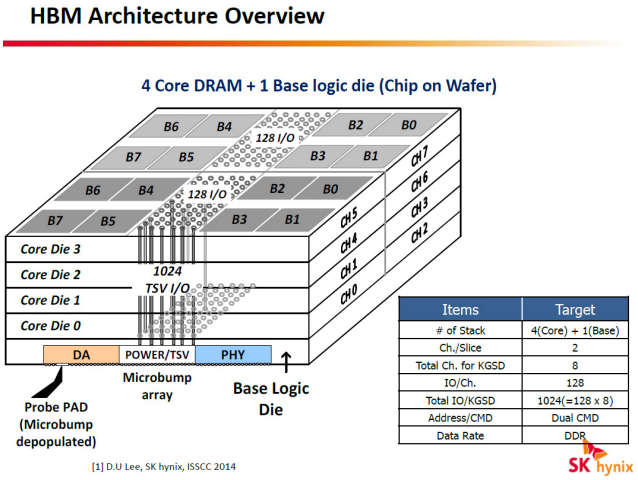


Figure 4: High Density Memory (HBM) architecture[2]. The 3D stacking of memory channels provides an orthogonal plane to pack a large amount of storage in a smaller footprint while maintaining the system's electromagnetic characteristics.

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 accordingly.

REMARK 1. *Note that the replication-based emulation described above incurs a large overall storage cost as this approach has information rate* $1/(r \cdot (w+1))$. *Moreover, even though the coding-based scheme is storage-efficient in the presence of only read requests, the need to accommodate write requests makes the storage cost of this approach prohibitively large as well. Assuming that the coding scheme has information rate* ρ, *the final rate after* $(r+w)$ *replications is* $\rho/(r+w)$. *In addition to the memory banks, these designs also require additional storage space to store pointers which keep track of which memory banks store the latest versions of the data elements. In order to manage this additional storage requirement, a practical implementation must periodically synchronize every bank with the most recent version of the data.*

In the memory architecture proposed in this paper, we completely do away with the additional replications required by the aforementioned emulation approaches to incorporate write requests. In this paper, we instead propose to handle the write requests by modifying the design of the memory controller (cf. Section 3.2). This way, we manage to preserve the storage-efficiency of a coding-based solution.

2.4 High Bandwidth Memory

High bandwidth memory (HBM) standard is defined to enable future high-performance devices such as GPUs. Figure 4 shows a stack of HBM, where it shows 4 memory core

[2]Courtesy: `http://www.skhynix.com`

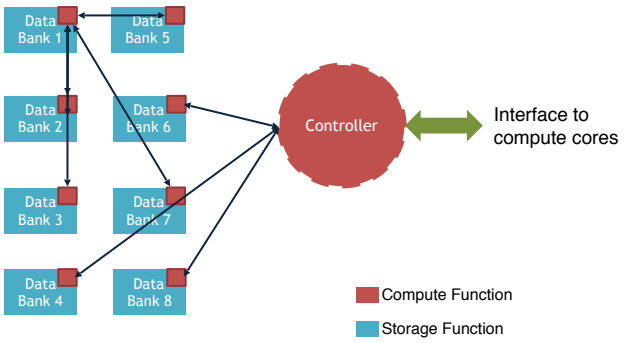


Figure 5: Block diagram description of new memory system architectures with distributed memory controllers. This architecture includes a control logic closer to the data bank. This control logic is capable of certain fixed computation that can be exploited to reduce the access bandwidth between the bank and the main controller.

dies vertically integrated together. Each die has 2 channels, each one with 8 DRAM banks ($B0$ - $B7$). This memory design enables a 3D scaling of memory, which packs more data in the same space as compared to 2D scaling, while maintaining the I/O signal integrity. The architecture also helps the computing cores to parallelize their access channels, meeting the high data access requirements while consuming less power. Here we only mention the key features of the HBM architecture that are relevant to our goal of reducing access latency using coding techniques. We refer the readers to [**?**] for a detailed account of various features and specifications of the architecture.

The key design feature of HBM that allows for parallel access to channels facilitates coded memory storage and retrieval. This can allow for various storage schemes where the data and codes can be stored.

REMARK 2. *We note that the Hybrid Memory Cube (HMC) architecture [**?**] is somewhat similar to the HBM architecture discussed here. While our proposed approach could be used in either system, this paper focuses on HBM because it has more publicly available information and data for conducting experiments. The HMC architecture has a provision where the memory controller can be stored closer to the memory. Additionally, it is also possible to integrate a small logic function next to each bank, which interfaces with the memory controller and performs certain preset arithmetic functions. As shown in Figure 5, each data bank is now capable of computing codes on the fly by accessing elements from its memory and constructing arithmetic combinations with its locally available logical block. This capability helps the code designers to structure codes which can be constructed dynamically to improve overall efficiency of the system. Local arithmetic also allows designers to draw from similar results and techniques from the field of distributed memory systems for large data servers. The HMC architecture also allows for inter-bank communication, where the data from one bank can directly transfer to another bank without going through the main controller. This can allow for optimal reorganization of the memory to improve the access latency.*

2.5 Related Work

Coding for distributed storage systems: In this paper, we employ coding techniques to store information across multiple memory banks. This is very similar to the application of coding to store data over distributed network of storage nodes in large scale storage systems [**?**, **?**, **?**, **?**, **?**]. However, coding for memory systems comes with stringent architectural constraints that need to be taken into account. A memory controller should not be required to perform complex computation during the accesses to the coded information. Moreover, the coding for memory systems also needs to support a significant portion of write requests during memory accesses. Therefore, even though application of coding for improved access latency has been previously explored in distributed storage systems [**?**, **?**, **?**, **?**], achieving the same goal in the context of memory systems is a challenging problem.

Coding for shared memory: In [**?**, **?**, **?**], the authors develop a coded data storage architectures which mainly applies to integrated memories in system-on-chip (SoC) subsystems. The proprietary memory design on SoC provides a lot of flexibility to design and implement a complicated coding scheme with less overhead. However, such architectures are limited to a certain subset of products and do not scale to ubiquitous DRAM memories deployed in most computing systems today. The more prevalent DRAM memories are manufactured by multiple vendors and are governed with set industry standards and protocols. Interoperability with commercial devices (and conformity to standards) requires a new design for coded data storage, one which can be implemented by enhancing the memory controller and enables the benefits of a coded architecture. The coding schemes described in [**?**, **?**, **?**] are suboptimal for the standard memory systems including DRAM and HBM as their specific architectures (cf. Section 2.4) provide the opportunities for further optimization.

In [**?**, **?**, **?**], the authors also explore coding theoretic techniques in memory systems in order to emulate multi-port memory using single-port memory banks. However, we note that the overall solution proposed in this paper significantly differs from this prior work in both the employed code design and the accompanied memory controller architecture.

As mentioned earlier, there have been multiple efforts in realizing multi-port memory functionality using various circuit designs (e.g., see [**?**] and references therein). In this paper we focus on a coding theoretic approach to explore an orthogonal design space. We believe that the previously studied techniques, e.g. pipelined, time-sharing access [**?**], may very well be combined with our coding theoretic framework in order to obtain a more comprehensive solution for high performance memory systems.

3. CODED MEMORY SYSTEM

In this paper, we aim to extend the benefits of coding theoretic techniques to dynamic random-access memory systems. In particular, we achieve this by storing the information across single port memory banks in a redundant manner. Traditionally, when multiple requests to a single bank are issued by the cores, a stall occurs as only one address from a single bank can be accessed at a time. As a result, the cores must wait for the first request to the bank be served before their other requests to the bank can be processed. This is where a redundant storage space based on a carefully designed coding scheme comes to the rescue. During a given memory cycle, any data bank that is blocked/busy serving an access can be viewed as a failure as far as other accesses are concerned. Now, other accesses intended for this blocked bank can be served by accessing the content stored in a group of other memory banks. This way, redundant storage space allows concurrent accesses to the same information and alleviates the stalls that are unavoidable in a memory system comprised of non-redundant storage space.

In this section, we describe our specific memory design in detail. First, we present how to use Reed-Solomon codes to introduce redundancy in the memory storage space. We then discuss the memory controller, the other essential block of our design. Our proposed controller exploits memory bank redundancies to reduce the effect of concurrent read accesses to the stored information. Additionally, the memory controller also maintains consistency and validity of the stored data, which arises due to the presence of write accesses.

3.1 Coded Multi-bank Storage Space

We rely on a systematic Reed-Solomon (RS) coding scheme as a building block to design the storage space of our proposed memory system. In particular, the storage space consists of 16 memory banks[3] with 8 memory banks serving as the data banks and the remaining 8 banks serving as the parity banks (cf. Section 2.2.2). As detailed later in this subsection, the parity banks are generated using a systematic $(6,4)$ RS code which is defined over a finite field of size 2^8, denoted by $GF(2^8)$. In any codeword of this systematic $(6,4)$ RS code, it's possible to generate all 4 associated message symbols given any 4 symbols of the codeword.

Before we describe the process of generating the parity blocks, we discuss the reasons behind employing RS codes as opposed to other possible coding schemes. First, RS codes are MDS codes (cf. Section 2.2.1), i.e., these codes are optimal in terms of their storage efficiency. Additionally, decades of study has lead to efficient encoding and decoding schemes for RS codes. Crucially, these schemes can be easily *hard-coded* into the memory controller in order to achieve good runtime performance. Finally, RS codes have been proven to be useful in previous work on distributed storage, both in theory and in practice [**?**, **?**, **?**]. In fact, $(14,10)$-RS code is used in production at Facebook in their HDFS-RAID system [**?**]. This makes these codes a natural starting point for our dense memory storage architecture.

We now describe the detailed design of the storage space of our proposed memory system. Let the underlying systematic $(6,4)$ RS code map 4 message symbols $u,v,w,x \in GF(2^8)$ to a codeword containing 6 symbols

$$\big(u,v,w,x,f(u,v,w,x),g(u,v,w,x)\big) \in GF(2^8)^6,$$

where $f,g : GF(2^8)^4 \to GF(2^8)$ are two linear maps. We arrange the information in 8 data banks $B0,B1,\ldots,B7$ as shown in Figure 6. The parity banks $P0,P1,\ldots,P7$ then

[3]We believe that a 16 bank design is quite reasonable given that the next generation memory architectures feature many more memory banks than current DRAM architectures.

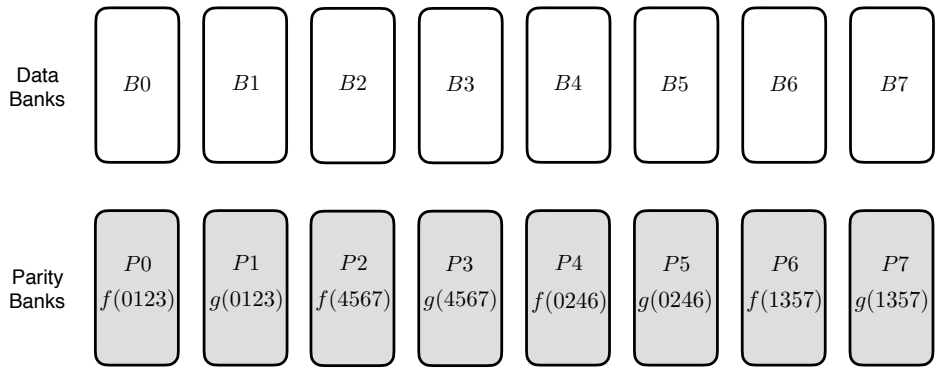


Figure 6: Data storage layout for coded memory system

store the parity symbol which are generated according to the $(6,4)$ RS code described above[4]. In particular, the first two parity banks $P0$ and $P1$ store the parity symbols that are functions of the information stored in the first four data banks $B0, B1, B2$, and $B3$. Note that this implies that the symbols stored on the same row address of the six banks $\{B0, \ldots, B3, P0, P1\}$ form codewords of the underlying $(6,4)$ RS code. Similarly, the parity symbols generated from the remaining four data banks $B4, B5, B6$, and $B7$ are stored in the parity banks $P2$ and $P3$.

In our design, we ensure that each data bank is part of 2 codewords of the $(6,4)$ RS code, which are formed by disjoint sets of parity banks. This creates disjoint ways of recovering the information stored in the same data bank. We utilize the parity banks $P4, P5, P6$, and $P7$ to generate the second codewords associated the data banks. In particular, parity banks $P3$ and $P4$ store the parity symbols generated from the information stored on the even numbered data banks $B0, B2, B4$, and $B6$. Similarly, the parity banks $P6$ and $P7$ store the parity symbols generated from the odd numbered data banks $B1, B3, B5$, and $B7$.

REMARK 3. *Note that the redundancy in the storage space helps enable concurrent read accesses by treating the blocked data banks as failures. For example, assume that $B0$ and $B1$ are blocked due to some read accesses. Now, another read access to $B0$ can be realized by using $B2, B3, P0 = f(B0, B1, B2, B3)$, and $P1 = g(B0, B1, B2, B3)$ to recover the symbol in $B0$ required by the access. Similarly, consider a different scenario, where $B0$ is busy serving a read access. Now, another read access for the information stored in $B0$ can be served by reconstructing that information with the help of three data banks $B1, B2, B3$, and one parity bank $P0 = f(B0, B1, B2, B3)$ or $P1 = g(B0, B1, B2, B3)$.*

3.2 Memory Controller Design

The proposed architecture of the memory controller consists of a scheduler which converts memory access requests from the cores into memory access patterns that are mapped to different memory banks. The scheduler has three main components: 1) *reorder buffer*, 2) *read algorithm*, and 3) *write algorithm*. The reorder buffer stores the list of pending or recently served access requests by the memory system. In addition, the reorder buffer also keeps track of the status of the data stored on memory banks as it gets *stale* (*invalid*) as a result of write requests. The scheduler compares each read and write request to the content of the reorder buffer in order to decide how the request gets served by the read algorithm and the write algorithm, respectively. Next, we describe each of these three main components of the memory controller in detail. Table 1 shows the behavior of our proposed memory controller architecture on an example request pattern.

3.2.1 Reorder buffer

As mentioned earlier, all access requests need to go through the reorder buffer. The reorder buffer stores the data downloaded from the memory banks in order to serve read access requests from the cores. Similarly, the reorder buffer also stores the new data that arrives with write requests and need to be written on the coded memory banks (after generation of consistent parity data). In addition to storing the actual data, the reorder buffer maintains metadata to keep track of the rows of the memory banks that are present in the reorder buffer and the validity of these rows across data banks and parity banks. Each write request invalidates the content of (some of) the data banks. The data on these banks (and on the corresponding parity banks which depend on them) then needs to be updated. The metadata of the reorder buffer is used to ensure this consistency across the data and parity banks when the memory controller maps a write request to be written on the memory banks. The reorder buffer size is a key design parameter which trades off faster access speeds for increased overhead, as shown in Section 4.

For each row address in the memory banks, the reorder buffer stores *at most* 1 entry along with the following 20 bits of metadata (cf. Table 1).

- *Done*: This bit indicates if this particular row address had a write request associated with it in the past that has not yet been written to the memory banks. If *Done* bit is set to 1, this implies that this row address was affected by the write requests and the content on the memory banks need to be updated when this row is removed from the metadata of the reorder buffer.

[4]Recall the encoding process and associated notation defined in Section 2.2.2. For example, the functions f and g are applied to the symbols stores in a given row address of $B0, B1, B2$, and $B3$ to obtain the symbols stored in the same row address of the parity banks $f(0123) \triangleq f(B0, B1, B2, B3)$ and $g(0123) \triangleq g(B0, B1, B2, B3)$, respectively.

	ROW	*Done*	R/W	D_{valid}	P_{valid}	DB_0	DB_1	DB_2	$\cdots$	DB_7	PB_0	PB_1	PB_2	$\cdots$	PB_7	
t=1	0x2	0	R	0	0	1	1	0	0	0	0	0	0	0	0	Read $B0 - B1$
t=2	0x2	0	R	1	1	1	1	1	1	1	1	1	1	1	1	Read $B0 - B7$
t=3	0x2	0	W	0	0	0	1	1	1	1	0	0	0	0	0	Write $B0$
t=4	0x2	0	W	1	0	1	1	1	1	1	0	0	0	0	0	
t=5	0x2	0	R	1	0	1	1	1	1	1	0	0	0	0	0	Read $B4 - B6$
t=6	0x2	1	R	1	1	1	1	1	1	1	1	1	1	1	1	
t=7	0x2	1	R	1	1	1	1	1	1	1	1	1	1	1	1	Read $B0 - B7$

Table 1: Example of a single entry 0x2 in the reorder buffer across time during a Read-Write-Read request pattern. At t=1, there is a read from $B0$–$B1$, followed by a read at $t = 2$ from $B0$–$B7$ which includes both data and parity banks. A write request to $B0$ occurs from $t = 3$ to $t = 6$, during which DB_0 is updated, D_{valid} is set to 1, bits for parities banks, i.e., PB_0–PB_7, are updated, P_{valid} is set to 1, and *Done* is set to 1. Since data banks are valid after $t = 4$, a read from $B4$–$B6$ is served using only data banks at $t = 5$, which sets the R/W bit to R. A read from $B0$–$B7$ occurs at $t = 7$, which does not change the state of the reorder buffer.

- R/W: This bit denotes the nature of the most recent access request for the row address associated with the entry in reorder buffer. R and W refers to the read request and the write request, respectively.
- D_{valid}: This bit indicates if the reorder buffer stores the valid data for the corresponding row address in each of the *data* banks. D_{valid} bit is set to 1 when the reorder buffer has the valid copies of the content for all the data banks.
- P_{valid}: This bit indicates if the reorder buffer stores the valid data for the corresponding row address in each of the *parity* banks. P_{valid} bit is set to 1 when the reorder buffer stores the valid copies of the data for all the parity banks.
- $DB_0, \ldots, DB_7$: Similar to the D_{valid} bit, for $0 \leq i \leq 7$, DB_i bit is set to 1 to denote that the reorder buffer stores the valid data for the associated row address in the i-th data bank Bi.
- $PB_0, \ldots, PB_7$: Similar to the P_{valid} bit, for $0 \leq i \leq 7$, PB_i bit is set to 1 to indicate that the reorder buffer stores the valid version of the content for the associated row address in the i-th parity bank Pi.

Note that D_{valid} and P_{valid} are logical *AND*s of $DB_0 - DB_7$ and $PB_0 - PB_7$, respectively.

REMARK 4. *Even though the data stored in a particular row in the memory banks might be invalid, the reorder buffer may store its valid version which would lead to the* D_{valid} *and/or* P_{valid} *bits to be set. It's the responsibility of the write algorithm to transfer this valid data from the reorder buffer to the memory banks when the corresponding row is evicted from the reorder buffer.*

REMARK 5. *Here, we note that the reorder buffer can also be utilized to incorporate different priority levels among access requests, if the memory systems is required to support multiple access types with different priority levels.*

3.2.2 Read algorithm

For a given state of the reorder buffer and the set of pending read requests from the cores, the main purpose of the read algorithm is to maximize the number of read requests that can be served in a given memory cycle. This is accomplished by utilizing the redundant parity banks in an efficient manner without increasing the algorithm's overall complexity. Note that in our coded memory design (cf. Section 3.1), a parity bank is useful during data access only if some of the corresponding data banks are also accessed. In particular, assuming that all the relevant parity banks are valid, there are two possible ways of serving a read request by employing the parity banks: 1) content of 2 data banks and 2 parity banks is utilized; and 2) content of 3 data banks and 1 parity bank is utilized (cf. Remark 3). In every cycle, the scheduler picks pending read requests *in order* and tries to serve them according to the read algorithm described in Figure 7. In addition to this, whenever all but 1 data bank is present in the buffer, that data bank is automatically read from memory and the algorithm proceeds with D_{valid} set to 1.

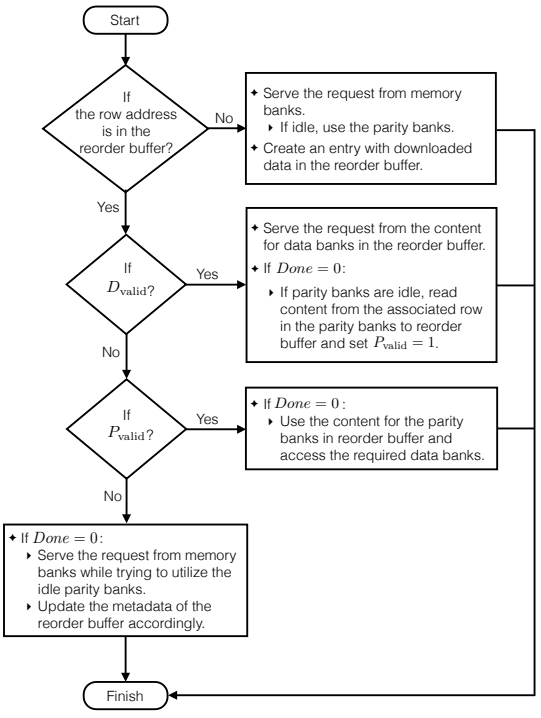


Figure 7: Description of the read algorithm.

3.2.3 Write algorithm

Note that each write request is accompanied by the content that needs to be updated in (some of) the data banks. However, in order to have consistency across all the memory banks, every update in the data banks should be reflected in the associated parity banks as well. The write algorithm ensures that the reorder buffer stores consistent and valid data for each row address available in the buffer. The write algorithm achieves this task by following the procedure described in Figure 8. In particular, the write algorithm picks the pending write access requests from the core in order. The algorithm then determines if the associated row address is present in the reorder buffer. If it is not present, it makes an entry for the row in the reorder buffer.

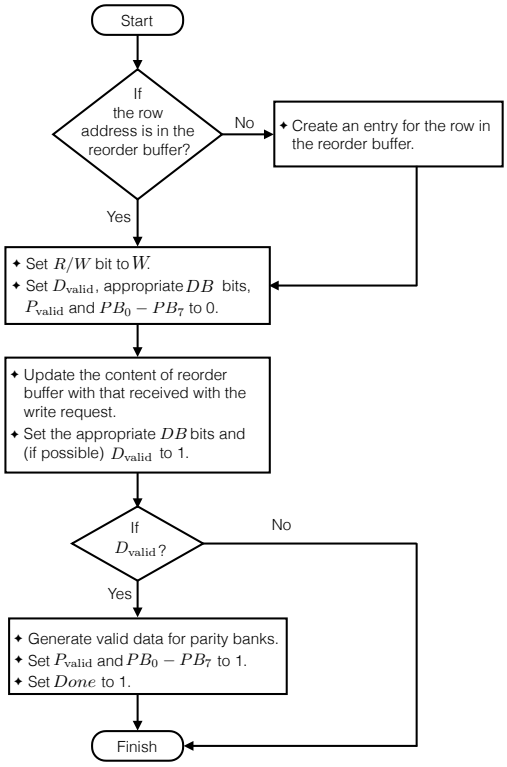


Figure 8: Description of the write algorithm.

Once the reorder buffer has the entry for the row address, the write algorithm ensures that the buffer stores the valid content for the row address. This involves 1) copying the content received with the write request to update the content already available in the buffer from the data banks; 2) regenerating the updated content for the parity banks; and 3) updating the metadata of reorder buffer by modifying the entry corresponding to the underlying row address. Once the reorder buffer has valid and consistent data for a given row address, *Done* is set to 1.

Table 1 illustrates these steps for an example write request. At $t = 3$, the write algorithm starts processing a write request for data bank $B0$ by setting the R/W bit with the associated address to W. It also sets the D_{valid}, DB_0, P_{valid} and $PB_0 - PB_7$ to zero. In the subsequent steps, the write algorithm updates the content for data bank $B0$ in the buffer by replacing it with the content received with the write request. It then updates DB_0 and D_{valid} by setting those to 1. If D_{valid} is 0, the reorder buffer entry remains in the buffer until all data banks are updated. Once D_{valid} is indeed 1, then the write algorithm generates the new content for the parity banks and sets P_{valid} and $PB_0 - PB_7$ to 1. At this point, it also sets *Done* to 1.

Once the *Done* bit is set to 1, the write algorithm schedules the updated row to be written to the memory banks. Once the write is successfully performed on the memory banks, the entry in the reorder buffer is updated by setting R/W bit to R and *Done* bit to 0. Here, we note that the R/W bit can get modified by a subsequent read request (as illustrated in Table 1). However, the $Done = 1$ bit reminds the scheduler that the memory banks have not yet been updated.

3.2.4 Writeback algorithm

The scheduler uses the limited size reorder buffer to fetch data from the bank rows and to perform read, write and code operations. In the process, the scheduler ensures that every time the reorder buffer reaches near its capacity, it clears up entries based on the following set of rules: Rule 1 is to pop out the oldest entry in the reorder buffer. Rule 2 is to ensure that the evicted row from the reorder buffer matches the data in the memory bank. This is ensured by using the *Done* bit in the reorder buffer. If a write instruction modifies a data bank entry, the scheduler reconstructs the code and makes $Done = 1$.

4. EXPERIMENTS

In this section, we describe our HBM implementation and Ramulator simulations on several SPEC2006 benchmarks.

4.1 HBM Implementation

Our coding scheme is based on a single 16-bank channel of HBM DRAM operating in Pseudo Channel Mode (8 banks per pseudo channel). To fit the layout of Figure 6 and Section 3, Pseudo Channel 0 is used for data banks and Pseudo Channel 1 is used for parity banks. Wherever possible, we try to interleave the banks. This ensures that most large, linear accesses will be spread across multiple banks with reduced contention.

In this mode, the 128-bit data bus is split into 2 individual 64-bit segments. However, the pseudo channels share the same address and command bus: commands and addresses may be sent to one pseudo channel or the other, but not to both. They also decode and execute commands individually. For commands that are common to both pseudo channels, strict timing conditions must be met to avoid conflicts. Table 2 describes additional design details.

Memory overhead	Storage of parity is limited to 50% of the overall memory.
Memory Banks	8 Data banks, 8 parity banks
Cache Line Size	128/256 bytes size
Element Size	Each element is 256 bytes
Number of Cores	6-8 cores for Wireless SoC platform
Access Rate	1GHz memory speed
Burst Length	4 (256-bit transfer in four 64-bit bursts)

Table 2: Summary table of design parameters.

REMARK 6. *Figure 9 describes the address mapping for each channel. The least significant "OFFSET" bits of the address signify the byte level addressing of the data. The* 6 *most significant "CA" bits indicate column address, the next* 14 *"RA" bits indicate row address, the* 3 *"BA" bits decide the bank, and the remaining* 3 *"CH" bits specify the channel.*

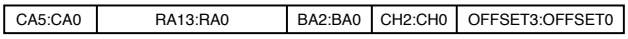


Figure 9: Address mapping for channels.

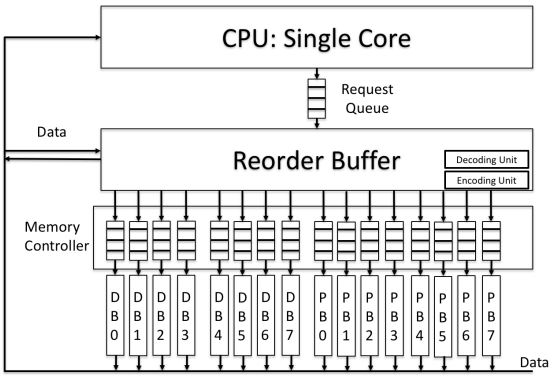


Figure 10: Single-core CPU Simulation.

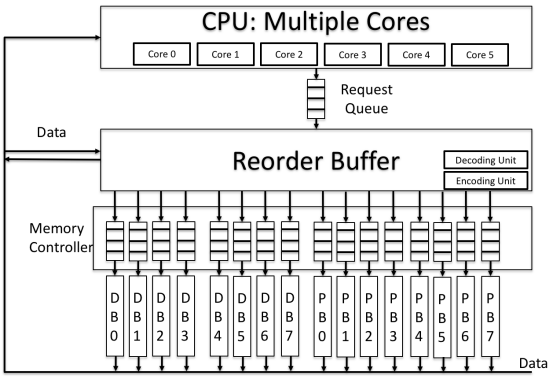


Figure 11: Multi-core CPU Simulation.

4.2 Methodology

Our code design was implemented on the Ramulator platform [**?**] to evaluate memory access improvements compared to a baseline, uncoded memory layout. This platform allows for excitation of the memory scheduler module with standard benchmarks that represent differentiated and distributed computational and memory access patterns. Experiments were conducted on a single-processor system using the SPEC2006 benchmark [**?**], and on a 6-core and 8-core systems using application traces from an off-the-shelf wireless SoC.

The single-core architecture for the SPEC2006 benchmark is shown in Figure 10, and Figure 11 shows the architecture for the 6-core processor simulation. The instruction traces are mapped to memory requests, which are pushed to a request queue and sent to the memory system's reorder buffer. For each request, the corresponding cycle number, read or write access, and address is stored. After the steps outlined in Section 3.2.1, the bank requests are generated and sent to the HBM memory controller. The corresponding data is sent back to the reorder buffer or to the CPU via the data bus. Since the simulator is cycle accurate, the 6-core processor simulation may pop multiple requests from the reset queue at the beginning of each cycle.

4.3 Cycle Simulation Results

Next, we present the results of SPEC2006 simulation cycles on the proposed coded model and a baseline model. The goal is to simulate the benefits of improved read accesses, which bounds the amount of improvement we can expect in a full hardware implementation. Encoding and decoding can be done offline in parallel and is therefore ignored. We also consider the delays of parity bank writebacks to be negligible. The coded model was evaluated with the following reorder buffer sizes: $\{8, 16, 32, 64, 128, \infty\}$. For larger reorder buffer size, the increased memory overhead leads faster accesses. Thus, the infinite reorder buffer simulation further bounds the performance improvements we expect in practice because no rows are evicted.

Figure 15 compares our memory scheme's performance improvements across different benchmarks. For cases such as `omnetpp` and `sjeng`, our scheme improves over the uncoded baseline and continues to improve as the buffer size increases. We also see that for the `mcf` benchmark, our scheme reduces the read request latency and also reduces the number of CPU cycles to approximately 30% of the uncoded case. This benchmark has the most significant improvements because most of its instructions are memory read requests. However, `mcf` hits a performance bottleneck and does not continue improving when the reorder buffer size exceeds 64. This suggests that the `mcf` memory pattern is clustered such that all the memory requests can be stored in a reorder buffer of this size. The `mcf` benchmark is made up of a program that implements network simplex code that often works on large chunk of data intermittently [**?**]. The speed of access (subsequently the speed of the program) in this case improves significantly with the coded memory access.

The `mlic` benchmark shows reduction to about 40% of the uncoded case for any buffer size. This variety of results suggests that additional improvements can be achieved by combining our scheme with other approaches.

For the application traces in Figure 16, the performance improvements are modest compared to those of the SPEC2006 benchmark. Our scheme hits a performance bottleneck with a reorder buffer size of only 16 bits. We attribute this to the fact that memory patterns for the application traces are clustered more closely. Again, additional techniques must be used to exceed this observed performance barrier.

4.4 Cycle Simulation Results

Next, we present the results of SPEC2006 simulation cy-

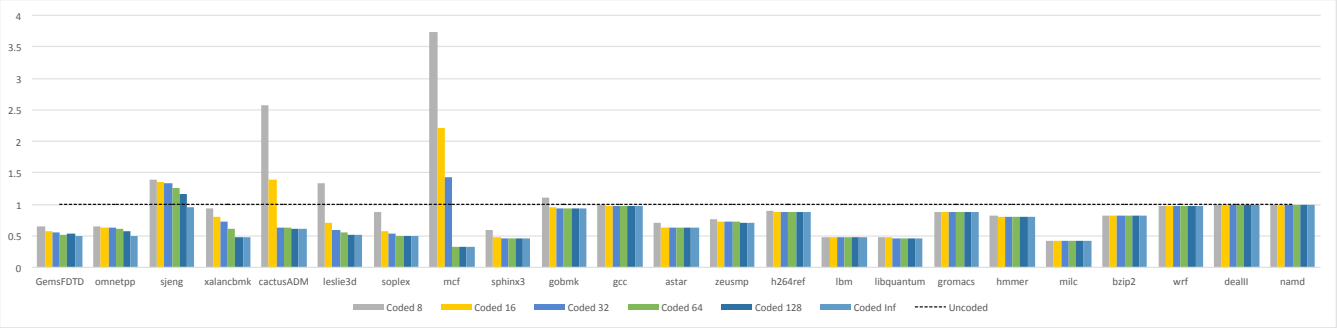


Figure 12: Simulated Number of CPU cycles: Baseline HBM versus Coded HBM across different benchmarks (with reorder buffer size of $\{8, 16, 32, 64, 128, \infty\}$).

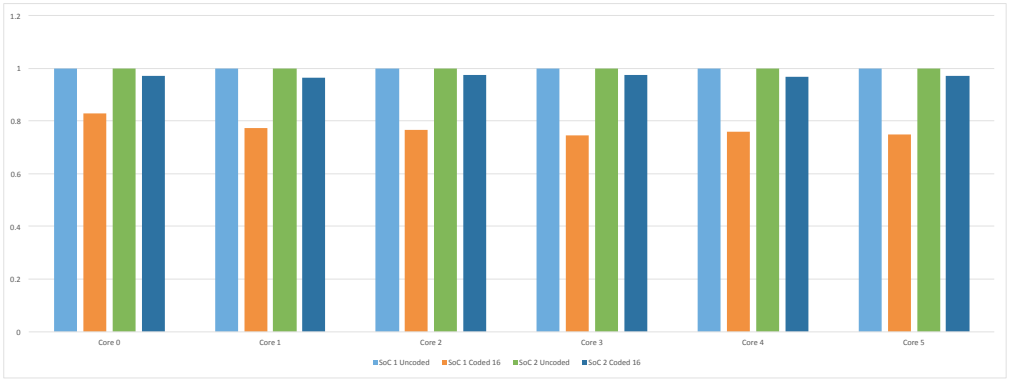


Figure 13: Simulated Number of active DRAM cycles: Baseline HBM versus Coded HBM across application-driven wireless SoC traces (reorder buffer size 16)

cles on the proposed coded model and a baseline model. The goal is to simulate the benefits of improved read accesses, which bounds the amount of improvement we can expect in a full hardware implementation. Encoding and decoding can be done offline in parallel and is therefore ignored. We also consider the delays of parity bank writebacks to be negligible. The coded model was evaluated with the following reorder buffer sizes: $\{8, 16, 32, 64, 128, \infty\}$. For larger reorder buffer size, the increased memory overhead leads faster accesses. Thus, the infinite reorder buffer simulation further bounds the performance improvements we expect in practice because no rows are evicted.

Figure 15 compares our memory scheme's performance improvements across different benchmarks. For cases such as `omnetpp` and `sjeng`, our scheme improves over the uncoded baseline and continues to improve as the buffer size increases. We also see that for the `mcf` benchmark, our scheme reduces the read request latency and also reduces the number of CPU cycles to approximately 30% of the uncoded case. This benchmark has the most significant improvements because most of its instructions are memory read requests. However, `mcf` hits a performance bottleneck and does not continue improving when the reorder buffer size exceeds 64. This suggests that the `mcf` memory pattern is clustered such that all the memory requests can be stored in a reorder buffer of this size. The `mcf` benchmark is made up of program that implements network simplex code that often works on large chunk of data intermittently[**?**]. The speed of access (subsequently the speed of the program) in this case improves significantly with the coded memory access.

The `mlic` benchmark shows reduction to about 40% of the uncoded case for any buffer size. This variety of results suggests that additional improvements can be achieved by combining our scheme with other approaches.

For the application traces in Figure 16, the performance improvements are modest compared to those of the SPEC2006 benchmark. Our scheme hits a performance bottleneck with a reorder buffer size of only 16 bits. We attribute this to the fact that memory patterns for the application traces are clustered more closely. Again, additional techniques must be used to exceed this observed performance barrier.

5. CONCLUSION

This paper proposes a coded memory system that is amenable to multi-bank DRAM architectures such as HBM. Our Reed-Solomon based coding scheme reduces bank conflicts by creating many redundant ways to access a requested data element. This is achieved by assigning some banks to be dedicated parity banks. A memory scheduler then decides which frequently-used data elements should be stored in these parity banks. Simulations on the SPEC2006 benchmarks show significant reduction in CPU latency when the scheduler's reorder buffer has both finite and infinite size.

One potential source of improvement is the addition of a

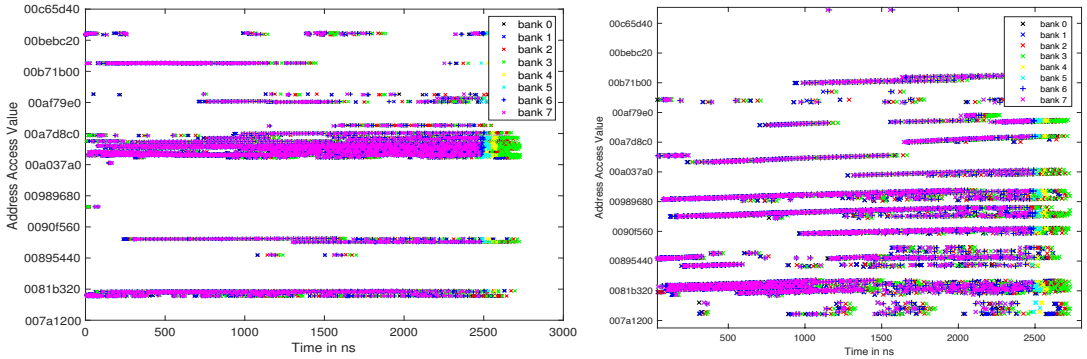


Figure 14: Memory bank accesses across a 3 μs trace, which suggest linear access patterns. The right figure is an enlarged version of the top half of the left figure.

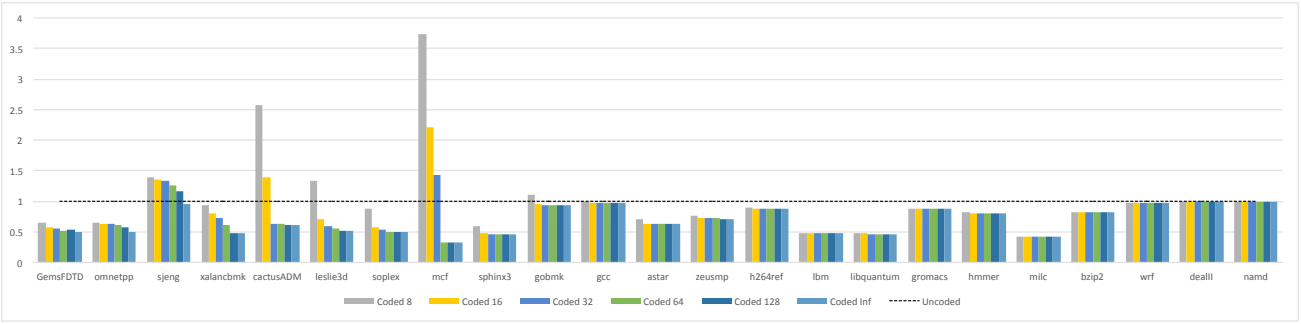


Figure 15: Simulated Number of CPU cycles: Baseline HBM versus Coded HBM across different benchmarks (with reorder buffer size of $\{8, 16, 32, 64, 128, \infty\}$).

memory prefetching unit, similar to an instruction or cache prefetching unit, which can detect linear access patterns to memory regions. For example, if a sequence of memory accesses is issued in increasing order spaced one byte apart, then a prefetching unit would predict the next access to be one byte past the previous one. Prefetching-based memory designs have been studied only in the context of *uncoded* memory systems [**?**, **?**, **?**, **?**]. We can augment our scheme by fetching a predicted address from a parity bank during accesses for which it remains valid but idle. Then for future memory accesses, the controller can check the prefetched data and attempt to complete the request using current accesses and prefetched data. This means that previously occupied banks are available to serve more accesses per cycle. As memory accesses wait to be issued in the bank queues, they can simultaneously be checked with prefetched data. Thus, no extra latency is anticipated by the addition of a memory prefetching unit. Figure 17 shows two plots of memory accesses for several banks across time. Both figures suggest linear access patterns and thus larger performance improvements when coding caching is combined with prefetching for this application.

6. HBM DETAILS: REMOVED FROM BACKGROUND SECTIONS

Each HBM channel is expected to provide a 128-bit data interface that supports up to 1-2 Gbps of data rate per signal equivalent to 500-1000MHz of DDR memory. This is equivalent to 16-32 GB/sec of bandwidth per channel. With 8 channels stacked together it can provide 128-256 GB/sec of access bandwidth per stack.

The HBM specification document lists two modes of operation: legacy mode and pseudo channel mode. A memory controller can support either the legacy or the pseudo channel mode.

In case of the legacy mode, each read or write transaction transfers 256 bits in a burst that consists of 2 cycles of 128 bits each. In pseudo channel mode, the 128-bit bus is split into 2 individual 64-bit segments. On each segment, a read or write transaction transfers 256 bits as well, but in a burst that lasts 4 cycles (of 64 bits each). The pseudo channel conceptually divides the memory of a single channel in half and assigns each half to a fixed pseudo channel. The pseudo channels share the same address and command bus: you can send a command and addresses to one pseudo channel or the other, but not to both.

7. ACKNOWLEDGEMENTS

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

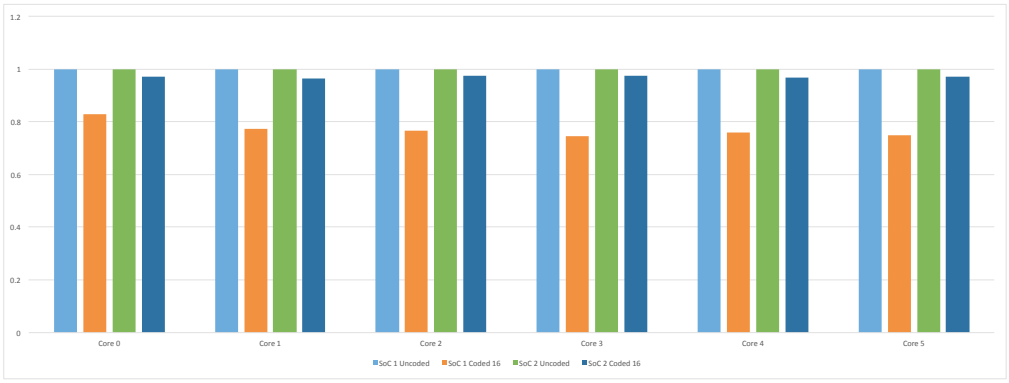


Figure 16: Simulated Number of active DRAM cycles: Baseline HBM versus Coded HBM across application-driven Wireless SoC traces (reorder buffer size 16)

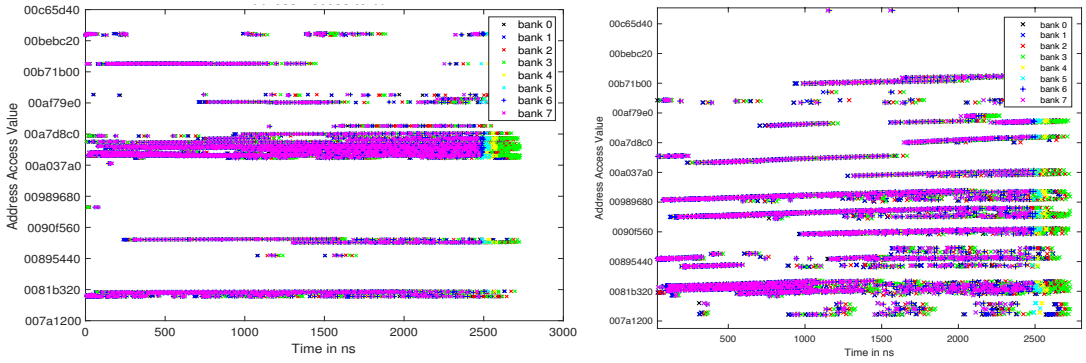


Figure 17: Memory bank accesses across a 3 μs trace, which suggest linear access patterns. The right figure is an enlarged version of the top half of the left figure.