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 data across multiple single-port memory banks in order to *algorithmically* realize the functionality of multi-port memory.

This paper proposes three code designs with significant less storage overhead compared to the existing replication based emulations of multi-port memories. To further improve performance, we also demonstrate a memory controller design that utilizes redundancy across coded memory banks to more 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*. We then show significant performance improvements in critical word read and write latency in the proposed coded-memory design when compared to a traditional uncoded-memory design. **[Ethan: real-life, PARSEC, or both?]**

1. INTRODUCTION

Memory accesses are one of the most vital parts of any computer 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. Even with the saturation/demise of Moore's law [1, 2, 3], processing power is expected to grow with the increased reliance on multi-core architectures [4]. Since the end-to-end performance of a program heavily depends on both processor and memory performance, slower memory becomes a bottleneck and slows down the whole system. This has been driving motivation for computer architects and researchers 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.

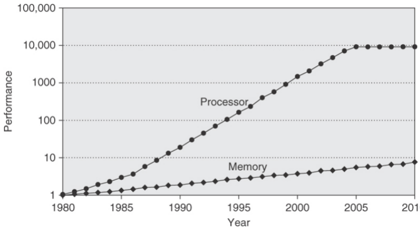


Figure 1: *The gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over a* 30 *year span [6].*

In a multi-core processor architecture, contention between various cores exacerbates access latency to the shared memory, resulting in large access request queues. Figure 2 illustrates a general multi-core architecture. The 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 memory locations on the same bank, the memory controller arbitrates them using bank queues. This contention between cores to access from the same bank is known as a *bank conflict*. As the number of bank conflicts increases, memory access latency also increases and the entire system slows down.

We address the issue of increased latency due to bank conflicts by introducing a coded memory design. The main principle behind our memory design is to distribute the accesses intended for the information stored in a particular bank across multiple banks in the memory by using coding theoretic techniques to create redundancy. We store the information in the memory banks such 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

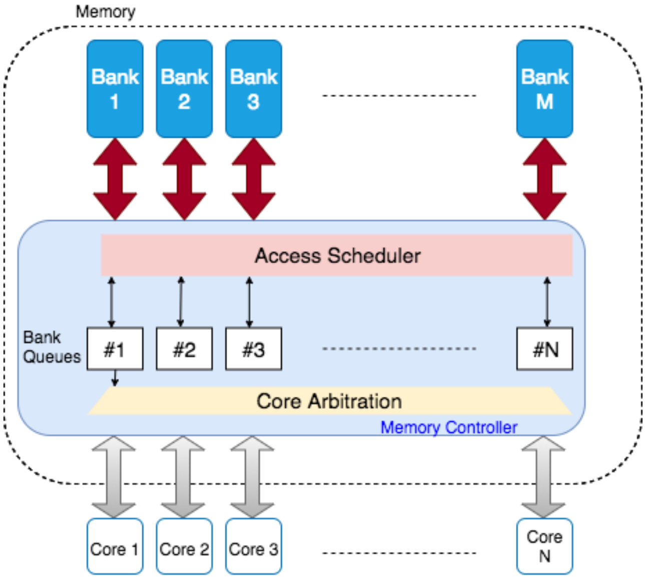


Figure 2: *General multi-core architecture with a shared memory. N processor cores share a memory consisting of M banks.*

the system. Figure 3 shows this with an example. Here, Bank 3 is redundant as its content is a function of the content stored on Banks 1 and 2. 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. Let $+$ denote the XOR operation, and additionally assume that the memory controller is capable of performing simple decoding operations, *i.e.* recovering $a(j)$ from $b(j)$ and $a(j) + b(j)$. Because the third bank store L rows containing $[a(1) + b(1), \ldots, a(L) + b(L)]$, this design allows us to simultaneously serve any two read requests in a single memory clock cycle.

The memory design in Figure 3 also 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 3) Arbitration/scheduling of accesses across multiple clock cycles. 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 $a(j)$, 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 $2/3$. 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 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.

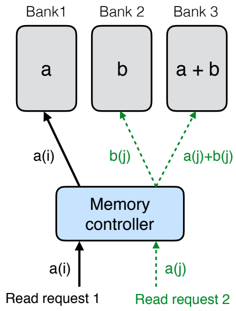


Figure 3: *Enabling multiple read accesses to a bank by coding. Given two read requests $\{a(i), a(j)\}$ directed to Bank 1, we can deal with bank conflict in the following manner: 1) First request for $a(i)$ can be directly served by Bank 1 itself, and 2) The read request for $a(j)$ can be served by 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.*

Main contributions and organization: In this paper we systematically address all key issues pertaining to a shared memory system that can simultaneously service 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 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. This includes careful utilization of the redundancy present in the memory banks while maintaining the validity of information stored in them.

[1] The information rate is a standard measure of redundancy of a coding scheme ranging from 0 to 1, where 1 corresponds to the most efficient utilization of storage space.

- Focusing on applications where memory traces might exhibit favorable access patterns, we explore two ways to improve the efficiency of our coding based memory design in Sections 4.5 and 4.6. First, we propose a dynamic coding scheme which is based on continuous detection of heavily accessed regions on memory banks. The second solution involves predicting the patterns of memory addresses in different access requests.

- Finally, we conduct a detailed evaluation of the proposed designs of shared memory systems in Section 5.3. 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 for I/O, 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]. Attempts have been made to emulate multi-port memory by [19], however they use replication based design that makes the resulting architecture very large in memory. **[Ethan: Move some of this earlier?]**

Due to space limitations we focus on specific, illustrative request patterns. We invite the reader to verify that our designs indeed handle the set of all possible requests.

2.1.1 *Read-only Support*

Replication-based designs are the most prevalent candidates in this design space. Assuming that a memory design is required to support only read requests, say r read requests 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 (see Figure 4). This way, the r-replication-based design completely avoids bank conflicts for up to r read request in a memory clock cycle.

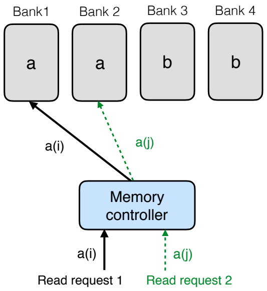


Figure 4: 2-replication based design to support multiple 2 read requests in the same memory clock cycle. The two banks' worth of data $\mathbf{a} = [a(1), \dots, a(L)]$ and $\mathbf{b} = [b(1), \dots, b(L)]$, all the data elements are stored on two distinct memory banks. Note that any 2 read requests 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.

Remark 1. *If we compare the memory design in Figure 4 with that of 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 3, the access process involves some computation. 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 comes at the expense of increased computation (XOR decoding). Therefore, it is important to employ those coding schemes that enable storage efficiency with as small computational overhead as possible.*

2.1.2 *Read and Write Support*

[ANKIT: Mainly describing the results from the work of Auerbach, Chen, and Paul[14].] 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 of (potentially multiple) write requests as well. A challenge that arises in the presence of write requests is that one also need to ensure 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 stored 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 as-

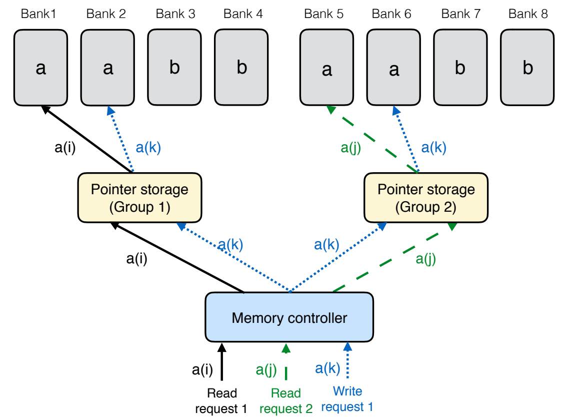


Figure 5: 4-replication based design to support $r = 2$ read requests and $w = 1$ write request in one memory clock cycle. Both collections of information elements $\mathbf{a} = [a(1), \ldots, a(L)]$ and $\mathbf{b} = [b(1), \ldots, b(L)]$ are replicated on $r \cdot (w+1) = 4$ different single-port memory 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.

sume 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 4 copies each of data elements $\mathbf{a}$ and $\mathbf{b}$ and partition the 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. **[Ethan: condense the beginning of this paragraph, and move the end of this paragraph later since it's not really background]** 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 replicate 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*

[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.

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. ***[Ethan: rewrite]***

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). **[Ethan: remove?]**

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. **[Ethan: remove above this line, put below this line somewhere earlier?]** 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.**[Ethan: good point!]** 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 replicate the obtained memory bank arrangement multiple times in order to support write requests as well. We instead 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**[Ethan: with? also rewrite this paragraph]** 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 effect on the overall performance (latency) into account.**[Ethan: good point]** 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 (see [29, 30, 31, 32] and references therein). More recently, an LSTM-based recurrent neural network was used to predict future memory access requests on the SPEC 2006 benchmark dataset [33]. This deep learning method may be used in addition to our proposed frequency-based approach. Our combination of coded memory and prefetching also shares some similarity with the recent line of work on coded caching [34] 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 [34]. 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 unit. In contrast, the setup of coded caching has a centralized storage system (server) and cache units that store encoded information 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 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 coding schemes which are designed with access efficiency as a key criterion. We first define some basic concepts with an illustrative example, and then describe our 3 schemes in detail.

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. **[Ethan: rewrite?]** 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. *Consider a setup with two data banks* $\mathbf{a}$ *and* $\mathbf{b}$*. We assume that each of banks store* $L\cdot W$ *binary data elements*[3] *which are arranged 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* $\mathbf{a}$ *and bank* $\mathbf{b}$*, respectively. Moreover, for* $i \in [L]$ *and* $j \in [W] \triangleq \{1,\ldots,W\}$*, we*

[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.

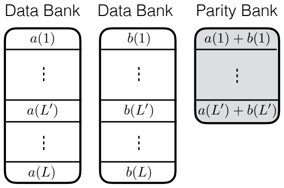


Figure 6: Notation of example parity design. ok?

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 $i \in [L'] \triangleq \{1, \ldots, L'\}$,

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

Remark 3. *Figure 6 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

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

Remark 4. *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 is reasonable assume the existence of shallow banks, especially in proprietary designs of integrated memories in a system on a chip (SoC).*

Remark 5. *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 4.5 we discuss a dynamic coding approach that changes 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.* ***[Ethan: rewrite, esp. last sentence]***

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 improve the **read** accesses to the original data elements by avoiding bank conflicts. Let's consider the coding scheme illustrated in Figure 6, where the parity bank stores $\mathbf{p} = \mathbf{a}([L']) + \mathbf{b}([L'])$. 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 it. In this case, the degraded read for $a(1)$ using $\mathbf{b}$ and $\mathbf{p}$ has locality 2.

3.2 Codes to emulate multi-port memory

We will now describe the code designs proposed in this work to emulate 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 strike a good balance among various quantitative parameters, 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 (the design scales with larger number of banks). This contrasts with the communications applications where encoding typically occurs with blocks of 1024 or more information symbols.

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

Design	Max reads (per bank)	Locality	Rate ($\alpha = 1$)	Storage overhead	Logical complexity
I	4	2	2/5	1.5α	Low
II	5	2	2/7?	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

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 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 regions. 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

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

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 access cycle. Consider the scenario when we receive accesses to the following 10 rows:

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

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

$$\{a(1), a(2), b(8), b(9), c(10), c(11), d(14), d(15)\}.$$

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, we still benefit from the prefetching mechanism discussed in Section 4.6. The worst case number of reads per cycle is equal to the number of data banks.

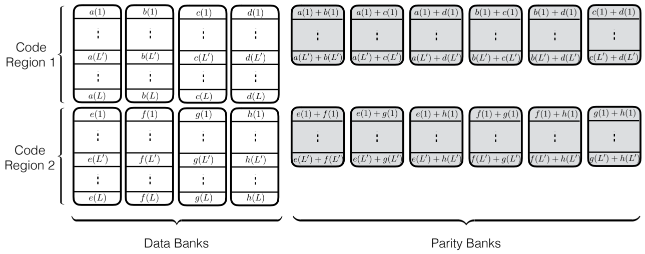


Figure 7: Code Design I

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. **[Ethan: NO!]** 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 requests that can be served in the best and worst case.

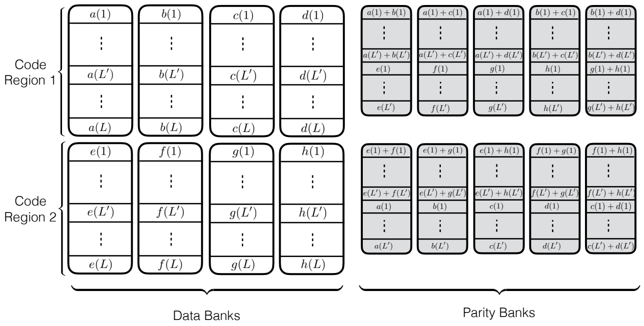


Figure 8: Code Design II

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 $\{a(1), b(1), c(1), d($ 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 $\{a(2), b(2), d($ 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 $\{a(1), b(6), c(9), d(15), e(20)\}$, 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 4.6) 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 **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 $a(4) + e(4) + z(4)$.
Best case analysis: Following the analysis similar to design I and II, the best case number of reads per cycle will be equal to sum of data bank and parity bank (18 reads per cycle in case of design III).
Worst case analysis: Similar to design I and design II, the number of reads per cycle is equal to the number of data banks (9 in case of design III).

Remark 6. *Note that the coding scheme in Figure 9 describes a system with 9 data banks. However, we have 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 **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.*

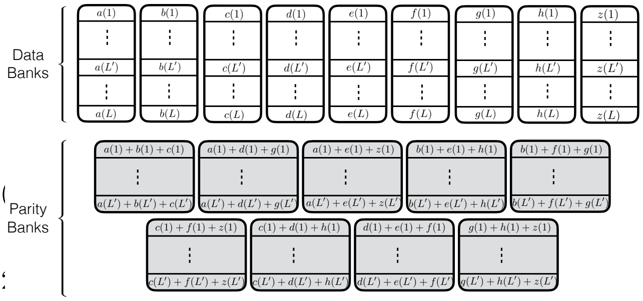


Figure 9: Code Design III

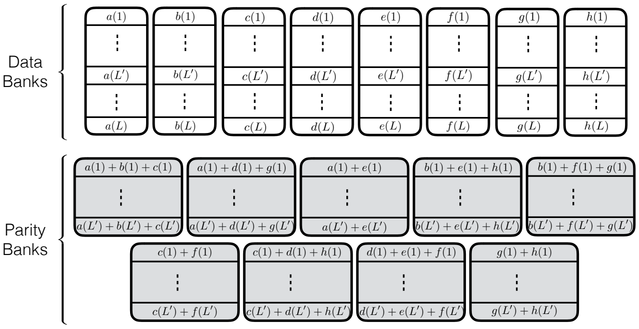


Figure 10: Code Design III with 8 data banks

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

4. MEMORY CONTROLLER DESIGN

The architecture of the memory controller is focused on exploiting redundant storage in the coding schemes to serve memory requests faster than an uncoded scheme. The following three stages are illustrated in Figure 2:

- **Core arbiter:** Every clock cycle, the *core arbiter* receives up to one request from each core which it stores in an internal queue. The core arbiter attempts to push these request to the appropriate bank queue. If it detects that the destination bank queue is full, the controller signals that the core is busy which stalls

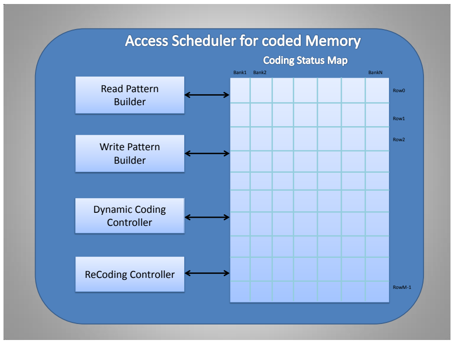


Figure 11: Access scheduler for coded memory

the core. The core arbiter also arranges the requests stored in its internal queue using a two-step priority order mechanism: first by Quality of Service (QoS) and then breaking ties with round-robin scheduling.

- **Bank queues:** Each data bank has a corresponding *read queue* and *write queue*. The core arbiter sends memory requests to the bank queues until the queues are full. In our simulations, we use a bank queue depth of 10. There is also an additional queue which holds special requests such as memory refresh requests.
- **Access scheduler:** Every memory cycle, the *access scheduler* chooses to serve read requests or write requests, algorithmically determining which requests in the bank queues it will schedule. The scheduling algorithms the access scheduler uses are called pattern builders. Depending on the current memory cycle's request type, the access scheduler invokes either the read or write pattern builder. A key design tradeoff of the pattern builder algorithms is the relationship between the complexity of the algorithm and the number of requests the algorithm schedules.

We note that the core arbiter and bank queues should not differ much from those in a traditional uncoded memory controller. The access scheduler directly interacts with the memory banks, and therefore must be designed specifically for our coding schemes.

4.1 Code Status Table

The code status table keeps track of the validity of data stored the data and parity banks. When a write is served to a row in a data bank, any parity bank which is constructed from the data bank will contain invalid data its corresponding row. Similarly, when the access scheduler serves a write to a parity bank, both the data bank which contains the memory address specified by the write request and any parity banks which utilizes that data bank will contain invalid data. The code status table keeps track of the locations of invalid data so the access scheduler does not erroneously serve read requests with stale data.

Figure 11 depicts our implementation of the code status table. It contains an entry for every row in each data bank, which can take one of three values indicating 1) the data in both the data bank and parity banks is fresh, 2) the data bank contains the most recent data, or 3) one of the parity banks contains the most recent data. We assume that the elements of the code status table are accessible at a very fast rate.

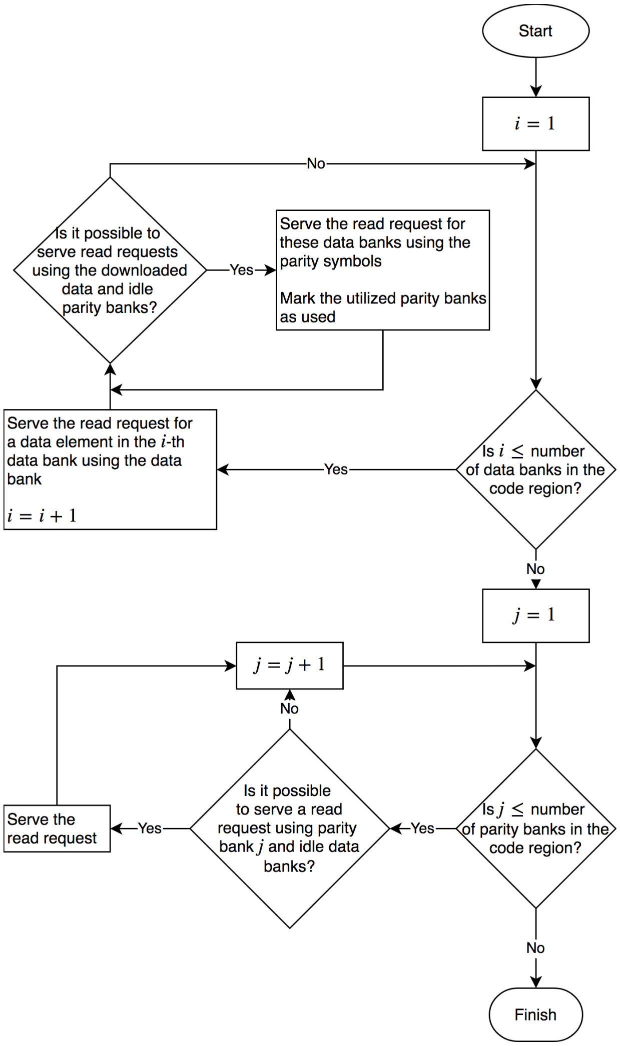

Start
$i = 1$
No
Is it possible to serve read requests using the downloaded data and idle parity banks?
Yes
Serve the read request for these data banks using the parity symbols
Mark the utilized parity banks as used
Serve the read request for a data element in the i-th data bank using the data bank
$i = i + 1$
Yes
Is $i \leq$ number of data banks in the code region?
No
$j = 1$
$j = j + 1$
No
Serve the read request
Yes
Is it possible to serve a read request using parity bank j and idle data banks?
Yes
Is $j \leq$ number of parity banks in the code region?
No
Finish

Figure 12: Description of the algorithm to build a read request pattern to be served in a given memory cycle.

4.2 Read pattern builder

The access scheduler uses the read pattern builder algorithm to determine which requests to serve using parity and which to use data banks. The read pattern builder selects which memory requests to serve and determines how requests served by parity banks will be decoded. The algorithm is designed to serve many read requests in a single memory cycle. Figure 12 shows our implementation of the read pattern builder.

Figure 13 shows an example read pattern constructed by our algorithm. First, $a(1)$ is marked to be read from data bank **a**. Then the algorithm looks through banks **b**, **c**, and **d** for requests for rows $b(1)$, $c(1)$, or $d(1)$ because these symbols can be decoded from a parity bank

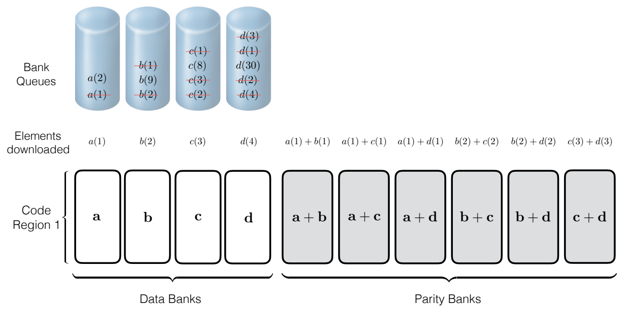


Figure 13: Illustration of the algorithm to build a read request pattern to be served in a given memory cycle. All the read requests associated with the strikethrough elements are scheduled to be served in a given memory cycle. The figure also shows the elements downloaded from all the memory banks in order to serve these read requests.

using the $a(1)$ symbol. In this scenario all three are present in the bank queues and are served using parity banks. Symbols equal to $a(1) + b(1)$, $a(1) + c(1)$, and $a(1) + d(1)$ are all downloaded from parity banks and decoded with $a(1)$. Next, $b(2)$ is read from a data bank. Similar to before, $c(2)$ and $d(2)$ are served by downloading $b(2) + c(2)$ and $b(2) + d(2)$ symbols from the parity banks. Again as before, $c(3)$ is read from data bank and $d(3)$ is decoded using $c(3)$ and $c(3) + d(3)$. Finally, $d(4)$ is read from a data bank. In this scenario, Only the top loop of the read pattern builder as pictured in Figure 12 schedules reads, but there are scenarios where the bottom loop is useful. **[Ethan: either construct an example which illustrates everything or don't draw attention to the fact that it doesn't illustrate the bottom loop] [MATT: I have an optimal algorithm for scheduling read requests. Should I include it in this section?] [Ethan: what's different about the optimal algorithm, optimal in what sense, and why aren't we using that one instead?!]**

Remark 7. *By increasing the number of read requests per cycle, we increase the risk of having out-of-order execution of memory access requests on the same core. We assume that the code arbiter only admits requests into the bank queues if they can be immediately served without introducing harmful out-of-order execution.*

4.3 Write pattern builder

Parity banks allow the memory controller to serve additional write requests per cycle. When multiple writes target a single bank, it can commit some of them to parity banks. The access scheduler implements a write pattern builder algorithm to determine which write requests to schedule in a single memory cycle. Figure 14 illustrates our implementation of the write pattern builder. Only when the write bank queues are nearly full does the access scheduler execute the write pattern builder algorithm.

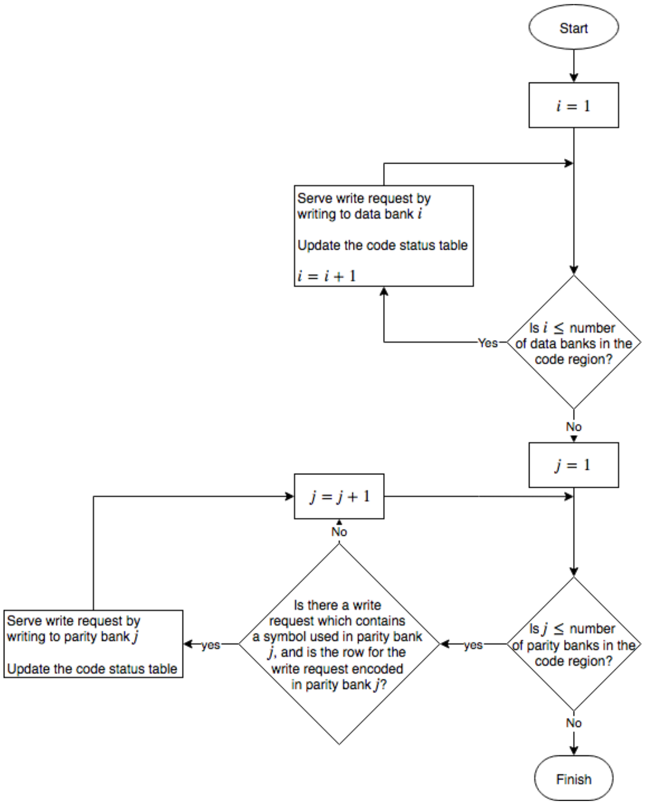

Start
$i = 1$
Serve write request by writing to data bank i
Update the code status table
$i = i + 1$
Yes
Is $i \leq$ number of data banks in the code region?
No
$j = 1$
$j = j + 1$
No
Serve write request by writing to parity bank j
Update the code status table
yes
Is there a write request which contains a symbol used in parity bank j, and is the row for the write request encoded in parity bank j?
yes
Is $j \leq$ number of parity banks in the code region?
No
Finish

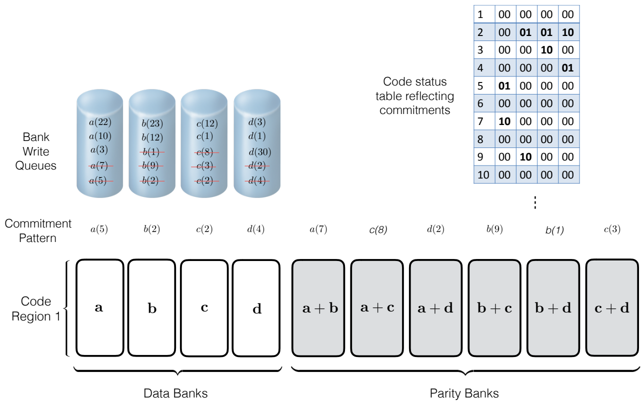
Figure 14: Flowchart of write pattern builder

Figure 15 shows an example write pattern produced by our algorithm. Parity banks increase the maximum number of write requests from 4 to 10. Note that an element which is addressed to row n in a data bank can only be written to the corresponding row n in the parity banks. In this scenario, the write queues for every data bank are full. The controller takes 2 write requests from each queue and schedules one to the queue's target data bank and the other to a parity bank. The controller also updates the code status table. **[MATT: The figure figure here contains an error - 10 write requests should be served] [Ethan: still error?]**

Figure 15 also demonstrates how the code status table changes to reflect the freshness of the elements in the data and parity banks. Here, the 00 status indicates that all elements are updated. The 01 status indicates that the data banks contain fresh elements and the elements in the parity banks must be recoded. The 10 status indicates that the parity banks contain fresh elements, and that both data banks and parity banks must be updated.

4.4 ReCoding unit

After a write request has been served, the stale data in the parity (or data) banks must be replaced. The *Re-*



1	00	00	00	00
2	00	**01**	**01**	**10**
3	00	00	**10**	00
4	00	00	00	**01**
5	**01**	00	00	00
6	00	00	00	00
7	**10**	00	00	00
8	00	00	00	00
9	00	**10**	00	00
10	00	00	00	00

Figure 15: Figure describing write algorithm access pattern

Coding Unit accomplishes this with a queue of *recoding requests*. Every time a write is served, recoding requests are pushed on to the queue indicating which data and parity banks contain stale elements, as well as the bank which generated the recoding request. Requests also contain the current cycle number so that the ReCoding Unit may prioritize older requests. Appendix

4.5 Dynamic Coding

To reduce memory overhead α, parity banks are designed to be smaller than data banks. The dynamic coding block maintains codes for the most heavily accessed memory subregions, so that parity banks are utilized more often. The *dynamic coding* block partitions each memory bank into $\lceil \frac{1}{r} \rceil$ regions. The block can select up to $\frac{\alpha}{r} - 1$ regions to be encoded in the parity banks. A single region is reserved to allow encoding of a new region.

Every T cycles, the dynamic coding unit chooses the $\frac{\alpha}{r} - 1$ regions with the greatest number of memory accesses. The dynamic coding unit will then encode these regions in the parity banks. If all the selected regions are already encoded, the unit does nothing. Otherwise, the unit begins encoding the most accessed region. Once the dynamic coding unit is finished encoding a new region, the region becomes available for use by the rest of the memory controller. If the memory ceiling $\alpha - r$ is reached when a new memory region is encoded, the unit evicts the least frequently used encoded region. **[Ethan: why $\alpha - r$?]**

4.6 Prefetching Codes

Dynamic coding works best when the most heavily accessed regions of memory vary little to none over time. When the memory access trend is not static, the proposed memory system can benefit from anticipating structured (*e.g.* sequential) memory accesses. Therefore, our design includes a prefetcher which detects sequential memory accesses and exploits idle memory banks to potentially serve future memory requests. The prefetcher prioritizes long sequential memory accesses as motivation for performing an anticipatory read. Because of the speculative nature of the prefetcher, it is given the lowest priority of all the components in the access scheduler. It only schedules a memory access if all the other units have not done so first. **[Ethan: More details here?]**

5. EXPERIMENTS

In this section, we discuss our method for evaluating the performance of the proposed memory system. We utilize the PARSEC v2.1 and v3.0 benchmark suites with the gem5 simulator [35, 36] to generate memory traces. Next, we run the Ramulator DRAM simulator to measure the performance of the proposed memory system. Next, we compare the baseline performance of the Ramulator DRAM simulator [11] against a modified version which implements the proposed system.

5.1 Memory Trace Generation

The PARSEC benchmark suite was developed for chip multiprocessors and is composed of a diverse set of multithreaded applications [35]. These benchmarks allow us to observe how the proposed memory system performs in dense memory access scenarios. The gem5 simulator [36] allows us to select the processor configuration used when generating the memory traces. We use 8 processors in all simulations. The PARSEC traces can be split into regions based on computation type. We focus on regions which feature parallel processing they have the greatest likelihood of bank conflicts.

Many attributes affect the performance of our proposed memory system, most importantly the density of traces, the overlap of memory accesses among processors, and the stationarity of heavily utilized memory regions. Appendix B shows example PARSEC traces with favorable attributes, *i.e.* they are likely to result in many preventable bank conflicts.

[Ethan: move Augmented PARSEC here] We augment the PARSEC benchmarks in two ways to test our system in additional scenarios, shown in Figures 16 and 17, respectively. First we split the given memory bands to simulate an increased number of bands in the system. Next, we introduce dynamic memory access patterns by adding a linear ramp to the previously static address locations.

5.2 Ramulator

We use the Ramulator DRAM simulator to compare the number of CPU cycles required to execute the PARSEC memory traces. First, we use the original Ramulator simulator to measure a baseline number of CPU cycles. Then we implement the proposed memory system and use the modified Ramulator (fixing all other configuration) to calculate improvements over the baseline. Our simulations vary the overhead parameter α.

5.3 Simulation Results

Given sufficient memory overhead, we see a consistent

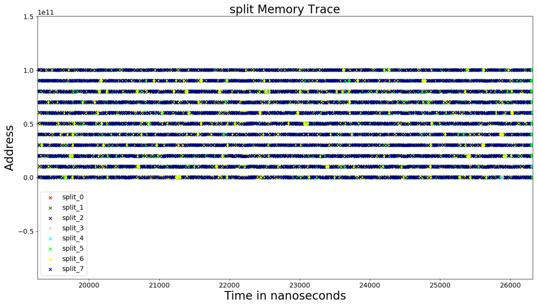


Figure 16: The vips benchmark after splitting the primary access bands into multiple additional bands.

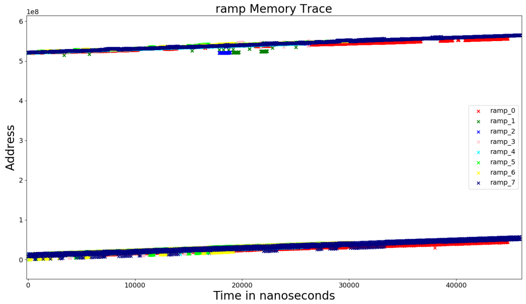


Figure 17: The vips benchmark after adding a ramp to the major memory bands.

25% reduction in CPU cycles over the baseline simulation, with Coding Scheme I generally performing best.

The proposed memory system performs consistently across the PARSEC benchmarks, and the three proposed schemes yield similar results. Figure 18 shows the simulation results for the dedup benchmark with a memory partition coefficient $r = 0.05$. The plot shows that the number of CPU cycles is reduced by $73\% - -83\%$ once sufficient memory overhead α is used. We also see that the number of memory region switches performed by the dynamic encoder. When $\alpha = 1$, the number of switches is always zero as expected because the dynamic encoder never needs to switch regions. The performance remains consistent for $\alpha > 0.1$. With this amount of overhead, the memory system finds and encodes the two heavily accessed memory bands in each of the PARSEC benchmarks. This is because $\lfloor \frac{\alpha}{r} \rfloor = 2$, which means we can select 2 regions to encode. When $\alpha = 0.05$, the number of coded region switches is very high because the memory system vacillates between the two most heavily accessed bands. When $\alpha = .1$, both of them can be encoded. We see a small numbers of switches when $\alpha \geq 0.25$ because the memory system is encoding less heavily accesses memory bands with little impact on number of CPU cycles.

The heavily accessed memory bands are narrow, which suggests that decreasing the memory bank partition will result in similar performance improvements with a lower α. Figure 19 shows that indeed α can be reduced by a factor of 5 by also decreasing r from 0.05 to 0.01.

5.3.1 *Augmented PARSEC*

Results on the the augmented PARSEC traces show that our system improves over the baseline to a lesser extent. Figure 20 shows that for a large number of memory bands, we can achieve the same performance as before only by increasing the memory overhead or increasing the memory partition coefficient.

Figure 21 shows the results of the ramp augmentation. Here we see that our system struggles to adapt to a constantly changing primary access region.

In the other simulator results, we see that the number of memory region switches decreases as a function of α. **[Ethan: show this in the appendix or don't mention it at all.]**

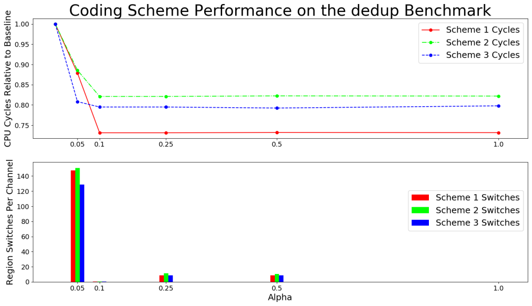


Figure 18: The simulation results for the dedup PARSEC benchmark. The line plot represents the number of CPU cycles needed and the bar plot represents the number of items the dynamic coding unit chooses to encode a new memory region. The results from the other PARSEC benchmarks are similar.

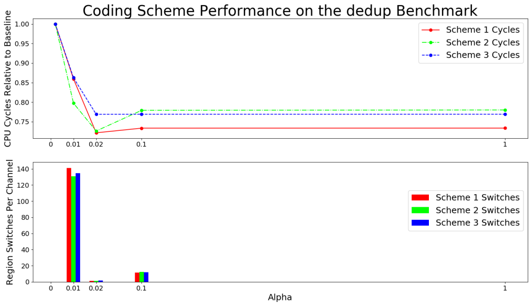


Figure 19: The simulation results for the same trace simulated in Figure 18 but with a memory partition coefficient $r = .01$

6. ACKNOWLEDGEMENTS

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

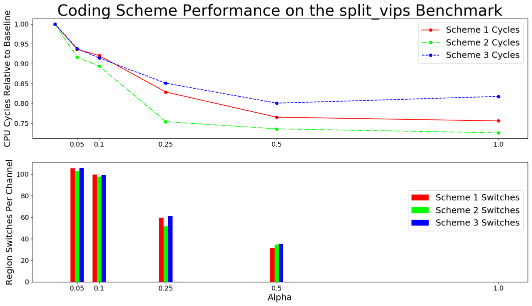


Figure 20: The simulation results of the augmented vips trace pictured in Figure 16

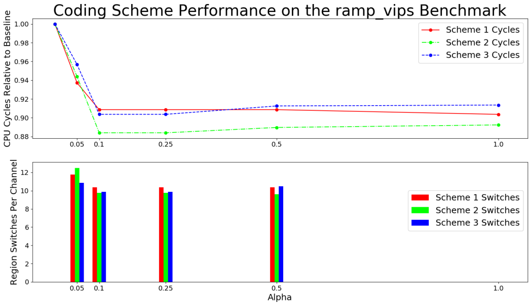


Figure 21: The simulation results of the augmented vips trace pictured in Figure 17

7. 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. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, “Learning memory access patterns,” in *International Conference on Machine Learning*, 2018, pp. 1919–1928.

[34] 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.

[35] C. Bienia and K. Li, “Parsec 2.0: A new benchmark suite for chip-multiprocessors,” in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[36] M. Gebhart, J. Hestness, E. Fatehi, P. Gratz, and S. W. Keckler, “Running parsec 2.1 on m5,” The University of Texas at Austin, Department of Computer Science, Tech. Rep., October 2009.

APPENDIX

A. CODE DESIGN

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

B. MOTIVATION FOR DYNAMIC CODING

[Ethan: move to the appendix, section 2, or cut altogether] Bank conflicts are most likely to occur when regions of shared memory are localized to certain memory regions. Multi-core systems often generate memory access requests to overlapping memory regions. By dynamically coding certain memory locales, the proposed memory system aim to resolve the bank conflicts which occur during periods of heavy memory access in multi-core systems.

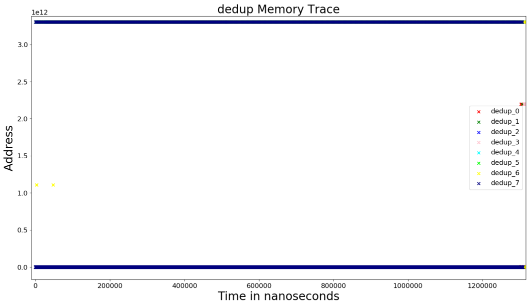


Figure 22: Memory Access from the Dedup PARSEC benchmark. This trace was generated using 8 cores.

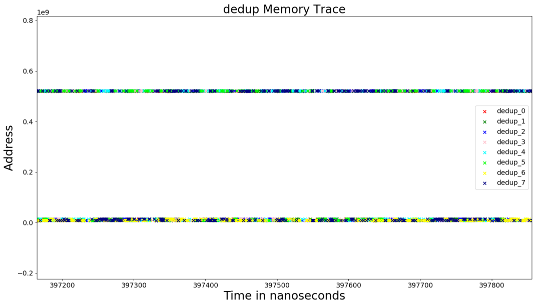


Figure 23: Memory Access from the Dedup PARSEC benchmark demonstrating the density of memory accesses

An examination of the memory trace from one of the PARSEC benchmarks illustrates a scenario where dynamic coding works well. Figure 22 shows the memory trace of a simulation of an 8-core system running the dedup PARSEC benchmark. The y-axis shows the address accessed by the cores. The x-axis shows the access time in nanoseconds. This plot shows that most of the accesses from various cores are primarily located in the lower memory band. Greater than 95% of all memory accesses are in this band. Figure 23 magnifies this band and reveals that the lower band is composed of two subbands of roughly equal density. In a scenario where the dynamic coder can choose to encode two memory blocks it would would detect that nearly all memory access are localized to the primary memory bands, so only those regions would be encoded.

Figure 23 shows that the Dedup benchmark contains very dense memory utilization. Across all processors, there is an average of 1.11 ns between accesses per core. This implies an average of 2.22 cycles between memory access requests per 2 Ghz processor. Crucially, the most heavily used memory regions are stationary with respect to time for all PARSEC benchmarks.

Figure 22 shows an entire Dedup memory trace. There are two major bands clearly visible in this image, and the bands stay in the same memory regions for the entirety of the trace. Figure 23 is a magnified view of the bottom band. This figure reveals that the bottom band is composed of two sub-bands which are also stationary with respect to time. The structure of the dedup the memory trace is representative for all the PARSEC benchmarks. It is also clear from this image that the memory regions utilized by all of the processors overlap sufficiently to create bank conflicts.

[Ethan: capitalize Dedup? change the font to `dedup`**??]**