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

Anonymous Author(s)

Abstract

Many performance critical systems today rely on multi-port memories to keep up with increasing memory-access demands. However, the large area footprints and complexity of existing multi-port memory designs limit their applicability. This paper explores a theoretic coding framework to address this problem. In particular, this paper introduces a framework for encoding data across multiple single-port memory banks in order to *algorithmically* emulate the functionality of multi-port memory systems.

We propose three code designs with significantly lower storage overhead compared to existing replication-based emulations of multi-port memories. We also demonstrate a memory controller design that utilizes redundancy across coded memory banks to schedule read and write requests. Furthermore, we explore how *dynamic coding* can be used to reduce memory overhead. We then show the performance improvements the code designs yield when compared to a traditional uncoded-memory design. **[MATT: Do we still need keywords in ISMM format?]**

Keywords keyword1, keyword2, keyword3

1 Introduction

Loading and storing information to memory is an intrinsic part of any computer program. As illustrated in Figure 1, the past few decades have seen the performance gap between processors and memory grow. Even with the saturation and demise of Moore's law [31, 33, 36], processing power is expected to grow as multi-core architectures become more reliable [9]. The end-to-end performance of a program heavily depends on both processor and memory performance. Slower memory systems can bottleneck computational performance. This has been driving motivation for computer architects and researchers to explore strategies for shortening memory access latency, including sustained efforts towards enhancing the memory hierarchy [21]. Despite these 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, and the processor is stalled as it waits for the shared memory to return the requested information. In multi-core systems, shared memory access conflicts between cores result 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 is sent back to the corresponding processor. In scenarios where multiple cores request access to memory locations in 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, the resultant increases in memory access latency causes the multi-core system to slow.

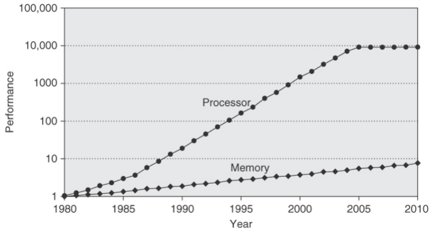


Figure 1. *The gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over a* 30 *year span [12].*

We address the issue of increased latency by introducing a coded memory design. The main principle behind our memory design is to distribute accesses intended for a particular bank across multiple banks. We redundantly store encoded data, and we decode memory for highly requested memory banks using idle memory banks. This approach allows us to simultaneously serve multiple read requests intended for a particular bank. 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.

PL'18, January 01–03, 2018, New York, NY, USA
2018.

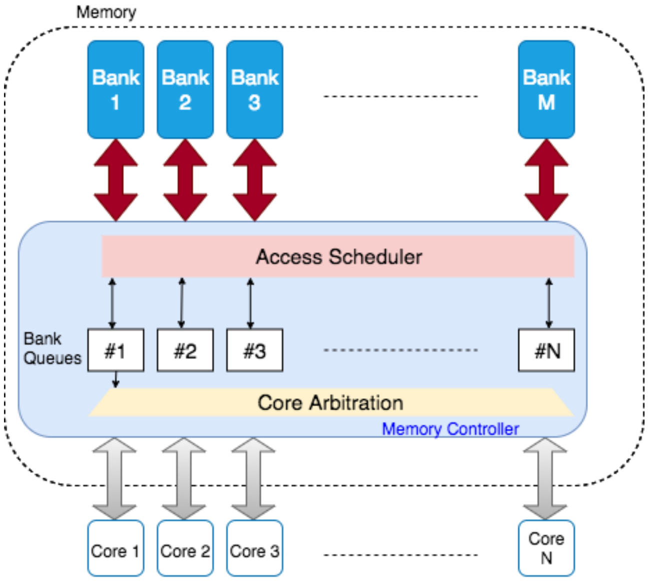


Figure 2. *General multi-core architecture with a shared memory. N processor cores share a memory consisting of M banks.*

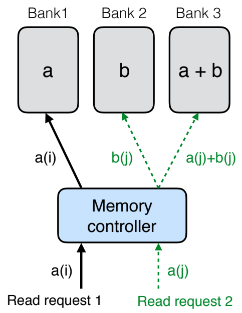


Figure 3. *Here the redundant memory in Bank 3 enables multiple read accesses to Bank 1 or 2. Given two read requests $\{a(i), a(j)\}$ directed to Bank 1, we can deal with bank conflict in the following manner: 1) The first request for $a(i)$ is directly served by Bank 1 itself. 2) The read request for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

Hybrid memory designs such as the one in Figure 3 have additional requirements on top of serving read requests. The presence of redundant parity banks raises a number of challenges while serving write requests. The memory overhead of redundant memory storage adds to the overall cost of such systems, so efforts must be made to minimize this overhead. Finally, the heavy memory access request rate possible in multi-core scenarios necessitates sophisticated scheduling strategies to bes performed by the memory controller. In this paper we address these design challenges and evaluate potential solutions in a simulated memory environment.

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 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 [6, 10, 14, 26], allow us to realize the functionality of multi-port memories from 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 operations.
- 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.
- 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 detection of continuous heavily accessed regions of memory. The second solution involves predicting the patterns of memory addresses over time.
- Finally, we conduct a detailed evaluation of the proposed designs of shared memory systems in Section 5. We implement our memory designs by extending Ramulator, a DRAM simulator. [20]. We use the gem5 simulator [8] to create memory traces of the PARSEC benchmarks [3] which are input to our extended version of Ramulator. We then observe the execution-time speedups our memory designs yield.

2 Background and Related Work

2.1 Emulating multi-port memories

Multi-port memory systems are essential for multi-core computation. Individual cores may request memory simulateously, and absent a multi-port memory system some cores will stall. Designing multi-port memory system has significant costs. Complex circuitry and large area requirements for multi-port bit-cells are significantly higher than those for single-port bit-cells [32, 34]. This motivates the exploration of algorithmic and systematic designs that emulate multi-port memories using single-ported memory banks [2, 7, 15, 16, 28]. Attempts have been made to emulate multi-port memory using replication based designs [4], however the resulting memory architectures are very large. **[Ethan: Move some of this earlier?]**

2.1.1 Read-only Support

Replication-based designs are often proposed as a method for multi-port emulation. Suppose that a memory design is required to support only read requests, say r read requests per memory clock cycle. A simple solution is storing 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 design completely avoids bank conflicts for up to r read request in a memory clock cycle.

Remark 1. *If we compare the memory design in Figure 4 with that 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, the access process for the design in Figure 3 involves some computation. This observation raises the notion that sophisticated coding schemes allow for storage efficient designs compared to replication based methods [22]. However, this comes at the expense of increased computation required for decoding.*

2.1.2 Read and Write Support

[ANKIT: Mainly describing the results from the work of Auerbach, Chen, and Paul[2].] A proper emulation of multi-port memory must be able to serve write requests. A challenge that arises from this requirement is tracking the state of memory. In replication-based designs where original data banks are duplicated, the service of writes requests results in differences in state between the original and duplicate banks.

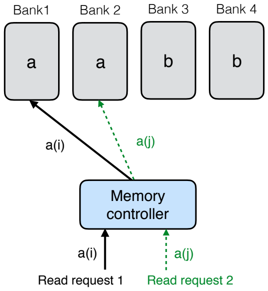


Figure 4. A 2-replication design which supports 2 read requests per bank. In this design, the data is partitioned between two banks $\mathbf{a} = [a(1), \ldots, a(L)]$ $\mathbf{b} = [b(1), \ldots, b(L)]$ and duplicated.

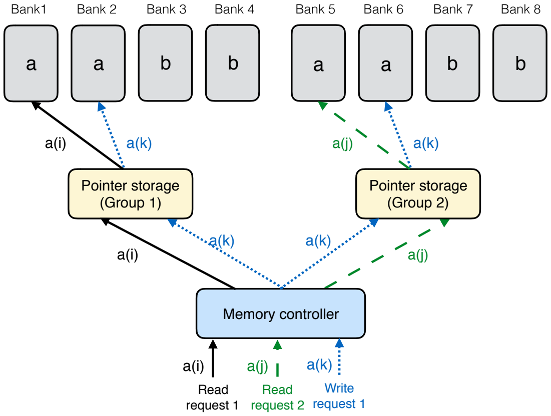


Figure 5. A 4-replication based design to support $r = 2$ read requests and $w = 1$ write requests. Both collections of information elements $\mathbf{a} = [a(1), \ldots, a(L)]$ and $\mathbf{b} = [b(1), \ldots, b(L)]$ are replicated to obtain $r \cdot (w+1) = 4$ single-port memory banks. These banks are then partitioned into $r = 2$ disjoint groups, Banks $1 - 4$ and Banks $5 - 8$. Suppose that there are two read requests for $\{a(i), a(j)\}$ and a write request for $\{a(k)\}$. The memory architecture here enables the memory controller to schedule all three requests targeting bank **a** in the same memory cycle. The two read requests are served using one instance of bank **a** in each of the disjoint groups. The write request is served to one of the instances of **a** in each group. We serve the write requests to each disjoint group to ensure that each group contains up-to-date data. Each group must contain up-to-date data so that any arbitrary set of 2 read requests can be served by the two groups. As the write is served, the pointer storage is updated to keep track of the state of the data in the banks.

Replication-based solutions to the problems presented when supporting write requests involve creating yet more duplicate banks. A replication-based multi-port memory emulation that simultaneously supports r read requests and w write requests requires 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. As in previous illustrations, we have two groups of symbols $\mathbf{a} = [a(1), \ldots, a(L)]$ 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 into $r = 2$ disjoint groups. Each group contains $(w+1) = 2$ memory banks. An additional storage space, the pointer storage, is required keep track the state of the data in the banks.

2.2 Storage-efficient emulation of multi-port memories

As described in Section 2.1, introducing redundancy to systems which use single-port memory banks allows such systems emulate the behavior of multi-port banks. In a setup where multi-port reads are supported (cf. Section 2.1.1) such emulation has little computational and storage cost. Emulating multi-port read and write systems is more costly (cf. Section 2.1.2). A greater number of single-port memory banks are needed, and systems which redundantly store memory require tracking of the various versions of the data elements present in the memory banks. Furthermore, the presence of varying version of elements in the banks complicates the process of arbitration, as some memory banks may contain stale elements. Many programs in multi-core environments involve significant numbers of write requests, so any system which emulates multi-port memory using single-port memory must take these complications 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 by the memory controller. It is possible to mitigate 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. Additionally, we believe that large storage overhead is a more fundamental issue that needs to be addressed before the emulation of the multi-port memories is feasible. In particular, the large replication factor in a naive emulation create so large a storage overhead that the resulting area requirements of such designs are impractical.

In order to reduce the storage overhead incurred by multi-port emulation, we avoid the native $r \cdot (w+1)$-replication design. Another approach arises from the observation that some data banks are left unused during arbitration in individual memory cycles, while other data banks receive multiple requests. We encode the elements of the data banks using specific coding schemes to generate parity banks. Elements drawn from multiple data banks are encoded and stored in the parity banks. This approach allows us to utilize idle data banks to decode elements stored in the parity banks in service of multiple requests which target the same data bank. We recognize that this approach leads to increased complexity at the memory controller. *However, we show that the increase in complexity can be kept within an acceptable level while ensuring storage-efficient emulation of multi-port memories.*

2.3 Related work

Coding theory is a well-studied field which aims to mitigate the challenges of underlying mediums in information processing systems [5, 22]. The field has enabled both reliable communication across noisy channel and reliability in fault-prone storage units. Recently, we have witnessed intensive efforts towards the application of coding theoretic ideas to design large scale distributed storage systems[13, 25, 29]. 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 [14, 26, 27, 35]. In this paper, we rely on such coding techniques to emulate multi-port memories using single-port memory banks. We note that the existing work on batch codes [14] focuses only on the read requests, but the emulation of multi-port memory must also handle write requests.

Coding schemes with low update complexity that can be implemented at the speed memory systems require have also been studied [1, 24]). Our work is distinguished from the majority of the literature on coding for distributed storage, because we consider the interplay between read and write requests and how this interplay effects memory access latency.

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 [17–19, 30] 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 [11]. 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 [23] 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 [23]. 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 [15, 16], 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 [15, 16] as we specifically rely on different coding schemes arising under the framework of batch codes [14]. Additionally, due to the employment of distinct coding techniques, the design of memory controller in our work also differs from that in [15, 16].

[ANKIT: Also cite the work by Rivest et al. [28] and Endo, Matsumura and Yamada [7].]

3 Codes to Improve Accesses

Introducing redundancy into a storage space comprised of single-port memory banks enables simultaneous memory access. In this section we propose memory designs that utilize coding schemes which are designed for access-efficiency. We first define some basic concepts with an illustrative example and then describe 3 coding schemes in detail.

3.1 Coding for memory banks

A coding scheme defines how memory is encoded to yield redundant storage. The memory structures which store the original memory elements are known as *data banks*. The elements of the data banks go through an *encoding process* which generate a number of *parity banks*. The parity banks contain elements constructed from elements drawn from two or more data banks. A linear encoding process such as XOR may be used to minimize computational complexity. The following example further clarifies these concepts and provides some necessary notation.

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

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

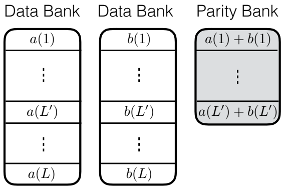


Figure 6. Notation of example parity design. ok?

Remark 2. *Figure 6 illustrates this coding scheme. Because the parity bank is based on those rows of the data banks that are indexed by the set* $[L'] \subseteq [L]$*, we 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 3. *Note that we allow for the data banks and parity banks to have different sizes, i.e.* $L \neq L'$*. This freedom in memory design can be utilized to reduce the storage overhead of parity banks based on the underlying application. The case when the size of a parity bank is smaller than a data bank, i.e.* $L' < L$*, we say that the parity bank is a* shallow bank*. We note that it is reasonable to assume the existence of shallow banks, especially in proprietary designs of integrated memories in a system on a chip (SoC).*

Remark 4. *Note that the size of shallow banks is a design choice which is controlled by the parameter* $0 < \alpha \leq 1$*. A small value of* α *corresponds to small storage overhead. The choice of a small* α *comes at the cost of limiting parity memory accesses to certain memory ranges. In Section 4.5 we discuss techniques for choosing which regions of memory to encode. In scenarios where many memory accesses are localized to small regions of memory, shallow banks can support many parallel memory accesses for little storage overhead. For applications where memory access patterns are less concentrated, the*

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

robustness of the parity banks allows one to employ a design with $\alpha = 1$.

3.1.1 Degraded reads and their locality

The redundant data generated by a coding scheme mitigates bank conflicts by supporting multiple **read** accesses to the original data elements. Consider the coding scheme illustrated in Figure 6 with a parity bank $\mathbf{p} = \mathbf{a}([L']) + \mathbf{b}([L'])$. In an uncoded memory system simultaneous read requests for bank **a**, such as $a(1)$ and $a(5)$, result in a bank conflict. The introduction of **p** allows both read requests to be served. First, $a(1)$ is served directly from bank **a**. Next, $b(5)$ and $p(5)$ are downloaded. $a(5) = b(5) + p(5)$, so a(5) is recovered by means of the memory in the parity bank. A read request which is served with the help of parity banks is termed as *degraded read*. Each degraded read has a parameter *locality* associated with it which corresponds to the total number of banks used to serve it. In this case, the degraded read for $a(1)$ using **b** and **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 [14] which enable parallel access to the content stored in a large scale distributed storage system. 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 $L' = \alpha L$. We partition the 8 data banks into two groups of 4 banks. The underlying coding scheme produces shallow parity banks by separately encoding data banks from the two groups. Figure 7 shows the resulting memory banks. The storage overhead of this design is $12\alpha L$ which amounts to the rate[2] of the coding scheme is

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

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 **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 **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 **c** and **d**.

Worst case analysis: This code design (cf. Figure 7) may fail to utilize any parity banks depending on the requests waiting to be served. The worst case scenario for this code design occurs when there are non-sequential

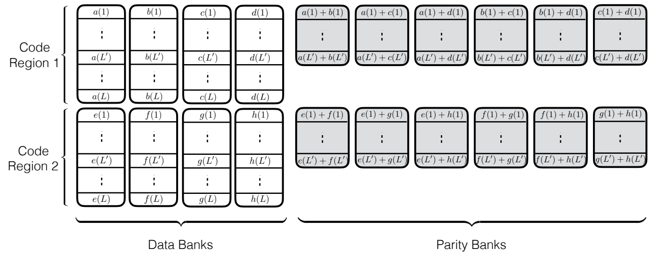


Figure 7. Code Design I

[2] 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.

and non-consecutive access to the memory banks. Take for example a scenario where we only consider the first four banks of the code design. The following read requests are waiting to be served:

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

Because none of the requests share the same row index, we are unable to utilize the parity banks. 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.

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. The first code region is similar to the previous code design, as it contains parity elements constructed from two data banks. The second code region contains data directly duplicated from single data banks. This code design further 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.

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.

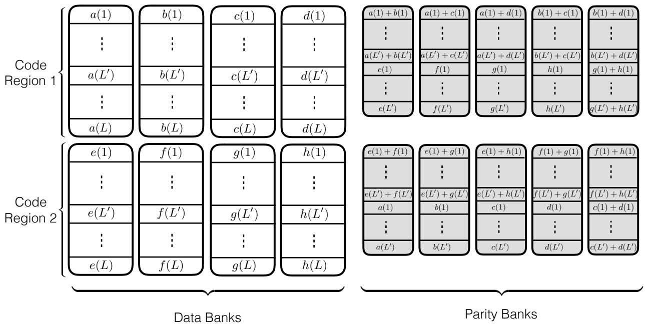


Figure 8. Code Design II

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

Worst case analysis: The code scheme can enable 5 simultaneous accesses in a single memory clock cycle in the worst case. These are non-sequential and non-consecutive accesses to the memory banks. For example, when the access pattern corresponds to the rows $\{a(1), b(6), c(9), d(15), e(20)\}$, 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 next code design we discuss has locality 3, so each degraded read requires two memory banks to be served. 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 shows this design. 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 as a result of its increased locality.

This design supports 4 simultaneous read access per bank per memory clock cycle as demonstrated by the following example. Suppose there are requests for rows $\{a(1), a(2), a(3), a(4)\}$. $a(1)$ can be served directly from **a**. $a(2)$ is served by means of a parity read and reads to banks **b** and **c**, $a(3)$ is served by means of a parity read and reads to banks **d** and **g**, and $a(4)$ is served by means of a parity read and reads to banks **e** and **z**.

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.

Worst case analysis: Similar to design I and design II, the number of reads per cycle is equal to the number of data banks.

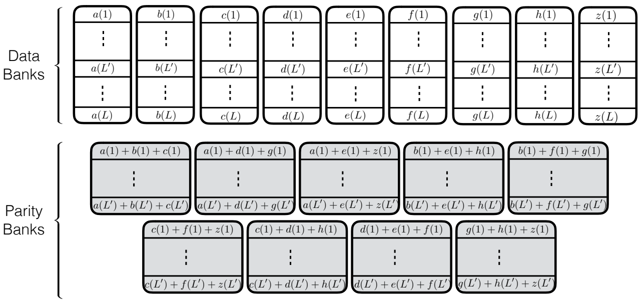

Data Banks
a(1) b(1) c(1) d(1) e(1) f(1) g(1) h(1) z(1)
a(L') b(L') c(L') d(L') e(L') f(L') g(L') h(L') z(L')
a(L) b(L) c(L) d(L) e(L) f(L) g(L) h(L) z(L)
Parity Banks
a(1)+b(1)+c(1) … a(L')+b(L')+c(L')
a(1)+d(1)+g(1) … a(L')+d(L')+g(L')
a(1)+e(1)+z(1) … a(L')+e(L')+z(L')
b(1)+e(1)+h(1) … b(L')+e(L')+h(L')
b(1)+f(1)+g(1) … b(L')+f(L')+g(L')
c(1)+f(1)+z(1) … c(L')+f(L')+z(L')
c(1)+d(1)+h(1) … c(L')+d(L')+h(L')
d(1)+e(1)+f(1) … d(L')+e(L')+f(L')
g(1)+h(1)+z(1) … g(L')+h(L')+z(L')

Figure 9. Code Design III

Remark 5. *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 $\{\mathbf{a}, \mathbf{b}, \ldots, \mathbf{h}\}$ as shown in Figure 10.*

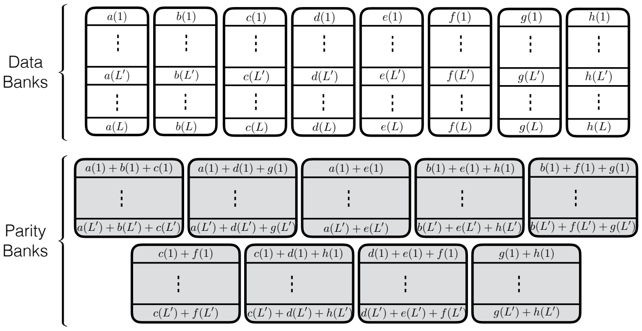

Data Banks
a(1) b(1) c(1) d(1) e(1) f(1) g(1) h(1)
a(L') b(L') c(L') d(L') e(L') f(L') g(L') h(L')
a(L) b(L) c(L) d(L) e(L) f(L) g(L) h(L)
Parity Banks
a(1)+b(1)+c(1) … a(L')+b(L')+c(L')
a(1)+d(1)+g(1) … a(L')+d(L')+g(L')
a(1)+e(1) … a(L')+e(L')
b(1)+e(1)+h(1) … b(L')+e(L')+h(L')
b(1)+f(1)+g(1) … b(L')+f(L')+g(L')
c(1)+f(1) … c(L')+f(L')
c(1)+d(1)+h(1) … c(L')+d(L')+h(L')
d(1)+e(1)+f(1) … d(L')+e(L')+f(L')
g(1)+h(1) … g(L')+h(L')

Figure 10. Code Design III with 8 data banks

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 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 trade-off of the pattern builder algorithms is the relationship between the complexity of the algorithm and the number of requests the algorithm schedules.

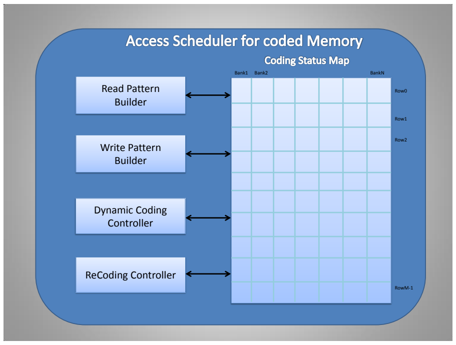


Figure 11. Access scheduler for coded memory

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

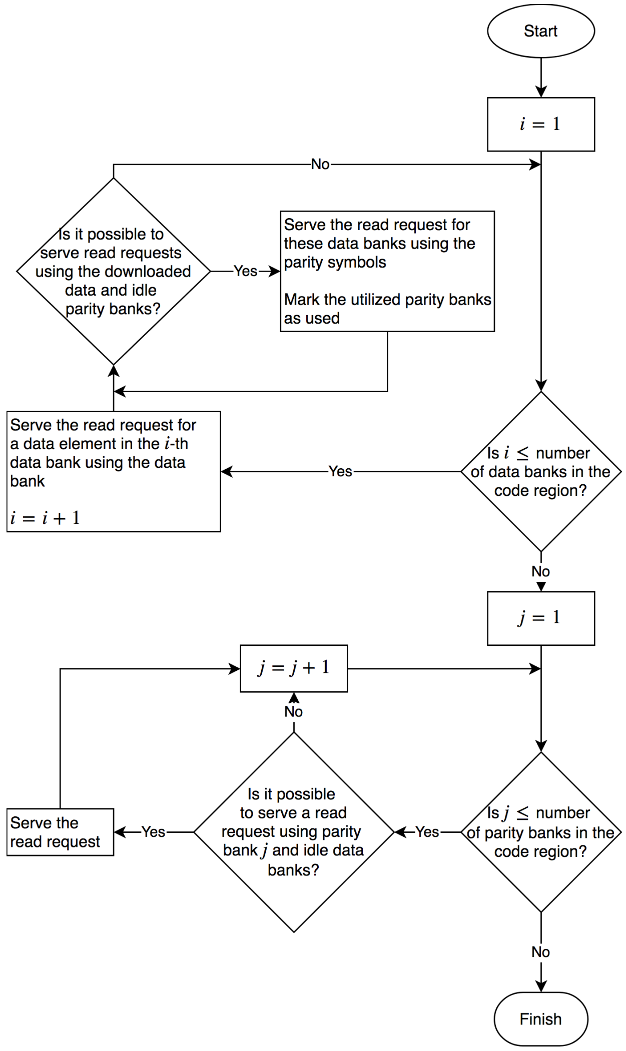

Start
$i = 1$
No
Is it possible to serve read requests using the downloaded data and idle parity banks?
Yes
Serve the read request for these data banks using the parity symbols
Mark the utilized parity banks as used
Serve the read request for a data element in the i-th data bank using the data bank
$i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

data. We assume that the elements of the code status table are accessible at a very fast rate.

4.2 Read pattern builder

The access scheduler uses the read pattern builder algorithm to determine which requests to serve using parity banks 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 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.

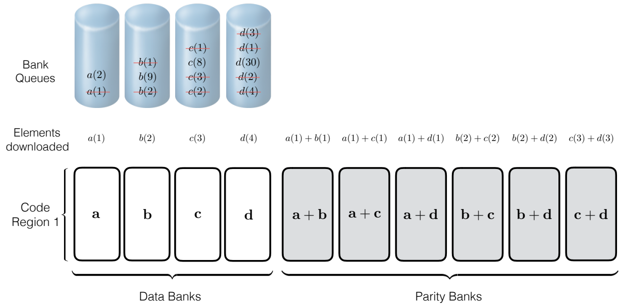


Figure 13. Illustration of the algorithm to build a read request pattern to be served in a given memory cycle. All the read requests associated with the strike-through elements are scheduled to be served in a given memory cycle. The figure also shows the elements downloaded from all the memory banks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

Remark 6. *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.

Figure 15 shows an example write pattern produced by our algorithm. Parity banks increase the maximum

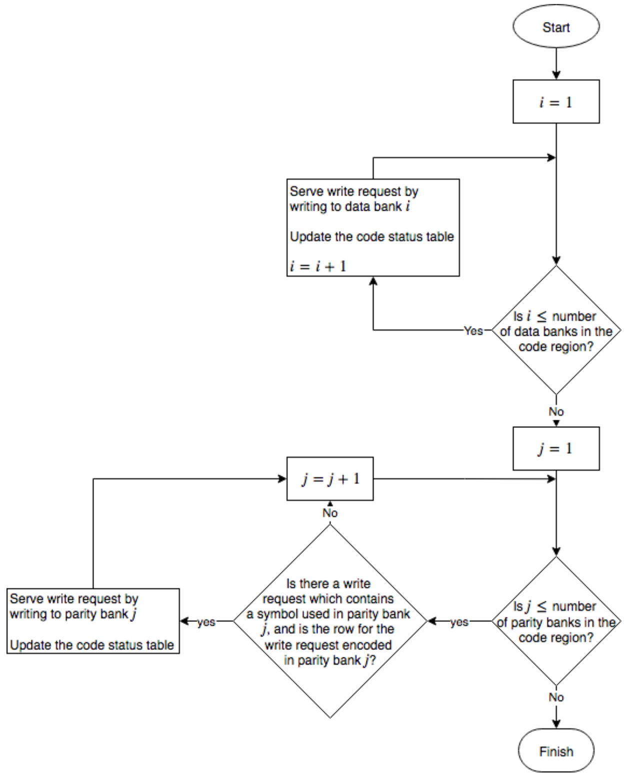

Start
$i = 1$
Serve write request by writing to data bank i
Update the code status table
$i = i + 1$
Is $i \leq$ number of data banks in the code region?
Yes
No
$j = 1$
$j = j + 1$
No
Is there a write request which contains a symbol used in parity bank j, and is the row for the write request encoded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

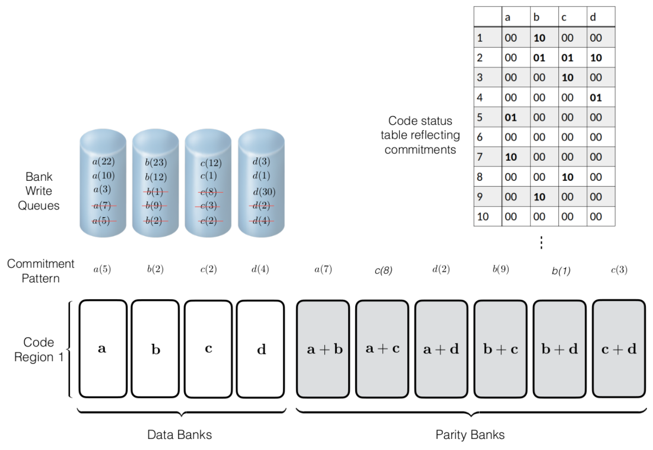
Figure 14. Flowchart of write pattern builder

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.

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 *ReCoding 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



	a	b	c	d
1	00	**10**	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	**10**	00
9	00	**10**	00	00
10	00	00	00	00

⋮

Figure 15. The behavior of the write pattern builder on a 4-bank memory system

contain the current cycle number so that the ReCoding Unit may prioritize older requests.

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. A memory region of length r is reserved by the dynamic coding unit for constructing new encoded regions, and a memory region of length $\alpha - r$ is reserved for active encoded regions. If the memory ceiling $\alpha - r$ is reached when a new memory region is encoded, the unit evicts the least frequently used encoded region.

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 [3, 8] 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 [20] 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 [3]. These benchmarks allow us to observe how the proposed memory system performs in dense memory access scenarios. The gem5 simulator [8] 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.

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.

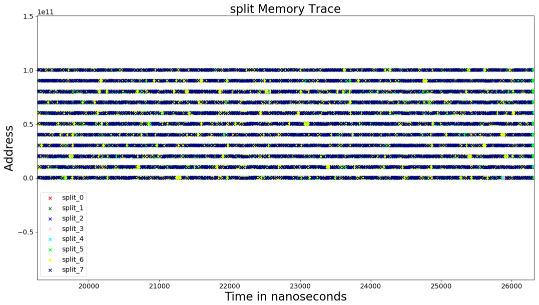


Figure 16. The vips benchmark after splitting the primary access bands into multiple additional bands.

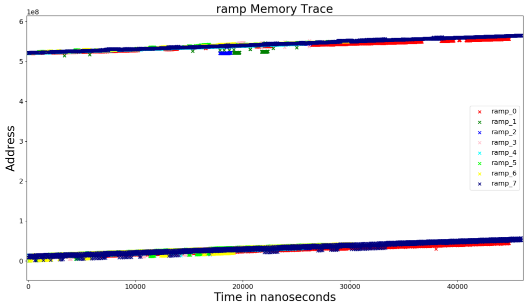


Figure 17. The vips benchmark after adding a ramp to the major memory bands.

5.3 Simulation Results

Given sufficient memory overhead, we see a consistent 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.

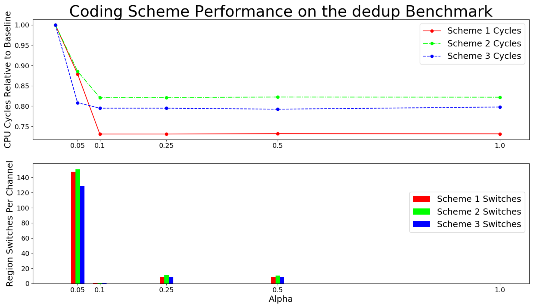


Figure 18. The simulation results for the dedup PARSEC benchmark. The line plot represents the number of CPU cycles needed and the bar plot represents the number of items the dynamic coding unit chooses to encode a new memory region. The results from the other PARSEC benchmarks are similar.

6 Acknowledgements

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

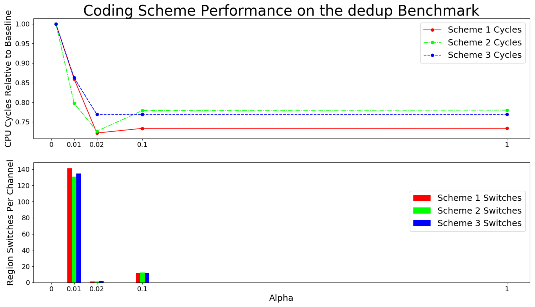


Figure 19. The simulation results for the same trace simulated in Figure 18 but with a memory partition coefficient $r = .01$

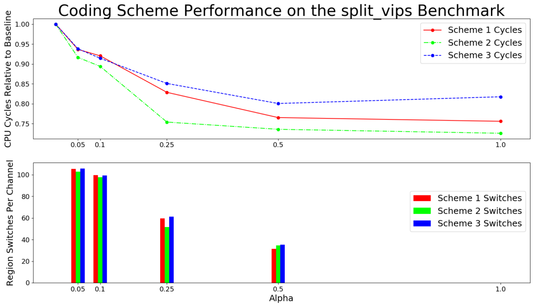


Figure 20. The simulation results of the augmented vips trace pictured in Figure 16

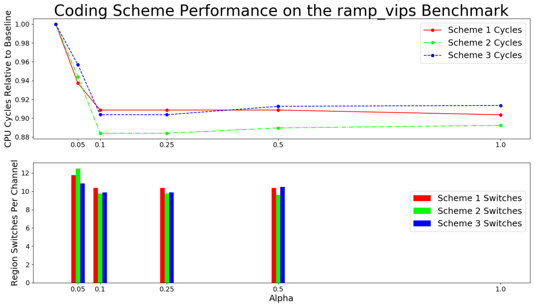


Figure 21. The simulation results of the augmented vips trace pictured in Figure 17

References

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

[2] D. J. Auerbach, T. C. Chen, and W. J. Paul. 1988. High-performance multiple port memory. https://www.google.com/patents/US4766535

[3] Christian Bienia and Kai Li. 2009. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*.

[4] B. A. Chappell, T. I. Chappell, M. K. Ebcioglu, and S. E. Schuster. 1993. Virtual multi-port RAM. https://www.google.com/patents/US5204841

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

[6] A. G. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. 2010. Network Coding for Distributed Storage Systems. *IEEE Transactions on Information Theory* 56, 9 (Sept 2010), 4539–4551.

[7] K. Endo, T. Matsumura, and J. Yamada. 1991. Pipelined, time-sharing access technique for an integrated multiport memory. *IEEE Journal of Solid-State Circuits* 26, 4 (Apr 1991), 549–554.

[8] Mark Gebhart, Joel Hestness, Ehsan Fatehi, Paul Gratz, and Stephen W. Keckler. 2009. *Running PARSEC 2.1 on M5*. Technical Report. The University of Texas at Austin, Department of Computer Science.

[9] D. Geer. 2005. Chip makers turn to multicore processors. *Computer* 38, 5 (May 2005), 11–13.

[10] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin. 2012. On the Locality of Codeword Symbols. *IEEE Transactions on Information Theory* 58, 11 (Nov 2012), 6925–6934.

[11] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning Memory Access Patterns. In *International Conference on Machine Learning*. 1919–1928.

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

[13] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. 2012. Erasure Coding in Windows Azure Storage. In *Proc. of 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX, 15–26.

[14] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. 2004. Batch codes and their applications. In *Proc. of thirty-sixth annual ACM symposium on Theory of computing (STOC)*. 262–271. https://doi.org/10.1145/1007352.1007396

[15] S. Iyer and S.-T. Chuang. 2013. System and method for storing data in a virtualized high speed memory system with an integrated memory mapping table.

[16] S. Iyer and S.-T. Chuang. 2015. System and method for storing multiple copies of data in a high speed memory system.

[17] A. Jain and C. Lin. 2013. Linearizing Irregular Memory Accesses for Improved Correlated Prefetching. In *Proc. of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, New York, NY, USA, 247–259.

[18] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jimenez. 2014. B-Fetch: Branch Prediction Directed Prefetching for Chip-Multiprocessors. In *Proceedings of 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 623–634.

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

[20] Y. Kim, W. Yang, and O. Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (Jan 2016), 45–49.

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

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

[23] M. A. Maddah-Ali and U. Niesen. 2016. Coding for caching: fundamental limits and practical challenges. *IEEE Communications Magazine* 54, 8 (Aug 2016), 23–29.

[24] A. Mazumdar, V. Chandar, and G. W. Wornell. 2014. Update-Efficiency and Local Repairability Limits for Capacity Approaching Codes. *IEEE Journal on Selected Areas in Communications* 32, 5 (May 2014), 976–988.

[25] K.V. Rashmi, Nihar B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. 2014. A ”Hitchhiker’s” Guide to Fast and Efficient Data Reconstruction in Erasure-coded Data Centers. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 331–342.

[26] A. S. Rawat, D. S. Papailiopoulos, A. G. Dimakis, and S. Vishwanath. 2016. Locality and Availability in Distributed Storage. *IEEE Transactions on Information Theory* 62, 8 (Aug 2016), 4481–4493.

[27] A. S. Rawat, Z. Song, A. G. Dimakis, and A. Gal. 2016. Batch Codes Through Dense Graphs Without Short Cycles. *IEEE Transactions on Information Theory* 62, 4 (April 2016), 1592–1604.

[28] R. L. Rivest and L. A. Glasser. 1991. *A Fast Multiport Memory Based on Single-Port Memory Cells*. Technical Report MIT/LCS/TM-455. Massachusetts Inst. of Tech. Cambridge Lab. for Computer Science.

[29] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. 2013. XORing Elephants: Novel Erasure Codes for Big Data. In *Proc. of 39th International Conference on Very Large Data Bases (VLDB)*, Vol. 6. 325–336.

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

[31] T. Simonite. 2016. Moore’s Law Is Dead. Now What? https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/. Online; accessed 31 July 2017.

[32] T. Suzuki, H. Yamauchi, Y. Yamagami, K. Satomi, and H. Akamatsu. 2008. A Stable 2-Port SRAM Cell Design Against Simultaneously Read/Write-Disturbed Accesses. *IEEE Journal of Solid-State Circuits* 43, 9 (Sept 2008), 2109–2119.

[33] M. M. Waldrop. 2016. The chips are down for Moore’s law. *Nature* 530, 7589 (2016), 144–147.

[34] D. P. Wang, H. J. Lin, C. T. Chuang, and W. Hwang. 2014. 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* 61, 3 (March 2014), 188–192.

[35] Z. Wang, H. M. Kiah, Y. Cassuto, and J. Bruck. 2017. Switch Codes: Codes for Fully Parallel Reconstruction. *IEEE Transactions on Information Theory* 63, 4 (April 2017), 2061–2075.

[36] W. A. Wulf and S. A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News* 23, 1 (March 1995), 20–24.

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.

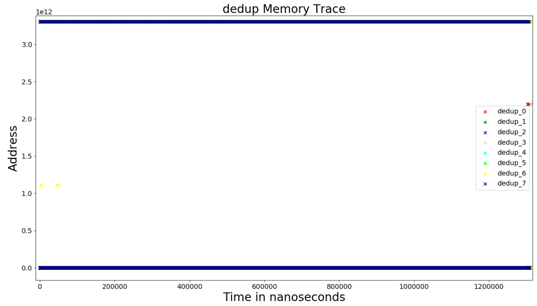


Figure 22. Memory Access from the Dedup PARSEC benchmark. This trace was generated using 8 cores.

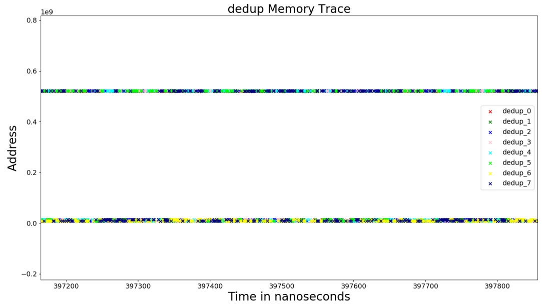


Figure 23. Memory Access from the Dedup PARSEC benchmark demonstrating the density of memory accesses

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: change the font to `dedup`?]