RJ 10321 (A0408-003) August 2, 2004
Computer Science

IBM Research Report

Performance Metrics for Erasure Codes in Storage Systems

James Lee Hafner, Veera Deenadhayalan, Tapas Kanungo, KK Rao
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

PERFORMANCE METRICS FOR ERASURE CODES IN STORAGE SYSTEMS

James Lee Hafner, Veera Deenadhayalan, Tapas Kanungo and KK Rao

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

ABSTRACT:

Erasure codes that have good theoretical properties for a communication channel, need not have good properties for use in storage systems. Choosing the code Hamming distance to meet a desired reliability goal is only the first step in the erasure code selection process for a storage system. A storage system designer needs to carefully consider many other metrics that are specific to storage systems and their specific system configuration. In this article we first present one model of a storage system and then describe the metrics that are relevant in that model. We then outline a set of array codes that have been proposed for use in storage systems and compare and analyze them using our metrics. Our metrics assume a particular hardware architecture; however, the methodology can be easily adapted to other hardware models. Consequently, the principles described here can be modified by a storage system designer to fit his/her specific environment and so provide quantitative tools for array code comparison and selection criteria.

Contents

1. Introduction

A desirable disk-based storage system should have three fundamental properties. It should be economical. It should be fast. And it should be reliable. Unfortunately, it is difficult (if not impossible) to find schemes that satisfy all these properties. Furthermore, matters are made more complicated because there are many quite independent ways in which cost, speed, and reliability affect the system user. Cost for a system includes both the overall purchase cost (disk costs, hardware and software design, etc.) and the cost of maintaining the system (energy, space, replacing bad disks, etc) – the so-called total cost of ownership. A high performance storage system has high data throughput (in particular for user data update) but also recovers quickly from disk failures. A reliable system should not only preserve data indefinitely, but it should do so with as little maintenance as possible.

Unfortunately, there is an inherent incompatibility between these three desired properties. For example, increased reliability requires more redundancy which adversely affects update and recovery times as well as costs for additional components and drives. For this reason, much work has been done to develop and explore a range of storage system redundancy schemes, each with their own advantages and drawbacks. The designer of a storage system must ultimately choose from among the many existing schemes (and any he/she might invent) the one whose tradeoffs best match the system's expected use.

In the context of the erasure code used to protect the data, cost is primarily determined by the efficiency of the code (or the rate of the code). Reliability is primarily determined by the Hamming distance of the erasure code (how many disk failures it can tolerate), the probability of errors or failures and the array size (the number of disks in the array). In general, the array size is determined by customer demand or expectation (and dependent on the efficiency of the code). Consequently, we view these two metrics of cost and reliability as rather coarse metrics and only a first step for code selection. The purpose of this paper is to provide a detailed framework for performance metrics that can then be used to compare codes within a given cost/reliability range. Our framework includes a variety of scenarios and a mixture of metrics. These go well beyond the typical metric of disk seeks required for small writes.

The performance of a complete storage system can be affected by many components of the system, including host/storage interconnect, host or storage caching, workload characteristics, internal system buses, disk interfaces, and the disk themselves. As our goal is to focus only on the effect of the choice of array code with all other components equal, we focus only on that part of the system that is directly affected by the array code itself. In particular, we do not consider a cache model or place constraints on bus or interconnect, nor do we consider performance numbers such as IOPs (IOs per second) or MB/s (megabytes per second on the host/storage interconnect). We measure those points in the storage system itself where the array code consumes resources; a designer can then use our metrics to determine where a

given code might create bottlenecks or otherwise consume too much of the system.

In our examples, we only consider distance three codes (that can tolerate two disk failures). We selected this because (a) distance two is well-understood (and there aren't that many choices), (b) distance three is the next level that many systems must reach to be reliable given the failure realities of current disk drives, and (c) there are many different distance three codes that have been proposed, so there is a rich set of examples from which to choose. As more practical interest is generated and more examples of distance four codes are defined, the methodologies here could easily be applied to that reliability level as well.

The outline of the paper is as follows. In the next section, we present our work in the context of the literature. We then give a summary section on notation and vocabulary. In Section 4 we give a description of our idealized system architecture, models of system costs and basic metric primitives. Section 5 contains descriptions of the use cases or host IO scenarios to which we apply the basic metrics. The set of example array codes are described briefly in Section 6. The remaining sections contain highlights of the significant differences between the codes under the various metrics and use cases and conclusions one can draw from these metric results. We conclude with a brief summary. The complete set of raw numbers (all metrics, all use cases) and some additional information is presented in the appendix.

2. Related Literature

Research on RAID algorithms and architectures has been aggressively pursued in industry and academia. Early work done at IBM on storage redundancy algorithms was reported in [22]. Thereafter, a UC Berkeley group published an article [23] (see also [8]) describing various RAID algorithms and also coined the term RAID. General overview of storage subsystems and RAID terminology can be found in [13] and in the RAID Advisory Board (RAB) book [20].

Coding theory [4, 19, 3] provides the mathematical formalism for designing RAID algorithms. The classic Reed-Solomon code [26] was designed for a communication model. Coding and reconstruction algebra is performed over some finite field (e.g., $GF(2^m)$, see [15, 14] for details). While this approach has been proposed for use in storage systems [24], the overhead of finite field arithmetic in software is prohibitive, requiring the use of special purpose hardware.

Various new erasure codes like Turbo codes [18], Tornado codes [17], LT codes [16] and Raptor codes [27] have been proposed in the communication literature. However their probabilistic nature does not make them usable in a RAID storage controller system for preventing against failures while providing guarantees of reconstruction times.

Low-Density-Parity-Check (LDPC) codes [11, 18] are codes that try to minimize the number of non-zero entries in the generator and the parity-check matrices. This property is good for RAID algorithm design because it minimizes the number of IO operations, which

can dominate the cost in a storage system. In general, the design principles behind RAID algorithms are most similar to those behind LDPC codes. However the LDPC codes were designed for communication models and do not necessarily appropriately account for all costs related to storage systems.

Choosing the code by Hamming distance to meet a desired reliability goal is only the first step in the erasure code selection process for a storage system. Two erasure codes with the same distance can exhibit vastly different performance/cost factors. The automatic selection process described in [1] helps in the choice between different RAID levels but not in the selection of a code within that level. Extending the ideas in [1] to choose an erasure code within a RAID level is not practical. Hence, a storage system designer needs to carefully consider many other metrics that are specific to the storage system and compare metrics by prioritizing them for that given system configuration.

To improve reliability over that provided by RAID5 (that is, for better tolerance to disk failures), a number of array codes have been proposed in the literature (see Section 6 for details about most of the codes mentioned here). The BR99 codes [6], EvenOdd [5] and Row-Diagonal Parity codes [9] have similar characteristics. They are all extensions of a RAID4 code with an additional strip or disk that contains parity computed by a more complex algorithm than the RAID4 parity. There are no limits on the arrays sizes for these codes, though larger array sizes increase the complexity of the parity computations. These codes all store their parity on separate disks from the data (not accounting for parity rotations as in RAID5 versus RAID4).

There are other families of codes that store parity on the *same* disks with the user data. The X-Code [31] has two rows of parity blocks (more generally "chunks"). The parity computation algorithms are "diagonal-based". The ZZS code [32] has only one parity row but the parity computation algorithm is somewhat more complex. It has the feature that one disk contains only user data, the others contain both user data and one chunk of parity. Both of these codes require that the number of disks in the array be a prime number, though the ZZS code can be "reduced" by assuming that the all user data disk contains zeros (and so physically need not be present). The methodology in [2] provides a means to construct, for many values of even array sizes, examples of codes with one row of parity on user data disks (it does not work for 8 disks, but *ad hoc* constructions with the same quantitative and qualitative characteristics for this array size are known). The resulting constructions are also very similar to the "reduced" ZZS code.

All of the aforementioned codes are MDS (Maximum Distance Separable), that is, have optimal ratio of parity to user data. The codes proposed in [12] are codes with separate parity disks that relax this requirement for minimal numbers of parity disks or blocks; the goal in [12] was better overall reliability. These codes are less efficient and so more costly.

For further discussion of system reliability issues the reader is referred to [28]. For a discussion on benchmarks on memory-bandwidth utilization under practical workloads

see [21]; for storage system performance see [29]; for availability see [7]; and for RAID level selection see [1].

Many of the papers mentioned in this section touch on some issues related to performance metrics. In particular, new erasure code papers typically discuss the IO seek costs for the host short write case (minimal update costs). Some (e.g., [5] and others) give a simple model of parity computation costs under normal operation and perhaps in cases of reconstruction. As mentioned in the introduction, our goal is to provide an extensive set of metrics under multiple host IO scenarios and under many failure scenarios in the context of one model of a realistic system.

.

3. Definitions and Notations

We begin this section with a list of terms, definitions and notations that we use throughout the paper.

data A chunk of bytes or blocks that hold user data (unmodified host-supplied data).

parity A chunk of bytes or blocks that hold redundancy information generated from user data (typically by XOR operations).

element A unit or chunk of data or parity; this is the building block of the erasure code. In coding theory, this is the data that is assigned to a letter in the alphabet (not to be confused with the symbol). An alternative definition is the maximal unit of data that can be updated (host write) using the minimal number of disk IO commands, independent of the LBA. For one-dimensional codes this corresponds to a strip (see below).

stripe A maximal set of data and parity elements that are dependently related by redundancy (e.g., XOR) relations. This is synonymous with "code instance" in that it is a complete instantiation of an erasure code and is independent of any other instantiation. In some circles, this is called a "stride" and in other circles, this term is restricted to only the data portion (does not include parity, as we do here). (This should be not be confused with the term "array" defined below.)

strip A maximal set of elements in a stripe that are on one disk. We prefer this term to "disk" because in a collection of disks, one disk may contain strips from multiple stripes. In coding theory, this is the data mapped to a symbol of the code. A strip may contain only data, only parity or some of each.

array A collection of disks on which one or more stripes are instantiated. Each instance may (and should for reasons such as load-balancing) have a different logical mapping of strip to disk. See "stack" definition below.

stack A collection of stripes in an array that are related by a maximal set of permutations of logical mappings of strip number to disk. Maximal here means that the loss of any two (or one) *physical* disks covers all combinations of loss of two (or one) *logical* strips. RAID5 parity and data strip rotation (N rotations on N disks) versus the static assignment of RAID4 is an example of this type of permutation. Thus, our stack is a set of stripes over which the RAID5 rotation principle has been applied maximally for the purposes of uniformizing strip failure scenarios under any disk failure case.

For an N drive array, there are $N(N-1)/2$ possible combinations of two disk failures. Over a stack, each such failure accounts for exactly the same number of occurrences of each of the $N(N-1)/2$ possible combinations of two logical strip failures. (So the number of stripes in a stack in this case must be a multiple of $N(N-1)/2$). With no special symmetry in the code, the stack has $N!$ (factorial) stripes. As in RAID5, symmetry can significantly reduce this number.

Normal The mode or state of a stripe in which all strips can be read or written without error.

Degraded The mode of a stripe in which exactly one strip is lost (cannot be read or written). (See Note A below.)

Critical The mode of a stripe in which exactly two strips are lost. (See Note A below.)

horizontal code An array code in which parity within a stripe is stored on separate strips from the data of the stripe. RAID4 is an example of a horizontal code. (See Note B below.)

vertical code An array code in which parity within a stripe is stored on strips containing data of the stripe. The X-Code (see Section 6.6) is an example of a vertical code. (See Note B below.)

distance The minimum number d of strip failures that result in user data/information loss (losing less than d strips necessarily means no data loss).

efficiency The percent of the stripe that contains data; that is, the number of data elements divided by the total number of elements in the stripe. This is usually denoted by *Eff*. An optimally efficient, distance d code with N strips has efficiency $(N-(d-1))/N$.

data out-degree Denoted by *DoutD* and a function of a given data element, it is the number of parity elements into which the data element contributes (or "touches").

parity in-degree Denoted by *PinD* and a function of a given parity element, it is the number of data elements that are needed to compute the parity (or are "touching" the parity element or are "touched" by the parity element).

dependent data For a given data element and parity elements it touches, the dependent data are the other data elements touched by this set of parity elements. (In a graph with nodes assigned to each data and parity element and edges assigned by the "touch" relation, the dependent data for a given data element is the set of data elements at distance 2 from the given data element.)

parity compute (PC) An algorithm for updating parity by computing the new parity values from *all* the dependent data in the stripe. In this case, no old parity or old data (data that is to be updated) is read from the disk.

parity increment (PI) An algorithm for updating parity by computing the new parity values from the old values of parity and data (that is, a read-modify-write of the parity).

Note A: More precisely, the term Critical should be used when some number of strips have been lost and loss of any additional strip (or perhaps data element or block) implies unrecoverability of user data. Degraded should be any intermediate state when the stripe has lost some strips but has not yet reached Critical state. So, a RAID5 array with one strip down is technically Critical (and is never Degraded), however, all our examples are distance three codes, so these definitions suffice for the present discussion.

Note B: The terms horizontal and vertical are used here in a somewhat different way than in the literature where you will find them used referring to directions in a 2-dimensional array of disks. So the second dimension has disks on the axis. In our case, the second dimension is within the disks. We are also using them to refer to the *placement* of parity, not the *direction* in the disk array in which the parity equation is generated.

Figure 1 shows a representation of our notions of element, strip, stripe, stack and array for a typical horizontal code with two parity strips. The middle graphic shows a set of boxes in rows. Each box is a strip containing some set of elements, as indicated by the graphic on the left. Each row is a stripe, a single inter-dependent set of data/parity relations (i.e., a code instance). The strips in each stripe are logically numbered. The collection of stripes is a stack; the strips in each stripe are permuted stripe-to-stripe within the stack to produce uniform distribution of error scenarios when any two disks fail. The array, on the right, consists of a "pile" of stacks that fill the disks' physical capacity.

For notational purposes, we use the following symbols:

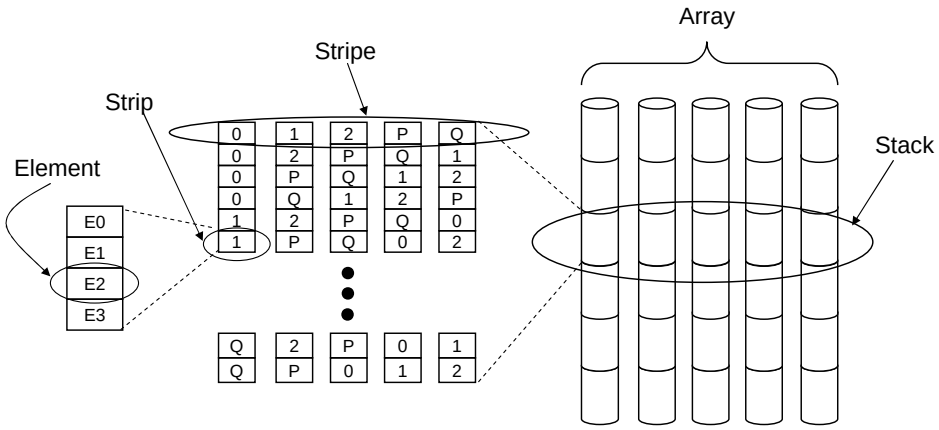

Figure 1: Strips are composed of elements aligned sequentially (vertically) on a disk, stripes are a collection of strips aligned horizontally across disks, stacks are sets of permuted stripes, an array is a collection of stacks,

- n represents the number of strips in a stripe that contain user data.
- q represents the number of strips in a stripe that contain parity data.
- N represents the strip count or total number of strips in the stripe (this is usually called the array size).
- r represents the number of elements per strip. A code with $r \geq 2$ is called 2-dimensional code. We assume that r is the same for every strip in the stripe.
- r_D represents the number of data elements per strip. For a horizontal code, we consider only the data strips so that $r_D = r$; for a vertical code, the quantity $(r - r_D)$ is the number of parity elements on the strip (which we assume is the same for every strip).
- e represents the number of 4KB chunks within an element[1]. We assume that all elements are multiples of 4KB in size.
- S_A represents the strip size, S_D the data substrip size and S_P the parity substrip size (all in number of 4KB chunks). For a horizontal code, $S_A = S_D = S_P = e \cdot r$; for a vertical code $S_A = e \cdot r$, $S_D = e \cdot r_D$ and $S_P = e \cdot (r - r_D)$.

The parameter e needs some explanation. In some of our metrics, we measure memory bandwidth costs. To use absolute costs (say, in megabytes), we would need to map each of our codes to stripes of equal size (in MBs). So the total KBs in a stripe (data and parity) is equal to $4(e \cdot r \cdot N)$KBs. Consequently, this e gives us a uniformizing parameter independent

[1]We represent chunks in units of 4KB as that is a typical IO unit for small host read/write operations.

of actual stripe size. When actually comparing the bandwidth costs, one should select e for each code so that the total stripe sizes are comparable (or equivalently, that the strip sizes $4e \cdot r$ are comparable).

As an example, for RAID4 with 8 strips, we have $N = 8$, $q = 1$, $n = 7$ and $r = 1$ and if we take $e = 64$ we have strip sizes of 256KB. For other codes with $r = 16$, say, we might take $e = 4$ to have an equivalent strip size. RAID5 is obtained from RAID4 by a stack of rotated RAID4 stripes. For horizontal codes, $N = n + q$ and for vertical codes $N = n = q$. Some horizontal codes are 2-dimensional, but all vertical codes are 2-dimensional (by definition).

All the codes we give as examples (see Section 6) are distance three codes, though the methodology can be applied to any distance. There is large variation in r between the codes but for a given code scheme, there is limited flexibility in its choice.

4. Metrics and Models

To define metrics for code comparisons, we need a formal model of the storage system that will use the code. The model selected here is one of many that can be (and has been) the basis for real systems (or subsystems). Other models are possible (for example, a different memory model, redundancy engine, or network model, etc.) and they can affect the final comparisons dramatically. The methodology presented here should provide a framework for adapting the metrics to other models.

In our model described below, we include components for the memory bus and the redundancy computation engine (in our case, an XOR engine; Section 4.1) and for the disks themselves and their utilization (Section 4.2). Our model contains a host interface, but only as a source/target for data; its characteristics are unrelated to the performance of the array code itself so is not measured directly. Our model does *not* contain a CPU, though we do roughly measure (in *XORO*; see Section 4.1) some aspects of CPU utilization. We assume that the CPU has infinite resources (or is relatively inexpensive compared to the other resources we model). With each of the key components in our model, we define metric primitives as the basis of our costs (Sections 4.1 and 4.2). In Section 4.3, we describe in general terms how the cost primitives are used to measure the costs of a specific host operation.

There are many ways to study a given metric. In our case as a simplifying assumption, we concentrate on a randomized model under uniform probabilities, both for host IO workload and for failure scenarios. We study only the expected value or average value of our metrics (see Section 4.4). Finally, we introduce one additional performance metric called Effective Capability (see Section 4.5) where we compare the performance of the system under normal operation (that is, when all components are functioning correctly) versus failure scenarios (when one or more disks are down). In Section 5, we describe the "workloads" we place on the system.

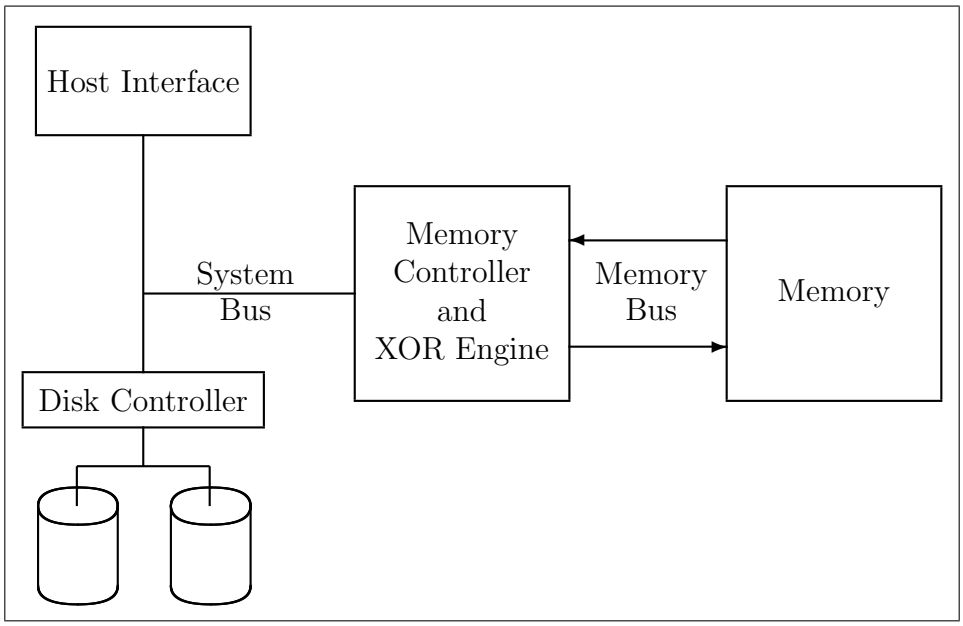

Figure 2: Memory and XOR Engine Model Diagram

4.1. XOR Overhead and Memory Bandwidth Costs

XOR Overhead is a measure of the computational cost of programming an XOR engine to complete all the XOR operations needed for a given task (e.g., computing all the parity during host write of the full stripe). Similarly, a given task will consume some amount of memory bandwidth for subtasks such as moving the data or parity read from disk to memory, sending user data from memory to the host through the host IO interface, or moving data or parity into and out of the XOR engine. To quantify these metrics, we need a simplified model of the system, given in the diagram in Figure 2. Data passing from the disk controller to the memory typically passes through the Memory Controller and XOR engine unmodified. However, the XOR-on-the-fly feature described below can alter the data on the way through.

Suppose we want the XOR engine to compute the XOR of some k (same-sized) chunks of data in memory, and to place the result in some (other) location back in the memory. An instruction to the engine then takes the form of a list of $k+1$ memory addresses and a length (say, in bytes, words or sectors). The first k memory addresses are the source addresses and the last memory address is the target address. The model assumes that the XOR engine has internal buffers where it stores intermediate results. So, if the length is no bigger than the internal buffer size, its operation is to start with an all-zero buffer, and for each source address (in any order), bring the data chunk in from memory and XOR it with the current contents of the buffer (for the first address handled, it need only *replace* the contents, so there is no assumed cost of zeroing the buffer). When the last source address has been

processed, the buffer contents are sent back over the memory bus to the memory itself, at the target address. For lengths exceeding the buffer size, the data is chunked into smaller pieces. This does not affect either the XOR Overhead as we define next, or our cost function for Memory Bandwidth utilization. The model does not preclude the possibility that the target address matches one of the source addresses.

With this model, we define the XOR Overhead for a given single computation of an k-ary equation as

$$XORO(k) = k + 1,$$

the number of addresses in the list[2]. (The only other input is a length; we consider this essentially a constant in the overhead.) For a given multi-step controller task, we measure the XOR Overhead as the sum of the XOR Overheads for all the required XOR computations:

$$XORO = \sum_{xor} XORO(k_{xor}), \tag{4.1}$$

where an XOR instruction *xor* has some k_{xor} source operands. As a consumer of resources, an XOR cost under this model measures computational cost of constructing the list as well as the cost of the XOR engine to retrieve the list and process it. For example, it can be used to contribute to a latency measure if the XOR engine takes some fixed amount of time to move/modify data into its buffer.

The basic model above can be supplemented by a feature called XOR-on-the-fly. Suppose we want to take a data chunk d_0 from the disk, XOR it with some data d_1 in memory and then store the result d_2 in some memory *and* we do not need d_0 for any other purpose. With the above model, we need to read the data d_0 into the memory, then instruct the XOR engine to compute the result and store it back into memory, then throw away the unmodified d_0. This requires the disk data d_0 to pass over the memory bus twice, once when coming in from the disk to memory and a second time from the memory into the XOR engine's xor-buffer. The XOR-on-the-fly feature saves these two transfers. For example, the XOR engine can load its internal buffer with the source data d_1 and as data d_0 is pulled from the disk, it can compute the XOR with the buffer contents and then store the final result d_2 into memory. The disk data d_0 never reaches the memory unmodified.

Memory bandwidth costs are measured in terms of numbers of bytes transferred into and out of memory during some controller task (say, a host write). On the other hand, the XOR overhead is measured in numbers of XOR engine operands, which are sizeless and unitless in our model.

[2]Some XOR engines may only be able to handle a 2-ary operation, that is, two source addresses and one target. This can greatly affect the cost function as applied to various scenarios and codes (the above cost becomes $3(k-1)$ since each XOR has a cost of 3). However, for our purposes, we assume that the XOR engine can handle any (reasonable) number of source addresses.

The above XOR model implies that for a given XOR computation, the number of bytes sent across the memory bus is exactly equal to $(k+1)$ times the length of the XOR operation when we do not have XOR-on-the-fly. With one use of XOR-on-the-fly, this number is $(k-1)$ times the length. That is, for each byte we XOR-on-the-fly, the memory bandwidth utilizations goes down by 2 bytes.

In the sequel, we normalize bandwidth numbers into 4KB chunks (for reasons that become clear later). Consequently, we measure total bandwidth consumed for a given task as:

Without XOR-on-the-fly:

$$\begin{aligned} MBWC \;=\; & \text{Number of 4KB chunks transferred to/from host} \\ & + \text{Number of 4KB chunks read from disk into memory} \\ & + \text{Number of 4KB chunks written from memory to disk} \\ & + \sum_{xor} XORO(k_{xor}) \cdot (\text{length}_{xor} \text{ in 4KB chunks}). \end{aligned}$$

With XOR-on-the-fly:

$$\begin{aligned} MBWC \;=\; & \text{Number of 4KB chunks transferred to/from host} \\ & + \text{Number of 4KB chunks read from disk into memory (not xor-ed on-the-fly)} \\ & - \text{Number of 4KB chunks read from disk and xor-ed on-the-fly into memory} \\ & + \text{Number of 4KB chunks written from memory to disk} \\ & + \sum_{xor} XORO(k_{xor}) \cdot (\text{length}_{xor} \text{ in 4KB chunks}). \end{aligned}$$

The summation here is, as in (4.1), symbolic of a sum over all the XOR engine instructions, where an instruction *xor* has k_{xor} source operands, each of size length_{xor}.

Here is a simple example of our model for host write of 4KB on a typical RAID4 stripe. The host data arrives in memory from the host interface through the Memory Controller/XOR engine. The old data and old parity are read from disk into memory. The XOR engine is instructed to compute the new parity as the ternary XOR of the three 4KB chunks (oldData + newData + oldParity = newParity). This requires giving the XOR engine four addresses (and one length), three as source and one as target for the result. Finally the new parity and new data must be sent to the disk controller. So, we have

$$\begin{aligned} XORO \;&=\; 4 \\ MBWC \;&=\; 1 \text{ (newData from host)} \; + 2 \text{ (read oldData and oldParity)} \\ &\qquad + 2 \text{ (write newData and newParity)} \; + XORO \cdot 1 \text{ (XOR computation)} \\ &=\; 9 \text{ (4KB units).} \end{aligned}$$

With XOR-on-the-fly, we could save 4 units of this data movement. After the host data has arrived in memory, the XOR engine starts the XOR computation by moving the newData

into its buffer, then XORs-on-the-fly both the oldData and oldParity as it comes from the disk, and finally stores the newParity in memory. This has the same number of instructions to the XOR engine ($XORO = 4$) but the $MBWC$ changes to

$$\begin{aligned} MBWC &= 1 \text{ (newData from host) } - 2 \text{ (read/XOR oldData and oldParity)} \\ &\quad + 2 \text{ (write newData and newParity) } + XORO \cdot 1 \text{ (XOR computation)} \\ &= 5 \text{ (4KB units).} \end{aligned}$$

We give the XOR-on-the-fly model here only for completeness and to indicate how the metric values can change to account for this feature. In general, XOR-on-the-fly does not significantly alter the relative comparisons of the different array codes, but only affects the absolute numbers for each code. Consequently, for comparative purposes, we can (and do in what follows) ignore the XOR-on-the-fly model.

Caveat: Our model assumes that all operands for all XOR computations needed for a single operation are of the same length. This both overstates and understates the costs with respect to practical implementations and as such is a reasonable compromise, as we now explain. With some codes in certain operations such as a host write to a full stripe (see Section 5.5), one parity computation can be done by a single XOR formula on strip length units whereas another must be on element length units. This means that our XOR metrics can overstate the actual value. On the other hand, most real systems tend to deal with memory chunks in pages of size much smaller than a strip or even an element. In this case, the XOR operands would have to be fragmented into page sized units anyway, and hence our metric may in fact underestimate reality. For these reasons, we make this simplifying assumption that all operands for the XOR computations in a given operation are of the same length.

4.2. IO Costs

A typical measure of IO costs is number of IO commands to disk. This is roughly equal to the cost of seeks and typically relates to minimum latency. However, this is too coarse in those cases where some IO commands request movement of small amounts of data and others request movement of large amounts of data. The latter consumes resources that factor into memory bandwidth as well as latency and can even dominate seek latency for very large IO sizes.

Consequently, we will use two separate measures for IO costs. First is IOC or IO command count. In the other cost metric, we include the size of data transfer as well. This metric will be called IOE, for IO Equivalent. We use the following rough model for an IO of length $4k$ KB (in multiples k of 4KB):

$$IOE(k) = 1 + \frac{\text{Time to transfer } 4k \text{ KB}}{\text{Average time per 4 KB}}$$

$$= 1 + \frac{4k \text{ KB}/1024}{\text{Avg. media transfer rate in MB/sec}} \cdot \text{Avg. IOs/sec.}$$

The following table gives the result of this formula using typical values for 10K rpm and 15K rpm drives:

	10K rpm	**15K rpm**	**Units**
IOs/sec.	250	333	IOs/sec.
Avg media transfer rate	53.25	62	MB/sec
IOE for k KB	$1 + k/55$	$1 + k/48$	*IOE*

The IOs/sec are based on 8 outstanding commands, quarter-stroke seek range, 18GB capacity for 15K rpm and 36GB for 10K rpm (or more correctly, drives of this generation)[3]. These are idealized numbers to start with and the formula varies only slightly between the different drive types. Consequently, we shall use the more approximate model:

$$IOE(k) = 1 + k/50. \tag{4.2}$$

In the previous two sections, we have described our basic cost primitives. In the next section, we detail how we apply these cost primitives to define a cost function for particular scenarios and modes.

4.3. Cost Model for Operations

For a given use case or host IO scenario (e.g., small write), we would like to provide a general model for computing the cost of a given operation (say, read or write) to a specific chunk of data. Such a cost must also take into consideration not just the failure state but also the specific failure instance (what exactly has failed relative to the chunk on which the operation is acting). Unfortunately, that can be quite difficult in general. In many cases, there are more than one specific algorithm that can be applied to complete a particular operation (e.g., for a write, we might use parity compute or parity increment to update the affected parities). Each algorithm involves different sets of read/write IOs and different XOR computations. It may be the case that one algorithm has lower cost with respect to one metric, and another has lower cost with respect to a different metric. In practice, one would select only one algorithm for each particular combination of operation, chunk and failure instance. That is, we cannot fairly compare minimum *IOE* costs for one code under one algorithm and the *MBWC* costs under another algorithm, particularly, since the algorithm choice may depend on the specific operation, chunk and failure instance combination.

[3]The number of outstanding commands (8) represents a moderately busy, medium-sized storage system with 15–30 disk drives. The quarter-stroke seek range represents a typical workload characteristic.

Furthermore, there are many ways to optimize specific implementations of algorithms to trade one cost against another. For example, if one needs to read (or write) multiple elements in a single strip, it might be (and generally is) less expensive in *IOE* costs to read the entire strip than to read each piece separately. But this increases the $MBWC$. Of course, it would be even more efficient to read the smallest portion of the strip that covers the required chunks, but that level of refinement is too detailed for our purposes.

With these remarks in mind, we define the following precise cost model. The notation Cost(c, op,f) will refer generically to some cost function Cost for the operation "op" acting on chunk c during the failure instance f. For example, "(c,op,f)" may be "write (op) of the first logical strip (c) in a stripe, when the P-parity strip is down (f)".

Now suppose we have a specific algorithm to implement a given operation, to a given chunk c under failure scenario f. Each such algorithm breaks down into the following steps (some of which do not occur in certain cases):

1. a data chunk c comes in from the host;
2. data and/or parity is read from some strips (or parts of strips);
3. XOR operations are performed;
4. data and/or parity is written to some strips (or parts of strips);
5. a data chunk c is returned to the host.

Only one of the first and last steps can occur (the first for writes, the last for reads). The first/last and fourth steps are essentially independent of the particular algorithm used to implement the operation (or more precisely, can be optimized independent of the second and third steps). So, the variability of costs comes in the read and XOR steps.

Now, suppose the algorithm requires reading certain data and/or parity from a set of strips and writing data and/or parity to a set of strips. Let rd_1 be the number of such read strips where the IOs touch exactly one element within the strip and let rd_2 be the number of such read strips where the IOs touch at least two elements. Similarly, we use the notation wr_1 and wr_2 for one and at least two write elements per strip. Based on the remarks above, we assume that for each strip counted in rd_1 we read only the minimal size (with one seek). For those IOs that touch the rd_2 strips, we amortize them into a single IOC to the entire strip (or data or parity substrip if that can be determined). We do the analogous modeling for wr_1 and wr_2.

With this, we have the following general formulas

$$XORO(c,\text{op},f) = \sum_{xor} XORO(k_{xor}) \tag{4.3}$$

$$IOC(c,\text{op},f) = rd_1 + rd_2 + wr_1 + wr_2 \tag{4.4}$$

$$IOE(c,\text{op},f) = rd_1 \cdot IOE(er) + rd_2 \cdot IOE(sr) + wr_1 \cdot IOE(ew) + wr_2 \cdot IOE(sw) \quad (4.5)$$

$$MBWC(c,\text{op},f) = |c| + rd_1 \cdot er + rd_2 \cdot sr + XORO \cdot \min(e,|c|) + wr_1 \cdot ew + wr_2 \cdot sw \quad (4.6)$$

where the sum in (4.3) is as in (4.1), $|c|$ is the length of the host IO, er is the size of subelement chunks we read, ew the size of subelement chunks we write, sr the size of the substrips we read, and sw the size of the substrips we write (all in multiples of 4KB). This assumes that $er \leq e$ and $ew \leq e$. Typically, $er = ew$ and these are either equal to $|c|$ when $|c| \leq e$ and equal to e when $|c| > e$. The $XORO$ term in $MBWC$ is the number of operands for the XOR computation, assuming each operand is of length either an element e or a subelement.

For example for a single 4KB host read to a good strip, we have $|c| = 1$, $er = 1$, $rd_1 = 1$, and all other parameters are zero. The (best) parity increment algorithm for a single 4KB host write to a good strip has $|c| = er = ew = 1$, $rd_1 = wr_1 = 1 + DoutD$, and $rd_2 = wr_2 = 0$ (recall the definition of $DoutD$ in Section 3).

To select an algorithm for a Cost(c, op,f) measurement, we always pick that algorithm that minimizes the read contribution to IOE (the first two terms of (4.5)). This is for the following reasons. First, by the remarks above, it is only the read/XOR steps that vary between algorithms (the host-side steps and write steps are constant). Second, this read term will contribute a large portion of the costs of IOE and $MBWC$. In fact, IOE variations will be dominated by this read term. In $MBWC$, we have two competing terms: the contribution from IOE and from $XORO$. In general, the former will dominate the latter (since the former acts on strip size chunks of data and the latter only on element-sized chunks (or less)). Finally, IOE reflects typically more expensive system operations (in latency, etc.) than does $MBWC$, so we choose the algorithm that minimizes IOE.

There is one additional remark concerning this model. Suppose, for example, that we are doing a write to a small chunk of the strip, but have to update two or more parity elements (actually subelements) on the same strip. Our model says that we will update the relevant parity subelements, then write the entire parity strip. This presupposes that we have already have in memory the balance of the strip that we do not update. We could increase the read costs to gather this old parity in order to ensure that it is present in memory or cache, but instead we simply *assume* that this is the case. This may not be entirely equitable, but it simplifies the calculations and is not an unreasonable assumption for a real system.

4.4. Averaging Model

For a given use case or host IO scenario (e.g., small write), we want to compare codes under uniformly distributed operations on an array (multiple stacks of stripes), under each

of its stripe modes (Normal, Degraded and Critical). We also assume uniformly distributed failure cases, but conditional on a given failure mode. The stack notion allows us to do the measurements by simple counting and averaging on a single stripe. We do this as follows.

Suppose that the operation in question acts on a set of chunks C of the user data substripe and that such chunks partition the user data. For example, for long writes the chunks are the data substrips, or for small writes the chunks might be 4KB chunks of user data (aligned on 4KB boundaries) both of which partition the stripe as well (by our assumptions).

We average the costs for a given operation, "op", over all failure cases F with the formula:

$$\frac{1}{\#C\#F}\sum_{c\in C}\sum_{f\in F}\mathrm{Cost}(c,\,\mathrm{op},f)$$

where Cost(c, op, f) is the cost of the operation "op" on the chunk c assuming the specific failure case f as defined in Section 4.3.

When in Normal mode, there are really no "failure" cases, but only one state, so F has one element (and can be ignored in the sum) to yield the formula:

$$\frac{1}{\#C}\sum_{c\in C}\mathrm{Cost}(c,\mathrm{op})$$

where now Cost(c,op) is the cost of that operation on the specific chunk c, when the stripe is all read/writable.

When Degraded, the set F consists of N elements, where each f corresponds to loss of one strip. When Critical, the set F consists of $N(N-1)/2$ elements, where each f corresponds to loss of two strips.

Note: There are, of course, other statistics besides average (expected value) that can be computed and used to compare codes. For example, the maximum value of any cost is also one that may be relevant to a system designer, as it may predict whether the system can even handle the code at all. Other distribution information may also be important. However, we do not consider other statistics here, and stay, for the sake of simplicity with just the average.

However, we do observe that the averaging process itself can hide some worst-case anomalies. Consider, for example, a host read-type scenario. In normal mode, costs are bare minimum (e.g., $XORO = 0$). In a failure mode, a read to "good" data also has minimal costs, yet a read to "lost" data will incur additional costs for reconstruction. But there may be relatively few such "lost" cases or few with excessively high costs, so that the overall average is within limits. We encourage the designer using these principles and methodologies to extract the statistics that are most relevant to their system constraints.

Finally, we emphasize that our probability models do not relate to likelihood of failures, that is, with reliability issues; we deal only with uniform IO distributions and, in a given failure mode (that is, conditional on a failure mode), uniform distribution of failed strips within that failure mode.

4.5. Effective Capability

We will apply the metrics $IOC, IOE, XORO, MBWC$ to a variety of use cases (see Section 5) under different stripe states (Normal, Degraded and Critical). The "Normal" state reflects the performance characteristics of the stripe (or stack or array) under optimal operating conditions. The **Effective Capability** metric is designed to predict the performance *variation* of an array in the various states relative to the Normal condition as viewed from the host's perspective. We apply this to IOE, as "Effective IO Capability", as that is considered the dominant factor that limits host IO performance. The normalization can also be applied to the $MBWC$ for example, if desired. This metric can be applied to any host driven use case.

The metric is defined for a specific use case as the ratio of the average (or expected value) of the IOE cost under Normal mode to the average under the Degraded or Critical mode, expressed as a percentage:

$$\text{Eff IO Cap} = 100\% \cdot \frac{\text{average } IOE \text{ Normal mode}}{\text{average } IOE \text{ current mode}} \tag{4.7}$$

This metric has two dimensions: mode (Degraded or Critical) and IO scenario or use case. (For Normal mode, the value is 100% and so is not interesting.) The averages are computed over a stack, or equivalently, an array fully populated with stacks (see Section 4.4).

As an example, consider the case of Short Reads or Writes (see Sections 5.2-5.1). For reads in Normal mode, the average $IOE = IOE(1)$. Consequently, the inverse of the average IOE (times 100%) in Degraded or Critical mode provides the formula for effective read capability in those modes. For writes in Normal mode, the optimal[4] IOE for distance d is $2d \cdot IOE(1)$ (this comes from applying the parity increment algorithm that does a read-modify-write to $d-1$ parity and read of old and write of new data). So, the formula in this case would look like:

$$\text{Eff Write Cap} = 100\% \frac{2d \cdot IOE(1)}{\text{average } IOE \text{ current mode}}$$

Note that it is possible for this value to be larger than 100%: in a horizontal code with all parity strips down, the average write $IOE = IOE(1)$ (but this is only one of the many Critical substates within a stack).

This metric is primarily designed for comparing a given code with itself (under failure mode versus Normal mode). It may be used for comparative purposes of different codes but *only if* the costs under Normal mode are comparable. In Section 7.5 we say more about this issue as well as provide an alternative definition designed for comparative application.

[4]This statement about optimal IOE is true only for codes of high efficiency. For example, for the LSI code (see Section 6.7) of efficiency 1/2 and distance 3, the average $IOE = 4IOE(1)$ in Normal mode.

5. Use Cases

The primary operations of a storage system are reads and writes of user data. To simplify the picture dramatically, these IOs come in only a few coarse types: small (typically about 4KB for a random workload), long (about 256KB for sequential IO pattern) and very long (say, 1–2MB). The long size approximates a value that is a cross-over point for striping of a typical RAID system. The very long size is typical of a RAID system where a cache sitting above the RAID layer may accumulate a very large number of blocks of new data before sending it into the RAID system.

In this section, we examine a set of host IO scenarios or use cases that model these coarse IO types. For each use case, we explain how each of our metrics are computed. Each use case has subcases as well. These depend on the state of the stripe: what elements are read/writable and what have been lost due to disk loss or hard errors in the disks. They also depend on whether the host IO is targeted to a "good" chunk of data (read/writable) or "lost" (on a failed strip). For simplicity, we assume only the three failure modes of Normal, Degraded and Critical (full strip losses) and not the more complex cases of multi-sector medium errors across the stripe. Appendix A provides some detailed examples of the calculations for numerous use cases.

5.1. Short Write

The term **Short Write** is defined as a uniformly random host write to a 4KB chunk of user data within the stripe (or stack or array). We assume that the IO is 4KB aligned within the element it touches. So, in our model formulas (4.5) and (4.6), the values $|c| = er = ew = 1$ (one 4KB chunk). There are generally two algorithms (parity increment and parity compute) that one can apply to a host write scenario. In some cases, the choice between the two is clear, but in others, the selection is more subtle.

Suppose that the affected element E has some $0 \leq m \leq DoutD(E)$ good parity elements that need to be updated as well (there is strict inequality here if any affected parity strip is lost). These parity elements may or may not all lie on separate strips. However, we can assume that they do *not* lie on the same strip as E. If E is good, then wr_1 includes a term for writing out the 4KB subelement of E; if E is lost, then wr_1 only contains terms for parity update. In either case, $wr_1 + wr_2$ is the number of write strips, partitioned by single or multi-element writes. Recall that sr and sw are the size of the substrips we need to read and write, respectively, for multi-element strips. Only parity elements (if at all) will require multi-element strip writes so $sw = S_P$. We now break down the subcases for short writes to good and lost elements, and parity increment versus parity compute.

Good: Suppose the affected element E is on a good strip. As observed, the number $(wr_1 - 1) + wr_2$ is the number of strips with parity elements that need updating.

- For a parity increment algorithm and $m \geq 1$, we have $rd_1 = wr_1$ and $rd_2 = wr_2$ since we have to read the equivalent data that we write. We also have $sr = S_P$ (only read parity during multi-element strip reads). If $m = 0$, clearly $wr_1 = 1$ and $wr_2 = rd_2 = 0$ and $rd_1 = 0$; we do not need old data in this case as well.

  For XOR costs, when $m \geq 1$, there are again two subalgorithms. We can compute each new updated parity by

  $$\text{oldData} + \text{newData} + \text{oldParity}_j = \text{newParity}_j \tag{5.1}$$

  or we can compute oldData + newData = dataDelta and then

  $$\text{dataDelta} + \text{oldParity}_j = \text{newParity}_j. \tag{5.2}$$

  The former has costs $XORO = 4m$ and the latter $XORO = 3 + 3m$; the former is less expensive if $m < 3$ which includes the case $m = 0$.

- For the parity compute algorithm, we need to read all the dependent data for all the good parity subelements touched by this element and write all the parity subelements touched by the element. So rd_1 and rd_2 now account only for data strips and $sr = S_D$. Note that wr_1 includes writing the new data.

  In addition,

  $$XORO = \sum_{j=1}^{m} (PinD_j + 1) = T + m, \tag{5.3}$$

  where $PinD_j$ is the parity in-degree of newParity$_j$ and $T = \sum_{j=1}^{m} PinD_j$.

  Of course, it is assumed here that all the dependent data in the parity formulas are on good strips (as in the case when the stripe state is Normal). If not, then parity formulas must be generalized to remove their dependency on these lost elements (effectively finding reconstruction formulas for these elements). In our metrics, we assume that these costs will necessarily exceed the costs of parity increment, so we do not try to pursue this analysis.

Lost: Now suppose the affected element is on a lost strip. It must be the case, then, that $m \geq 1$ (or else we have unrecoverable user data). We again consider the parity increment versus the parity compute algorithms.

- For parity increment, we need to, in effect, reconstruct the lost old data before applying

the increment algorithm. Assume that there is some L-ary XOR formula[5] that can be used to reconstruct the lost data from some known data and known parity elements. If $m \geq 2$, the formula

$$\sum_{j=1}^{L} \text{depDataParity}_j + \text{newData} = \text{dataDelta}$$

(of $XORO = L+2$) together with the the computations in (5.2), yield a total $XORO = L + 2 + 3m$. If $m = 1$, then we might use the formula

$$\sum_{j=1}^{L} \text{depDataParity}_j + \text{oldParity}_1 + \text{newData} = \text{newParity}_1.$$

However, in this case, there must be at least one parity term in the initial sum equal to the oldParity_j term. This means that the oldParity_j term cancels for a total cost $XORO = L + 1$. Note that such a formula may contain parity elements that are not directly touched by the new data, but are needed for reconstructing the old data.

In these cases, $sr = S_A$ (the full strip, since we expect to read both data and parity subelements, if applicable, from any of the rd_2 strip).

- For parity compute, we apply the standard parity compute algorithm for an $XORO$ cost as in (5.3). However, in this case, we only need to read data, so $sr = S_D$.

To compute these costs for a given code under the failure cases, we need to first determine the state of the element we affect, then the set of good parity elements it touches. Next, we need to determine the number of multi-element reads that are required for the parity increment and parity compute algorithms (if they both apply). We then select the algorithm with lowest derived read IOE. Finally, we use the special subformulas above to compute the full IOE and $MBWC$ costs for that selected algorithm according to (4.5) and (4.6).

We summarize these subcases in Table 1. See Appendix A.1 for examples.

[5] We do not describe precisely how the L-ary XOR formula for reconstruction is derived. For Degraded mode, typically, a standard RAID4 algorithm is sufficient. In Critical mode, most codes have some recursive approach for reconstructing lost elements (see also Sections 5.3, 5.4, 5.8). One approach for generating the necessary formula in this Short Write use case (and Short Read below) may be derived from the recursion by combining all the intermediate equations and removing dependence on data or parity elements that appear an even number of times in the formulas. For example, suppose c and e are lost elements, a, b, d are known and the recursive relations are $a + b = c$ and $c + d = e$. Then $a + b + d = e$ combines these two formulas into a single formula depending only on known elements (by removing the dependence on c).

Element	# Good Parity	Algorithm	sr	$XORO$
good	$m = 0$	n/a	n/a	0
	$m \geq 1$	PC	S_D	$T + m$
	$1 \leq m \leq 2$	PI	S_P	$4m$
	$m \geq 3$	PI	S_P	$3m + 3$
lost	$m \geq 1$	PC	S_D	$T + m$
	$m = 1$	PI	S_D	$L + 1$
	$m \geq 2$	PI	S_A	$3m + L + 2$

Table 1: Summary of Short Write costs, where m is the number of good parity elements affected by the updating element, L is the in-degree of the minimal XOR formula to recompute the lost element, $T = \sum_{j=1}^{m} PinD_j$ over all the good parity elements and $PinD_j$ is the parity in-degree for the jth parity, and sr, S_A, S_D and S_P are as above.

5.2. Short Read

A **Short Read** is a uniformly random host read to a 4KB chunk fully aligned within an element. This case is significantly simpler than the previous case, but much of the notation and notions of the previous section apply.

Since no strip is changed for a host read operation, we always have $wr_1 = wr_2 = 0$ and the parameter sw is not relevant. The term $|c|$ in our base formulas (4.5) and (4.6) is $|c| = 1$.

Good: If the affected element is good, then there is only one reasonable algorithm, namely, read the element and return to the host. So $XORO = 0$, $rd_1 = 1$, $rd_2 = 0$ and the other parameters are not relevant. We find that $MBWC = 2$ and $IOE = IOE(1)$.

Lost: For a lost element, we have to reconstruct that element. This means that (as for the parity increment algorithm for Short Write) we must find the best L-ary XOR formula of data and parity elements that can be used to reconstruct the lost data. This means that, in some cases, $rd_2 > 0$ and that $sr = S_A$ when it is relevant.

The $XORO$ costs are given by the reconstruction formula and so $XORO = L + 1$. Note that the total number of good parity elements is not relevant; only the parity subelements required for the reconstruction formula are needed.

See Appendix A.2 for examples.

5.3. Strip Write

A **Strip Write** is a uniformly random host write to a complete aligned strip (or data substrip in the case of vertical code) in a stripe. As above, we need to consider the cases of a good strip or a lost strip and the subcases that depend on state of the stripe.

Suppose the affected strip S has $0 \leq m \leq \sum_{E \in S} DoutD(E)$ total good parity elements touched by all the elements in S. These all must be updated as well. As before, we can assume that parity elements all lie on strips other than S. Because S is a strip, wr_1 contains terms only for writing single parity elements. If S is good, then wr_2 contains a term for writing out the full strip S of size S_D and terms for writing out multiple parity elements on the same strip of size S_P. This means that sw may not be constant (e.g., for a vertical code). If S is lost, then wr_2 contains terms only for writing parity strips of size S_P.

In all cases, the XOR operations contribute to $MBWC$ a term equal to $XORO \cdot e$, since each XOR is on data chunks of size equal to an element.

Good: Suppose the strip S is good. For the parity increment algorithm, we need to read the old data on strip S (so a term in rd_2 of size S_D), plus read all the m affected old parity. As for Short Write, there are different algorithms for updating the parity. Note that it is possible that multiple updating elements touch the same parity element, so this must be accounted for as well.

Suppose there are t_j updating data elements that touch the jth parity element affected. One algorithm precomputes all the dataDelta for the r_D elements in S (at a total cost of $3r_D$) and then updates the parity element as

$$\sum_{i=1}^{t_j} \text{dataDelta}_i + \text{oldParity}_j = \text{newParity}_j.$$

The total $XORO$ cost is then

$$\sum_{j=1}^{m} (t_j + 2) + 3r_D = T_I + 2m + 3r_D, \tag{5.4}$$

where T_I is the subtotal, restricted to the data elements being updated, of the total parity in-degree of all the affected parity. Alternatively, compute each new parity element using both old and new data elements (and old parity) together for a total cost of

$$\sum_{j=1}^{m} (2t_j + 2) = 2T_I + 2m. \tag{5.5}$$

The latter is less expense if $T_I < 3r_D$ and we always select the one of minimum cost.

For a parity compute algorithm, we have to find all the dependent data elements, find which strips they lie one (singly or multiply with affected parity), determine from this the costs rd_1 and rd_2, then compute the total parity in-degree T_C for all the affected parity, for an $XORO = T_C + m$, as for Short Write (now accumulated over a larger set of parity, typically). If in fact, any dependent data elements are lost, we assume that parity compute will necessarily be too expensive and abandon this algorithm.

Lost: When the strip S is lost, our work for parity compute is quite similar to the above case, with the exception that we do not have to write out the new data, only the new parity.

For parity increment, we need to find formulas for reconstructing the lost data. In some cases, this is quite simple (e.g., in Degraded mode, this is typically not much more complex than RAID4 algorithm). However, in Critical mode, there is no obvious and general approach to this since other lost data may implicitly be needed.

Consequently, we perform the following algorithm to find the reconstruction formulas[6].

1. Let R represent the list of reconstruction formulas for the elements of S that have been implicitly reconstructed; initialize R to the empty set.
2. While all elements of S have not been implicitly reconstructed (i.e., while $\#R < \#S$):
   (a) Find an element in S and not represented in R that has a reconstruction formula that depends only on known data and parity elements and the elements reconstructed by the formulas in R.
   (b) Implicitly reconstruct this element and append its formula to the set R.
3. Output the set of reconstruction formulas R.

The output formulas represent reconstruction formulas for all lost elements of S using only known data and parity elements and (iteratively) elements of S. These formulas can be inserted into the formulas for computing dataDeltas or newParitys (as replacements for oldData). The set of known data and parity elements in these formulas together with the affected list of parity elements (those that need updating) determine the read costs for this parity increment algorithm.

The XOR costs are determined by the algorithm that computes the dataDelta for each element, the total cost of which is $2r_D$ plus the total weight of all the output formulas in R plus $2m + T_I$ as we now show. Suppose the ith formula in R has weight ℓ_i; then the cost of computing dataDelta for this element is $\ell_i + 2$ (that is, ℓ_i plus one for newData plus one for output dataDelta). So the total cost of computing all dataDeltas is

$$\sum_{i=1}^{r_D} (\ell_i + 2) = L + 2r_D. \tag{5.6}$$

[6]This is a general description of reconstruction methodologies for some codes. E.g., the X-Code (see Section 6.6) in Critical mode always has at least one element on a lost strip one of whose diagonal does not touch the other lost strip. We recover that, then use it to bootstrap recovering an element on the other strip, then back to the first strip. Our algorithm here logically leaves out explicit reconstruction of the element on the other strip.

From these, the total cost of the Strip Write *XORO* is

$$T_I + 2m + L + 2r_D \tag{5.7}$$

where the term $T_I + 2m$ is the cost of computing the new parity element from the dataDeltas as in (5.4).

The formulas in R together with the parity increment formulas easily tell us the read numbers rd_1 and rd_2; in addition, we have $sr = S_A$, since we generally need both data and parity elements from multi-element strips.

We summarize these subcases in Table 2. See Appendix A.3 for examples.

Strip	Algorithm	*XORO*
good	PI	$\min(T_I + 2m + 3r_D, 2T_I + 2m)$
	PC	$T_C + m$
lost	PI	$T_I + 2m + L + 2r_D$
	PC	$T_C + m$

Table 2: Summary of Strip Write costs, where m is the number of good parity elements affected by the updating element, $L = \sum_{i=1}^{r_D} \ell_i$, the sum over all updating elements of the in-degrees of the minimal (recursive) XOR reconstruction formulas, $T_I = \sum_{j=1}^{m} t_j$, the sum over all updating parity elements of the in-degree of each limited to the updating data elements, and $T_C = \sum_{j=1}^{m} PinD_j$, the sum over all updating parity elements of the parity in-degree.

5.4. Strip Read

A **Strip Read** is a uniformly random host read to a complete aligned strip (or data substrip in the case of vertical codes). As for Short Read, this largely depends on state of the stripe, and, if Degraded or Critical, which strips are lost.

Good: In any mode, either Normal, Degraded or Critical, if the requested strip is not lost, then $IOC = 1$, $IOE = IOE(S_D)$, $XORO = 0$ and $MBWC = 2S_D$.

Lost: When the read strip is lost, then the data on that strip needs to be reconstructed. For Degraded mode, the reconstruction is typically as in RAID4 and involves one parity element per lost data element, though possibly more than one known data element per strip. For Critical mode, we need to reconstruct lost data elements as efficiently as possible via the algorithm given in the previous section. So, we find that $XORO = L + r_D$ (we only need old data). We will have $sr = S_A$ just as for Strip Write. The other costs can be easily derived from *XORO* and the read pattern of known data and parity.

As for Strip Write, the XOR contribution to *MBWC* is $XORO \cdot e$.

From these remarks, we see that the greatest variation in costs between the different codes will come from *XORO* (as the major contributor to *MBWC*). A slight variance will occur in *IOE* and *IOC* in some cases.

The costs for Strip Read can be used to directly determine costs in Degraded mode for rebuilding data strips (but not for any lost parity); this is described in Section 5.7.

See Appendix A.4 for examples.

5.5. Full Stripe Write

A **Full Stripe Write** is a host write to the complete data substripe of a uniformly random stripe within the array (or stack). The costs here again depend on state of the stripe (Normal, Degraded or Critical) and what strips are down. Since this involves no read IOs (all the data is new), there are three metrics to consider.

The *XORO* is the cost of a parity compute (for any parity element on a writable strip). This is the sum of the *XORO* for each parity computation, over all the good parity:

$$XORO = \sum_j (PinD_j + 1) = T + m.$$

This sum may in fact be empty (e.g., a horizontal code with two parity strips lost).

The *IOE* is the cost of writing the full stripe (the writable portion, anyway) so we have

$$IOE = (N - x)IOE(S_A),$$

where $x = 0, 1, 2$ is the number of lost strips.

Finally, the *MBWC* is given by

$$MBWC = n \cdot S_D + XORO \cdot e + (N - x) \cdot S_A$$

since we pay $n \cdot S_D$ in moving the stripe's user data from the host interface into memory, then $XORO \cdot e$ in bandwidth costs computing the parity, and finally $(N - x)S_A$ in writing all the data and parity elements to the "good" strips in the stripe.

As with Critical Rebuild (Section 5.8), the *MBWC* have significant variation between codes almost solely because of the *XORO* term (assuming each stripe's user data size, as well as their strip size, are comparable).

5.6. Full Stripe Read

In Normal mode, the costs are reading the data portion of the stripe and sending it to the host interface. In Degraded mode, we have to read essentially all of the stripe (perhaps excluding some of the parity), compute a lost data strip (if any), then send the full stripe of

data to the host. For Critical, we have to read the entire stripe (good elements), reconstruct any lost data strips, and send the full stripe data to the host. Consequently, the differences between codes here will be entirely reflected in the cost of rebuilding lost strips, and these costs are covered below. Hence, we give no metrics for this case.

5.7. Degraded Rebuild

The term **Degraded Rebuild** refers to reconstruction of all lost data (and any lost parity) on a lost strip when one strip is lost. This is essentially equivalent to (or derivable from) the Degraded Strip Read costs when the strip contains data. The only differences are (a) reconstructed data flows to a new disk instead of to the host and (b) any lost parity elements must be reconstructed and stored to disk. In horizontal codes, difference (b) only applies when the lost disk contains only parity and this is not covered in the Strip Read case, however, a simple parity compute algorithm can be applied in this case. In vertical codes, (b) applies only to typically one (sometimes two) parity elements on the lost disk, so these costs are easily determined from the parity compute formulas.

5.8. Critical Rebuild

The term **Critical Rebuild** refers to reconstruction of all lost data and/or parity when two strips are lost. The associated metrics can be used to derive total rebuild costs, including time (assuming a time cost model for the system), but we do not do that here.

It should be clear that in this scenario, our read costs generally (but not always) include the entire stripe (good portion) so equals $(N-2)IOE(S_A)$ and our write costs are for two full strips $2IOE(S_A)$. Consequently, total IO cost is

$$IOE = N \cdot IOE(S_A).$$

Recall that the absolute strip size equals $4S_A$KB. Consequently, for codes with comparable strip sizes, the IO costs are essentially equal.

Also, $MBWC$ includes a term $N \cdot S_A$ for reading and writing data to disk plus the contribution to bandwidth costs from the XOR computations of $XORO \cdot e$. As a result, the dominant metric here is $XORO$. There is no simple formula for $XORO$ as it is very code dependent and there is large variation between codes.

The above formulas and remarks apply generally to MDS codes (or highly efficient ones). The LSI code (see Section 6.7) has different characteristics which are quantified in the tables in Appendix B.

As for the case of Strip Write or Read in Degraded mode, we need to find formulas to reconstruct any lost data elements. As before, these can be determined by an algorithm similar to that in Section 5.3, with the modification that the expression S represents all the lost data elements (from both strips).

6. Codes

In this section we describe the various array codes that we compare with our metrics. We give only distance three versions of these codes. Our descriptions will be somewhat terse and provide only a simple example in each case, typically done graphically by showing data layout on strips within a stripe together with some parity equations, where such equations are needed to complete the description. The reader is referred to the original sources for more complete descriptions.

We also introduce a notion of the "ideal" code (Section 6.1), an imaginary code that provides the optimal characteristics in all dimensions, but primarily efficiency and *IOE* for short write. This is provided as a theoretical benchmark and a means for normalization of metrics under a uniform model.

Our list of codes is by no means exhaustive. The point here is to give a fairly rich subset of codes which we can use to illustrate our metrics.

We add some loose terminology here to help distinguish between the general characteristics of these codes. We call a code "geometrical" if the parity computations can be visualized by geometric patterns overlaid on the data layout in the stripe. This should be contrasted with "combinatorial" codes, where the parity is computed by combinations of elements in the stripe for which there is no obvious visual pattern.

In addition, we need to specify the strip size S_A for each code (since we assume some divisibility conditions on element size by 4KB and strip size by number of elements). The values we choose provide approximately equal strip sizes (in KBs) for all considered codes and are reasonably realistic for the system model defined in this paper. There are clearly other choices for strip size and the metric values can change under different assumptions, but again, our focus is on the methodology, so that system designers have the tools needed to compare and select codes in the context of their particular constraints.

We have chosen strip counts per stripe of $N = 8$ and $N = 16$, as well as strip sizes approximately 240KB (for $N = 8$) and 256KB (for $N = 16$). The strip counts are typical for real systems in use today. The strip size was chosen as a reasonable value for large sequential disk IO.

6.1. The Ideal Code

The "ideal" code is an imaginary code that, for a given distance, provides the optimal characteristics and metrics in all dimensions simultaneously and is first and foremost an optimally efficient code for a given strip count and distance.

For distance d, the ideal code is a one-row horizontal code, with $q = d - 1$ parity strips, $n = N - q$ data strips and has the following characteristics:

1. Parity in-degree is n for all parity.

2. Data out-degree for any *chunk of data* in a strip is $d-1$, and furthermore the size of the parity chunks touched matches that of the data chunk.

3. Any n chunks of identically aligned and sized chunks of data or parity (a horizontal slice through the stripe) can be used to reconstruct the remaining $d-1$ aligned/sized chunks of data or parity.

We consider here only the case where $d = 3$.

The third assumption means that any lost chunk of data or parity can be reconstructed from any n equal sized and aligned chunks of good data and/or parity. That is, all reconstruction formulas (and parity compute formulas by the first assumption) have n summands. Another way to view this is that data and parity are interchangeable from a computational point of view. Unfortunately, such a code based on XOR is not known (and may not exist, see [6]). But as mentioned, it can provide a theoretical benchmark as well as a means for normalization of the different classes of metrics into a uniform model.

There are (at least) two ways to conceptualize this code. First, imagine a RAID4 code with two mirrored parity strips. This has the ideal costs in all the metrics (except that it is *not* distance three!). For example, a Short Write 4KB of data touches only that much of the data strip and exactly 4KB of parity on each parity strip. That is, all reads and writes to any strip are of the size of the host request (so that the parameters e_r, e_w, s_r, s_w in equations (4.5)-(4.6) all equal $|c|$ and $rd_2 = wr_2 = 0$). As another example, a Full Stripe Write would send each of the data strips into the ideal XOR engine once for each parity that needs updating (so XOR cost in this case is exactly $q(n+1)$).

The other conceptualization is as a Reed-Solomon code. This has the ideal properties in many respects, but the parity computations are not XOR-based. In some sense, the ideal code could be realized with a Reed-Solomon code and an ideal XOR engine that can compute finite field arithmetic as easily as XOR.

We will give numbers for the ideal code with $\langle n, S_A \rangle = \langle 6, 60 \rangle, \langle 14, 64 \rangle$ for strip counts $N = 8, 16$ and strip sizes $240, 256$KB, respectively.

6.2. Blaum-Roth Code

The Blaum-Roth (or BR) codes [6] are a family of LDPC codes theoretically optimal in that regard for binary (XOR) codes. These codes are combinatorial in nature and are horizontal codes with two parity strips. They are defined by two parameters p, a prime, and n, the number of data strips in the stripe with the requirement that $n \leq p$. These codes are best described by their systematic generator matrices; they have the form (this notation is

somewhat different from that in the citation):

$$\left(\begin{array}{c|c|c|c||c|c} I_{p-1} & 0_{p-1} & \cdots & 0_{p-1} & I_{p-1} & I_{p-1} \\ \hline 0_{p-1} & I_{p-1} & \cdots & 0_{p-1} & I_{p-1} & Q_{p-1}^{(1)} \\ \hline \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \hline 0_{p-1} & 0_{p-1} & \cdots & I_{p-1} & I_{p-1} & Q_{p-1}^{(n-1)} \end{array}\right)$$

where, I_{p-1} is a $(p-1)$-dimensional identity matrix, 0_{p-1} is a $(p-1)$ dimensional all-zero matrix and for $1 \leq j \leq n-1$, we define $Q_{p-1}^{(k)} = \left(a_{i,j}^{(k)}\right)_{0 \leq i,j \leq p-2}$ with

$$a_{i,j}^{(k)} = \begin{cases} 1 & \text{if } j \neq p-1-k \text{ and } i-j = k \pmod{p} \\ 1 & \text{if } j = p-1-k \text{ and either } i = k-1 \text{ or } 2i = k-2 \pmod{p} \\ 0 & \text{otherwise.} \end{cases}$$

For example, with $p = 3$, the generator matrix is

$$\left(\begin{array}{cc|cc|cc||cc|cc} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{array}\right)$$

We will give the metrics for the cases where $\langle p, n, S_A \rangle = \langle 7, 6, 60 \rangle$ and $\langle 17, 14, 64 \rangle$ to cover cases of strip counts $N = 8, 16$ and strip sizes 240, 256KB, respectively.

6.3. EvenOdd Code

The EvenOdd (or EO) code was described in [5] (among other places). It is a 2-dimensional horizontal and geometrical code with one parity strip computed by standard RAID4 computation (laterally across the stripe) and the other along diagonals through the stripe.

The code has defining parameters p, a prime, and a number $n \leq p$. The number of rows is $r = p-1$ and the number of *data* strips is n. The strip count is $N = n+2$. The following layout example for $p = 3, n = 3$ shows the basic construction.

S_0	S_1	S_2	P	Q
$d_{0,0}$	$d_{0,1}$	$d_{0,2}$	P_0	Q_0
$d_{1,0}$	$d_{1,1}$	$d_{1,2}$	P_1	Q_1

(6.1)

This stripe logically has an additional $p-1$st row (3rd in the example) that is all zero. Here, the parity is computed by the formulas: for $0 \le i \le p-2$,

$$P_i = \bigoplus_{j=0}^{p-1} d_{i,j}$$

and

$$S = \bigoplus_{j=0}^{p-1} d_{p-1-j,j}$$

$$Q_i = S \oplus \bigoplus_{j=0}^{p-1} d_{i-j,j}$$

where the first subscript is taken modulo p and we assume that $d_{p-1,j}$ is zero for all j. That is, each P-parity is computed as a row parity; each Q-parity is computed as an UP-diagonal parity, with the "main diagonal" S that starts in row $p-1$ (with a logically zero data element) embedded in each of the other Q-parity elements.

When $n < p$, simply assume that the last $p-n$ data strips are logically zero. The result is that the upper index in each sum is replaced with $n-1$.

Our results will include the cases of $\langle p, n, S_A \rangle = \langle 7, 6, 60 \rangle, \langle 17, 14, 64 \rangle$ of strip counts $N = 8, 16$ and strip sizes $240, 256$KB, respectively. (This is just as for Blaum-Roth).

6.4. Row-Diagonal Parity Code

This code from Network Appliance, Inc., called the Row-Diagonal Parity code (or RDP), is described in [9] and in US Patent publication [10]. It claims to improve on the EvenOdd code described above in Section 6.3. This is also geometrical, but the pattern is slightly different from EvenOdd. As for EvenOdd, the parameters are p, a prime, and n the number of data strips. For this code, $n \le p-1$ (not p as for EvenOdd). The data/parity layout is exactly as in (6.1) but we logically assume that the last data strip (S_{p-1}) is all zero; that is, $d_{i,p-1} = 0$ for all i. Also, the formula for Q_i are different:

$$Q_i = P_{i+1} \oplus \bigoplus_{j=0}^{p-1} d_{i-j,j}.$$

In effect, this formula replaces the data element $d_{i-(p-1),p-1} = d_{i+1,p-1} = 0$ (logically zero) from strip S_{p-1} with the P-parity element P_{i+1} from that row and furthermore removes the term S.

Geometrically, this is the same UP-diagonals as in EvenOdd, but instead of computing the "main diagonal" S and storing it embedded in all the other diagonals, each diagonal includes the P-parity column in its computation. That is, Q-parity is computed on the combined strips

of data and P-parity (not just on the data portion). As such, it is a *concatentation* parity code (UP-diagonals) computed on top of another parity code (the row P-parity).

The restrictions, compared to EvenOdd, allow only for one fewer data strip in the stripe for a given p. As before, to get a configuration for $n < p - 1$, assume more data strips are logically zero.

Here, we use the cases where $\langle p, n, S_A \rangle = \langle 7, 6, 60 \rangle, \langle 17, 14, 64 \rangle$ and strip sizes $240, 256$KB, respectively. (These match the parameters sets considered for EvenOdd and Blaum-Roth.)

6.5. BCP Codes

The BCP codes, described in US Patent [2], are not so much codes as a methodology for constructing vertical codes. We give one example below for strip count $N = 16$. Curiously, the methodology does not apply directly to $N = 8$ strips, so in that case we give an alternate construction. The codes generated by the methodology in the patent have both geometric and combinatorial characteristics. They have some rotational symmetry, which allows for some visualization, though the basic rotated pattern is rather random looking (and so more combinatorial).

Besides the strip size, the codes have only one defining parameter, an even number N, corresponding to both the strip count and the data strip count (so $n = N$). The number of rows is always $r = N/2$. We give two examples with $\langle N, S_A \rangle = \langle 8, 60 \rangle, \langle 16, 64 \rangle$ and strip sizes $240, 256$KB, respectively.

Here is a construction for $N = 8$ (we place the parity elements in the last row, though most examples we have seen in the literature have it in the first row). Each data element touches $DoutD = 2$ parity elements in the last row. On the left picture, we show the mapping from each data element into its first parity element by labeling the data element by the lower case letter of its parity element label; in the right picture, we do the same for the second parity element:

S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
x	w	v	x	v	t	u	u
v	v	t	t	u	s	w	t
t	s	s	s	t	u	s	s
S	T	U	V	W	X	Y	Z

S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
y	y	y	z	x	v	x	w
w	z	w	y	y	w	z	x
u	x	z	u	z	z	v	y
S	T	U	V	W	X	Y	Z

When the BCP code is defined with rotational symmetry, there is an alternative (and shorter) description of the pattern (this is the case for all $N \neq 8$). For the parity element P_0 in strip S_0, we specify a list, ordered by strip number, of the row indices for elements that are touched by this parity element. We use an $*$ to indicate a skipped strip. All other parities are computed by rotational symmetry (e.g., rotate the list to the right for P_1 on S_1).

The $N = 16$ construction is given for P_0 by the pattern

$$P_0 = [*, 0, *, 3, 5, 5, 0, 6, 1, 4, 3, 6, 4, 2, 1, 2]$$

and other parities are computed by rotational symmetry. So this indicates that the elements in strip1-row0, strip3-row3, etc., touch parity element P_0.

The ZZS codes [32] provide an alternative code construction very similar to a BCP construction in essentially all metrics, but only in the case where $N = p - 1$ for an odd prime p (for example, $N = 12, 16$). The ZZS code typically has strip count $N = p$, but they have an all-data strip that can be logically assumed to be zero to give strip count $N = p - 1$. Similarly, the BCP codes can have an additional all-data strip appended to get a code metrically similar to the ZZS code.) As a result, we do not give any ZZS examples (the curious reader is welcome to apply the metrics in this paper to ZZS).

6.6. X-Code

The X-Code [31] is another vertical code, though it differs from ZZS and BCP in that this code has two rows of parity elements whereas the others have only one row. Both rows of parity elements are computed from diagonals (one up, one down) through the stripe, and so this family of codes is geometrical.

These codes depend on a parameter p, an odd prime. The number of data strips is exactly p and the number of rows $r = p$ as well. Of these rows, two are parity rows and $p - 2$ are data rows.

An example for $p = 5$ is given here. The left picture shows the pattern of computing the first row of parity elements and the second shows the pattern for the second row of parity elements. Each parity element is indicated by an upper case letter; such a parity element is computed as the XOR of the set of data elements labeled by the corresponding lower case letter.

S_0	S_1	S_2	S_3	S_4
v	w	x	y	z
w	x	y	z	v
x	y	z	v	w
Y	Z	V	W	X
$*$	$*$	$*$	$*$	$*$

S_0	S_1	S_2	S_3	S_4
v	w	x	y	z
z	v	w	x	y
y	z	v	w	x
$*$	$*$	$*$	$*$	$*$
X	Y	Z	V	W

Since this code always has an odd number of strips, we will give the results for $\langle N, S_A \rangle = \langle p, S_A \rangle = \langle 7, 68 \rangle, \langle 17, 70 \rangle$, the nearest values to our basic strip counts of $N = 8, 16$ and strip sizes $280, 272$KB, respectively. Because each strip contains parity, it is *not* possible to change the stripe size by logically zeroing a data strip.

6.7. LSI Code

LSI Logic Corp. presents a code in US Patent [30]; we call this the LSI code for convenience. It is a simple, one-dimensional geometrical pairing code, with distance three, and efficiency equal to mirroring (RAID1).

The basic layout is the following (on 8 strips):

S_0	P_0	S_1	P_1	S_2	P_2	S_3	P_3
d_0	$d_0 \oplus d_1$	d_1	$d_1 \oplus d_2$	d_2	$d_2 \oplus d_3$	d_3	$d_3 \oplus d_0$

The strips alternate data and parity, each parity strip contains the XOR of the data on each side, the stripe logically wraps around on itself. This code works with any *even* number N of strips with $N \geq 6$.

This is the only code in our set of examples that can survive loss of some but not all combinations of more than two strips (for N large enough), but it is also the only one that is not optimally efficient. We give metrics for the cases of $\langle N, S_A \rangle = \langle 8, 60 \rangle, \langle 16, 64 \rangle$ and strip sizes $240, 256$KB, respectively.

7. Comparisons

We are now in a position to present the comparative results for our metrics and codes.

We concentrate here on the key differences between the codes; Appendix B provides a complete listing of all the raw values (for both $N = 8$ and $N = 16$ strip counts) and Appendix C contains some distribution data on frequencies of good/lost element counts for simple element reconstruction (e.g., for Short Read). In the next subsection, we go over the highlights from the raw data; in the following subsections, we give more comprehensive summaries organized by use cases.

7.1. Cost Comparisons Highlights

In this section we will present the highlights from those tables in Appendix B that show significant or noticeable differences between the codes. As can be seen from these tables, there are very few differences between the codes for host read operations (either Short Read or Strip Read) in both Normal and Degraded modes. The lack of difference in the Normal mode case occurs since there are no errors and data is just read from the strips without any additional costs. For Degraded mode, the codes generally function as if they were each a simple RAID4 code on $N-1$ strips; the extra parity is not required for simple reconstruction.

For Short and Strip Reads or Writes, there is noticeable variation in the costs for all four metrics ($IOC, IOE, XORO, MBWC$). However, we believe IOE is a more informative metric (as it more accurately reflects system performance) than IOC. Recall that $MBWC$ are computed primarily from disk IO-induced costs (indirectly reflected in IOE) and by

XORO costs. By examining the raw data, it is clear that for some large host operations (e.g., Strip Read/Write and Full Stripe Write) there are huge discrepancies in *XORO* though significantly smaller discrepancies in *MBWC*. This is because, in general, there is a trade-off between many *XORO* operations on small (element size) data chunks versus a small number of *XORO* operations on relatively large (strip size) data chunks. So the total contribution to *MBWC* can balance out. For example, for $N = 16$ and a Strip Write in Critical mode, the Blaum-Roth code requires $XORO = 170.23$ and $MBWC = 1176.40$ on average whereas the Ideal code requires only $XORO = 8.70$ but $MBWC = 1052.27$. Since we feel that *MBWC* more accurately models consumption of limited system resources (at least more so than *XORO*), we will not present summary data for *XORO*. However, the large discrepancies in this metric do provide some separation between the codes, so we encourage the reader to review the raw *XORO* data.

In addition, for Full Stripe Write and Rebuild, there is essentially no difference between the codes in Normal or Degraded mode (with the exception of *XORO* as noted above). In Critical mode, as one might expect, the *IOE* costs are essentially identical between the codes. So, in these use cases, we summarize only the *MBWC*.

In almost all cases, the LSI code does exceptionally well, even better than the Ideal code in many instances. For example, for a Short Write in Normal mode (and $N = 16$), LSI's $IOE = 5.10$ and $MBWC = 12.00$ whereas for the Ideal code these values are 6.12 and 15.00, respectively. This is for three related reasons: (a) $PinD = 2$ for all the parity elements, so reconstruction formulas, data/parity requirements, etc., are all reduced, (b) $PinD = 2$ also implies that the parity compute algorithm is more efficient than parity increment in almost all write use cases, saving typically one *IOC*, (c) the one-row or one-dimensional nature of the code allows the LSI code to avoid any multi-element read/writes (more precisely, the alignment property of the Ideal code holds for the LSI code). However, all of these performance gains come at the cost of efficiency (50% vs 87.5%). This highlights and quantifies one aspect of the efficiency performance trade-off (as one also sees with RAID1 versus RAID4). As these remarks apply to most use cases, we refrain from repeating them in what follows, with exceptions noted below.

In the next subsections, we give summary charts of the key metrics, organized according to these remarks. For brevity, we only concern ourselves with the $N = 16$ case, though similar remarks apply to other strip counts.

7.2. Short and Strip Write Costs Comparisons

Figures 3 to 6 show the relative costs for *IOE* and *MBWC* for all codes we tested on $N = 16$ strips for the Short Write and Strip Write use cases. We discuss the highlights of each table in turn.

Short Write: We observe in Figures 3 and 4 that generally all the codes are comparable for Short Write operations with the following exceptions:

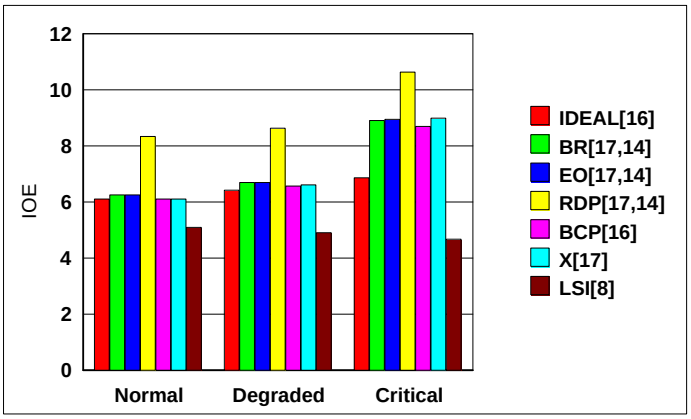

Figure 3: Short Write *IOE* comparison, all modes.

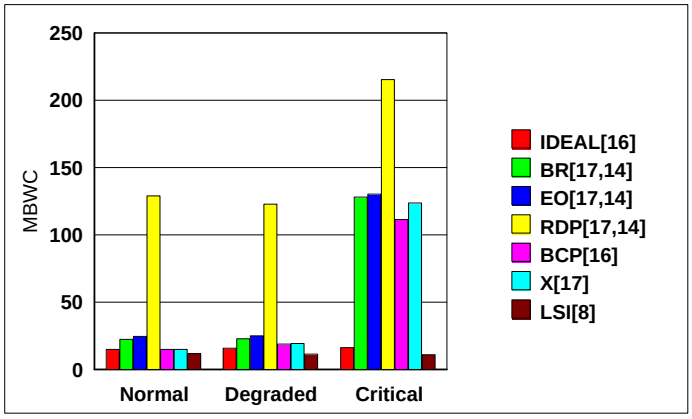

Figure 4: Short Write *MBWC* comparison, all modes.

- The Row-Diagonal Parity code has significantly more *IOE* (around 8.3–10.6 versus 6.1–9.0) than the other codes in all modes. Similarly, for *MBWC*, the Row-Diagonal Parity code is much more expensive, by a factors of 675%, 560% and 175% in Normal, Degraded and Critical modes, respectively. This is because there are more data elements that touch three parity elements (versus the optimal two parity elements) and that these extra parity elements are on the same strip. That is, a Short Write almost always invokes a full strip parity read/write. For EvenOdd and Blaum-Roth, this happens significantly less often. For the vertical codes and the Ideal code, it never happens.

- In Critical Mode, all codes are substantially more expensive than the Ideal code. This

is for similar reasons to the previous observation, namely, these codes trigger full strip read costs for reconstruction formulas. The Ideal code does not suffer from this burden by design (see item 3 in Section 6.1).

Strip Write: In a Strip Write a large number of parity elements need to be updated. For the vertical codes, these parity elements are on many different strips and thereby induce significant *IOC* costs (the dominant term in *IOE* for small lengths) — see Table B.9. For the horizontal codes these parity elements are all on two strips allowing their read/writes to be amortized into fewer *IOC*s. As a result, we see in Figure 5 that the vertical codes are more expensive in *IOE* in all modes (by factors of 2.5–3.0). Furthermore, even in Degraded mode for vertical codes, a write to a lost strip will typically require a read of most of the entire strip.

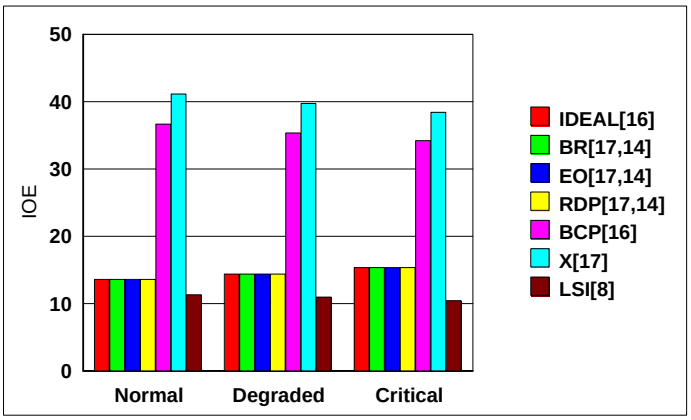

Figure 5: Strip Write *IOE* comparison, all modes.

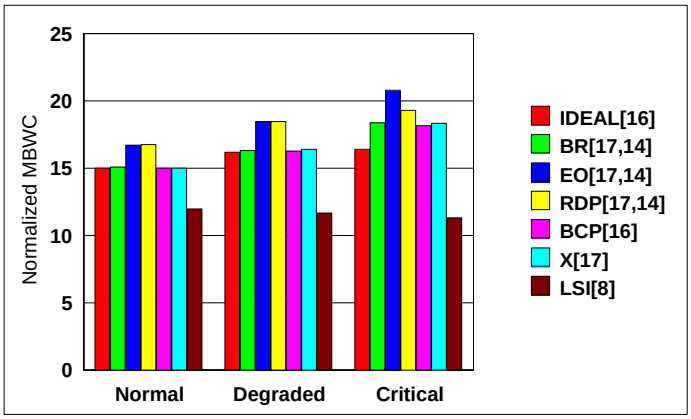

Figure 6: Strip Write Normalized *MBWC* comparison (normalized by user data strip size), all modes.

For *MBWC* in Figure 6, we observe that the codes are generally quite comparable, though the EvenOdd and Row-Diagonal Parity codes have somewhat larger values. This difference results from the fact that these codes are *not* LDPC (have too many ones in the generator matrix). This inflates *XORO* costs (see Table B.9), which in turn contributes to *MBWC*, though the effect is not excessive. We see this also in the case of Full Stripe Write and Rebuild (see Section 7.4).

Finally, it should be noted that the *XORO* (Table B.9) for BCP code is about $1/3 - 1/2$ that of the other codes, though the *MBWC* are more comparable. That is because in the BCP codes the elements are twice the size (half as many rows) as the other codes: fewer but larger chunks moved through the XOR Engine.

7.3. Short and Strip Read Costs Comparisons

In Figures 7 and 8, we show the relative costs for *MBWC* for all efficient codes we tested in Critical mode (on $N = 16$ strips). As we noted above, for Normal and Degraded modes, all the codes are very similar. Similarly, only *MBWC* shows significant differences; *IOE* is essentially the same in all cases.

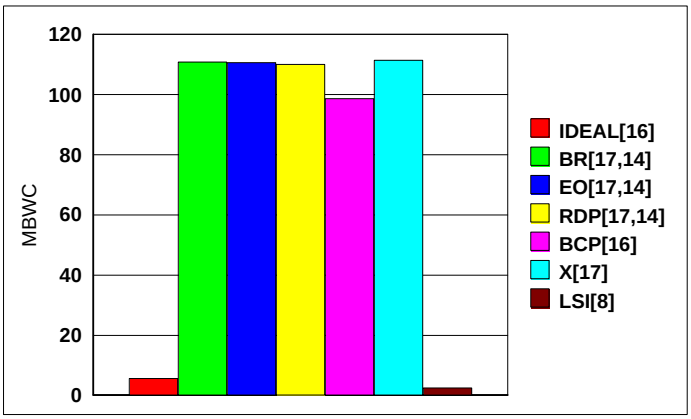

Figure 7: Short Read *MBWC* comparison, Critical mode.

Short Read: In this use case, the *IOE* costs (in Critical mode, see Table B.8) for all codes are very comparable (in the range of $4.59 - 4.70$) with the exception of the Ideal code with value 2.68 and LSI with value 1.15. This suggests that codes with multiple rows are in general more costly because even a small host read can trigger costly multi-element strip reads.

For *MBWC*, we observe from Figure 7 that generally all the codes are comparable for Short Read operations (with the very notable exception of the Ideal code and the LSI code). The BCP code does a bit better because it is LDPC; the X-Code, though LDPC, is somewhat larger because it is on $N = 17$ disks and has longer strip sizes.

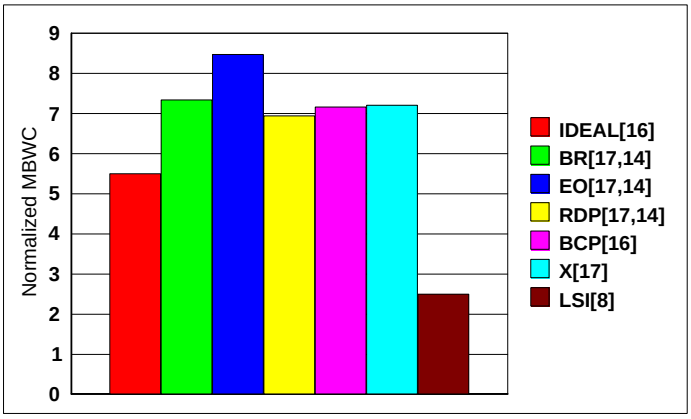

Figure 8: Strip Read Normalized *MBWC* Comparison (normalized by user data strip size), Critical mode.

Strip Read: For Strip Read (see Table B.10), the *IOE* costs are essentially equal for all efficient codes (about 5.85–6.11) because in this case, one typically needs to read all the available data and parity. (Note these numbers are less than $N-2=14$ because the average also includes the cases of reading from "good" elements where the $IOC=1$.)

In Critical mode as seen in Figure 8, the EvenOdd code has noticeably more *MBWC* in Strip Read than the other codes. This is primarily because of the extra *XORO* costs of reconstructing multiple chunks of lost data (see Table B.10). For the other codes (excluding the Ideal and LSI codes), the reconstruction formulas seem to have better "recursive" properties. Recall that the methodology for strip reconstruction is to recover one element at a time, using the recovered elements to help defray the reconstruction costs of the subsequent elements. For the EvenOdd code, it appears that there is less of an advantage in this defraying of costs.

7.4. Full Stripe Write and Rebuild Costs Comparisons

Figures 9 and 10 show the relative costs for *MBWC* for all efficient codes we tested in Critical mode (on $N=16$ strips).

We make the following observations about this data:

- In Full Stripe Write, the EvenOdd and Row-Diagonal Parity codes have higher *MBWC* costs. This is because these codes are rather far from LDPC; that is, the total weight of all the parity computations is rather high, particularly compared to Blaum-Roth and the vertical codes. Curiously, this weight is the same for both EvenOdd and Row-Diagonal Parity, though they distribute that extra weight in different ways (EvenOdd overloads some data elements in lots of parity elements, while Row-Diagonal Parity overloads lots of data elements into relatively few parity elements each).

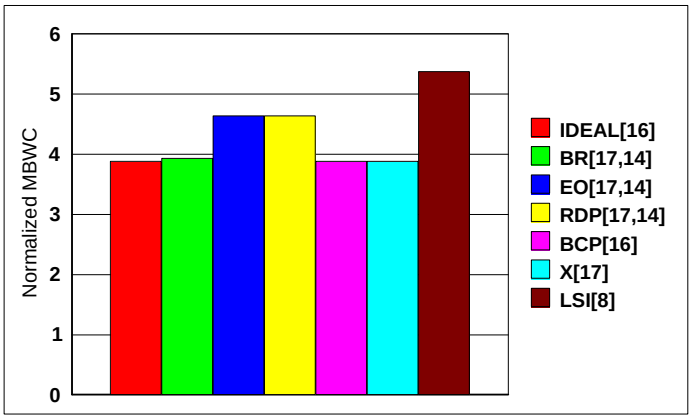

Figure 9: Full Stripe Write Normalized *MBWC* comparison (normalized to user data stripe size), Critical mode.

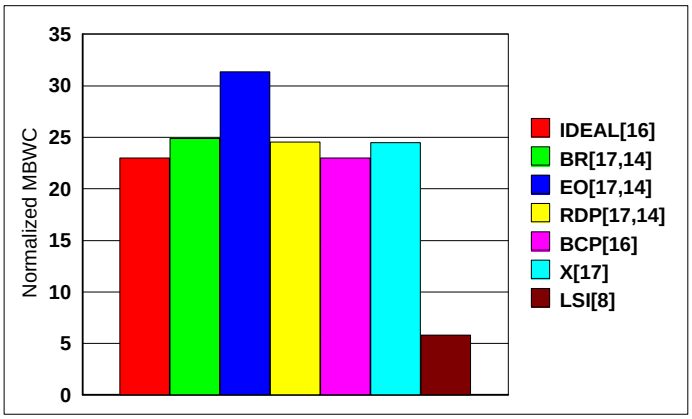

Figure 10: Rebuild Normalized *MBWC* comparison (normalized to strip size), Critical mode.

- Also in Full Stripe Write, The LSI code has much higher normalized MBWC because it writes approximately the same amount of total data, but the efficiency means that a significantly smaller portion (about 50% versus 87.5% for the other codes) is user data – that is, a lot of work for less net gain. This is the only metric where the LSI code does not excel.

- In Rebuild, the BCP code has the same *MBWC* as the Ideal code. This is because it has the same *IOE* costs (as all codes do, see Table B.11) but optimum memory bandwidth consumption for *XORO* costs. These are fragmented versus the Ideal code, but the total data movement is the same. That is because the BCP code has the optimal data out-degree (this is the counter-point to the observation above). The X-

code also has this property but has 17 strips, which explains the relative difference with the BCP code.

- Also in Rebuild, the EvenOdd code's normalized $MBWC$ stands out (approximately 33 versus 25 for both Row-Diagonal Parity and Blaum-Roth). This is seen from its inflated $XORO$ costs (about 50% more than these two codes, see Table B.12). This again suggests, as remarked above for Strip Read, that "recursive" reconstruction process for EvenOdd is less efficient than for the other two codes.

7.5. Effective IO Capability

The Effective IO Capability metric is useful to determine the change in performance of an array from the Normal mode (with all disks functioning correctly) to a failure mode (with one or two disks failed).

When applied to use cases where all the codes have generally comparable IOE under Normal mode (in particular, Short Read and Strip Read), the Effective IO Capability can provide a comparative measure between codes. From the raw data (see last column of Tables B.8 and B.10), we see that only the LSI and Ideal codes stand out (and the latter only for Short Reads in Critical mode). The data does show that one should expect only 55% and 22% throughput in Degraded and Critical mode for Short Reads for all the codes except the Ideal code and the LSI code. For the Ideal code, this is 55% and 38%, respectively. In particular, this shows a significant drop off in Critical mode from the Ideal code for the Short Read use case. For the LSI code, these numbers are 94% and 89%, quantifying the performance/efficiency trade-off.

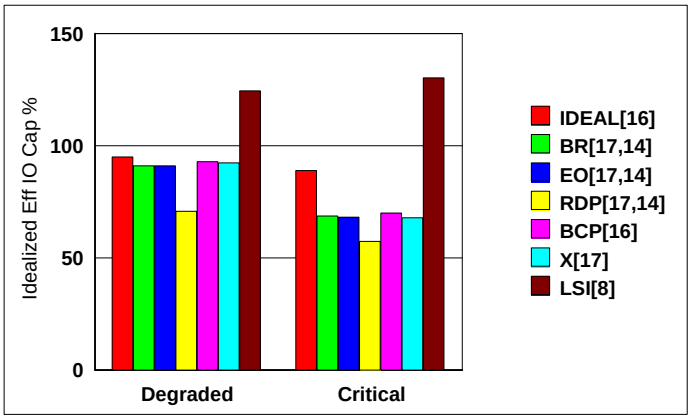

Figure 11: Short Write Idealized Effective IO Capability.

For use cases where the codes are significantly different under Normal mode (e.g., Short Write where Row-Diagonal Parity and LSI codes stand out on opposite sides of the norm, and Strip Write), using the raw Effective IO Capability is somewhat of an unfair comparison.

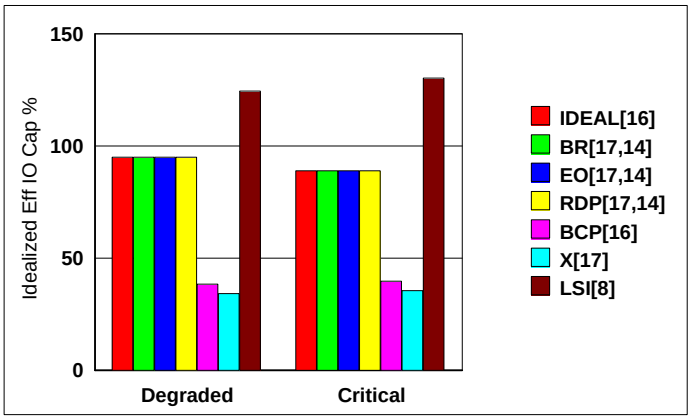

Figure 12: Strip Write Idealized Effective IO Capability.

For example, in Short Write and Critical mode (see Table B.7), the Row-Diagonal Parity's value is 78% where the other horizontal codes are 70%. For Strip Write, the BCP code and X-code exhibit even more change with value 107% in Critical (versus the norm of 89%). These are due primarily to the relatively large value of *IOE* for these codes in Normal mode. Consequently, care must be taken when comparing these values for different codes.

To mitigate this effect, we define a revised version of the Effective IO Capability, called "Idealized". In this definition, we renormalize the formula in (4.7) by replacing the numerator with the term for the Ideal code as follows:

$$\text{Idealized Eff IO Cap} = 100\% \cdot \frac{\text{average } IOE \text{ Normal mode Ideal code}}{\text{average } IOE \text{ current mode}} \tag{7.1}$$

For Read operations, formulas (4.7) and (7.1) agree. In Figures 11 and 12 we show the Idealized Effective IO Capability for Short Writes and Strip Writes. Note that in these tables large values indicate better performance.

We see in the case of Short Write from Figure 11 that all codes do somewhat poorly with respect to the Ideal code with a larger relative drop in Critical mode versus Degraded mode; the Row-Diagonal Parity code falls even farther below all other codes. In Strip Write (Figure 12), the horizontal codes are comparable to the Ideal code, but the two vertical codes fall significantly below this norm.

We observe from Table B.11 that all codes have generally equivalent Effective IO Capability for the Full Stripe Write case with values 107% (Degraded) and 114% (Critical), showing that the codes should actually improve performance with failure. This is expected because failures typically mean there is less to write (fewer strips on line).

As always, the LSI code does very well because it has very good Normal mode numbers and better numbers in Degraded or Critical mode. Note that the Short and Strip Write cases are analogous because the code has only one row.

8. Discussion

In this section, we give some broad-stroke conclusions from the observations and remarks of the previous section.

- Vertical codes should not generally perform well in scenarios where Strip Writes are common, because of the dispersion of the parity elements across many disks. Their *XORO* costs are generally better because of the LDPC property.

- Of the horizontal codes, the Row-Diagonal Parity code is not well-suited to Short Write scenarios because of the over-abundance of data elements that touch more than minimum number of parity elements. The EvenOdd code has expensive Rebuild. Of the three, the Blaum-Roth code is (overall) better than either of these – this is certainly because of its LDPC property.

- The Ideal code stands out only in Short Read and Writes. This is certainly a consequence of the parity alignment assumption we make for the Ideal code (item 3 in Section 6.1). In all other cases, these summary charts show it is relatively similar (with the possible exception of Strip Read in Critical). But these all hide the *XORO* costs which are significantly less expensive (and this relative difference explains to a large part the exception for Strip Read).

- All efficient codes have less than "Ideal" performance in Short Reads and Writes in Critical mode. This cost is borne by the excess *IOE* to account for multi-element strip accesses.

- Though we do not give the numbers here, our measurements show that codes with many rows are predicted to perform on average worse than similar codes on fewer rows. For example, one could try EO$\langle 17, 6\rangle$ for a code on $N = 8$ drives, but the results would be worse than for EO$\langle 7, 6\rangle$. This is primarily because of the *XORO* costs, but is also reflected in excess parity element touches, and effects similar to those we have observed above.

- As the LSI code exemplifies, there is clearly a design trade-off possible between performance and efficiency. These metrics provide some quantitative measures that can be used in such design choices.

As we noted in Section 6, our measurements were done assuming moderate sized strips (about 240–256KB) as a reasonable value for large sequential IOs. Our host logical block addressing model was within a strip, then strip-to-strip, then stripe-to-stripe.

The high costs of *IOE*(strip) for these codes suggest that alternative layouts of a code on disk may be desirable. Given that 4KB is a typical minimum IO size, perhaps one should

explore implementations where elements are each 4KB. In yet another option, as is the recommended implementation of Row-Diagonal Parity [9], strips of size 4KB may be used (this assumes that the number of rows is a power of 2). In this latter case, the host Short IOs become Strip IOs where all the codes are more comparable.

These alternative layouts do have some trade-offs. To map codes to smaller element or strip sizes, one either has to map logical host IO addresses across strips at shorter boundaries or map them across stripes in order to service large host IOs from/to one disk (assuming that is a requirement). The following graphic shows this distinction. The numbers in the cells represent logical host addressing in multiples of 64KB. The "P" and "Q" represent the two parity strips (as if for a horizontal code). The chart on the left shows logical addressing that stays within stripes first (as we did in our computations). In this version, the largest sequential write to a single disk would be at most 64KB and a 256KB host write to logical addresses 0–3 would cover one stripe as a Full Stripe Write scenario. The chart on the right shows logical addressing across stripes. Here, a host write of 256KB (say, to addresses 0–3) spans one disk only for the user data, but it crosses the four stripes, incurring the update costs, particularly *XORO*, on four independent strip writes.

Stripe0	0	1	2	3	P0	Q0
Stripe1	4	5	6	7	P1	Q1
Stripe2	8	9	10	11	P2	Q2
Stripe3	12	13	14	15	P3	Q3

Host Addressing Within Stripes

Stripe0	0	4	8	12	P0	Q0
Stripe1	1	5	9	13	P1	Q1
Stripe2	2	6	10	14	P2	Q2
Stripe3	3	7	11	15	P3	Q3

Host Addressing Across Stripes

Expanding this picture so that each chunk was 4KB with 64 stripes, we see a combinatorial expansion in costs proportional to the number of stripes. Admittedly, this can be ameliorated by some clever programming, but there would still remain a high level of fragmentation.

In short, when selecting a code and comparing codes by metrics similar to those defined in this paper, one must first look at the host IO use case model, and then perhaps examine a variety of mappings of host addressing to stripes to find the code and mapping that best fits the expected use cases.

9. Summary

In this paper, we have defined a set of performance metrics ($IOC, IOE, XORO, MBWC$ and Effective Capability) for storage system array codes as a basis for quantitative comparisons. The IO metrics are based on a realistic model of drive performance (with parameters that can be modified for other generations of drives). The *XORO* and *MBWC* metrics are based on a model of a processor with a separate XOR engine as part of the memory control unit. This is typical of many hardware based RAID5 systems. The Effective Capability

metric measures the performance variation from Normal to failure state for a given erasure code. It can also be normalized (using the theoretical Ideal code) for comparisons. We did not include in our model any costs for dual-headed failover scenarios as these design points are orthogonal to the actual array/erasure code issues.

We explained how these metrics are computed for a variety of use case scenarios, under reasonable assumptions about code layout in strips and stripes on disks in an array. We computed these metrics for a variety of known array codes with differing qualitative and quantitative characteristics (e.g., both horizontal and vertical, efficient and non-efficient) and including a notion of the Ideal code for normalization. We summarized the results of the computations, describing where certain codes standout (either for better or for worse) and what contributed to these exceptions. We drew some general conclusions about qualitative characteristics of codes that can be used as guidelines for code design and selection, under the metrics and layouts we defined.

Our broader goal for this paper was to provide a methodology or framework for a system designer to define the type of fine-grained metrics exemplified here, but in a manner more suitable to his/her specific system. We encourage designers to adapt the IO, memory, and XOR engine models, etc., as well as the mapping of host logical addressing and other variables, to better reflect their system and usage model. As an example, suppose the designer's system has no XOR engine external to the processor, but has, say, an MMX-type engine within the processor. Now assume that all the data for the XOR engine is in the processor's cache and need not traverse the memory bus multiple times. In this case, the XOR contribution to memory bandwidth costs might be modified significantly. Looking farther ahead, alternatives to disk drives such as MRAM, when they become practical, will alter basic system architectures dramatically. Such alternative models will make significant changes to the conclusions one might draw about what codes are "good" and "bad" for a particular system. Along these lines, we hope in the near future to apply this methodology to the analysis and comparisons of erasure codes in the context of network-based redundancy.

Acknowledgements

The authors would like to thank Bruce Cassidy, Steve Hetzler, Jeff Hartline, Jai Menon, Shmuel Winograd, Joe Hyde, Ian Judd, Karl Nielsen, John Fairhurst, Andy Walls and Richard Freitas (in no special order) for many helpful contributions to this work including support, probing discussions on various issues related to this work in particular and storage systems in general.

References

[1] E. Anderson, R. Swaminathan, A. Veitch, G. Alvarez, and J. Wilkes. Selecting raid levels for disk arrays. In *Conference on File and Storage Technologies (FAST)*, pages 189–201, 2002.

[2] S. Baylor, P. Corbett, and C. Park. Efficient method for providing fault tolerance against double device failures in multiple device systems, January 1999. U. S. Patent 5,862,158.

[3] E. R. Berlekamp. *Key Papers in the Development of Coding Theory*. IEEE Press, New York, NY, 1974.

[4] E. R. Berlekamp. *Algebraic Coding Theory*. Aegean Park Press, Walnut Creek, CA, 1984.

[5] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: an efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computers*, 44:192–202, 1995.

[6] M. Blaum and R. M. Roth. On lowest density MDS codes. *IEEE Transactions on Information Theory*, 45:46–59, 1999.

[7] A. Brown and D. A. Patterson. Towards availability benchmarks: A case study of software RAID systems. In *Proceedings of the USENIX Technical Conference*, pages 263–276, 2000.

[8] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26:145–185, 1994.

[9] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure. In *Proceedings of the Third USENIX Conference on File and Storage Technologies*, pages 1–14, 2004.

[10] P. Corbett, S. Kleiman, and R. English. Row-diagonal parity technique for enabling efficient recovery from double failures in a storage array, January 2001. U. S. Patent Application US2001000035607.

[11] R. G. Gallager. *Low-Density Parity-Check Codes*. MIT Press, Cambridge, MA, 1962.

[12] G. A. Gibson, L. Hellerstein, R. M. Karp, R. H. Katz, and D. A. Patterson. Failure correction techniques for large disk arrays. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–132, Boston, MA, 1989.

[13] J. H. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, 2003.

[14] I. N. Herstein. *Topics in Algebra*. John Wiley and Sons, New York, NY, 1985.

[15] R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications*. Cambridge University Press, Cambridge, UK, 1994.

[16] M. Luby. LT codes. In *Proceedings of the 43rd Annual IEEE Symposium on the Foundations of Computer Science*, pages 271–280, 2002.

[17] M. G. Luby, M. Mitzenmacher, A. Shokrollahi, and D. A. Spielman. Efficient erasure correcting codes. *IEEE Transactions on Information Theory*, 47:569–584, 2001.

[18] D. J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. `http://www.inference.phy.cam.ac.uk /mackay/itprnn/`.

[19] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. Northholland, Amsterdam, The Netherlands, 1977.

[20] P. Massiglia. *The RAID Book*. The RAID Advisory Board, Inc., St. Peter, MN, 1997.

[21] J. D. McCalpin. Sustainable memory bandwidth in current high performance computers. `http://home.austin.rr.com/mccalpin papers/bandwidth/bandwidth.html`.

[22] N. K. Ouchi. System for recovering data stored in failed memory unit, May 1978. U. S. Patent 4,092,732.

[23] D. Patterson, G. Gibson, and R. H. Katz. Reliable arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD Conference*, pages 109–116, 1988.

[24] J. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software: Practice and Experience*, 27:995–1012, 1997.

[25] F. Preparata and M. Shamos. *Computational Geometry: an Introduction*. Springer-Verlag, 1988.

[26] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8:300–304, 1960.

[27] A. Shokrollahi. Raptor codes, 2003. preprint.

[28] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation*. A. K. Peters, Ltd., Natik, MA, 1998.

[29] Storage Performance Council. SPC benchmark. `http://www.storageperformance.org`.

[30] A. Wilner. Multiple drive failure tolerant raid system, December 2001. U. S. Patent 6,327,672 B1.

[31] L. Xu and J. Bruck. X-code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory*, pages 272–276, 1999.

[32] G. V. Zaitsev, V. A. Zinovev, and N. V. Semakov. Minimum-check-density codes for correcting bytes of errors. *Problems in Information Transmission*, 19:29–37, 1983.

A. Metric Computation Examples

In this section, we give examples of how some of the metrics are computed from information contained in the generator matrix of a code.

We do this in the context of the EvenOdd code with parameters $\langle p, n, S_A \rangle = \langle 5, 4, 64 \rangle$ (so $N = 6$ and strip size is 256KB). This is simple enough to work out by hand, but complex enough to show some of the subtleties in the computations.

We consider only example costs for given scenarios, and not the full averaging (as that would take too much space).

Here is the generator matrix G for EO$\langle 5, 4 \rangle$:

	S_0				S_1				S_2				S_3				S_4				S_5			
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	P_0	P_1	P_2	P_3	Q_0	Q_1	Q_2	Q_3
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0
2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0
3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
4	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0
6	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
7	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
8	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1
A	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	1	1	1
B	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0
C	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	1
D	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	1	1	1
E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0
F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0

There are $N = 6$ strips with $n = 4$ as data strips and $q = 2$ as parity strips. Each strip contains $r = 4$ elements. Each row of G corresponds to a user data element. The block structure indicates the partitioning into strips (with 4 elements per strip indicated by the 4 columns per block). Each column with exactly one 1 shows the placement of the corresponding user data element into its strip. Since there are exactly 16 rows, we label the rows by the hex character between $0 - \mathrm{F}$. Each column with more than one 1 maps to a parity element; such a parity element is computed by the XOR of all the data elements corresponding to rows with a 1.

The data/parity element layout is shown here:

S_0	S_1	S_2	S_3	S_4	S_5
0	4	8	C	P_0	Q_0
1	5	9	D	P_1	Q_1
2	6	A	E	P_2	Q_2
3	7	B	F	P_3	Q_3

Note that S_4 represents the P-parity strip and S_5 represents the Q-parity strip.

The generator matrix tells us, for example, that parity element Q_1 is computed as

$$Q_1 = 1 \oplus 4 \oplus 7 \oplus A \oplus D \oplus F.$$

For our examples, we assume we are in Critical mode and that S_0 and S_2 are lost.

A.1. Short Write Cases

We consider two examples of a short write, first to the first 4KB chunk in element 7 on strip S_1 and then to the first 4KB chunk of element B on strip S_2. (Note that which 4KB chunk is irrelevant, only the element affected and the size matters.)

A.1.1. Short Write to "good" element Since element 7 is "good", we first study the generator matrix to see what parity elements (on good strips) we need to update. These are P_3, Q_0, Q_1, Q_2 and Q_3 (as indicated by the 1s in row 7 of the matrix). This means that we must update four parity elements on strip S_5 and one on strip S_4. Because element 7 is good, we must also update the element on strip S_1. That is, $wr_1 = 2$ and $wr_2 = 1$. For the multi-element strips, we write (and read) the full strip of size 64 (in 4KB units).

We next examine the two algorithms (parity increment and parity compute) for updating parity elements.

For parity increment, we have $rd_1 = wr_1$ and $rd_2 = wr_2$ because the element 7 is good. Since there are $m = 5$ parity elements that need updating, the XOR overhead costs

is $XORO = 4 \cdot m = 20$ if we use equation (5.1) or $XORO = 3 \cdot m + 3 = 18$ if we use equation (5.2). We select the latter for lower costs.

In short, we have for parity increment (by equations (4.4), (4.5) and (4.6) with $er = ew = 1$, $sr = sw = 64$, and $|c| = 1$):

$$\begin{aligned} IOC &= rd_1 + rd_2 + wr_1 + wr_2 = 2 + 1 + 2 + 1 = 6 \\ IOE &= 2 \cdot IOE(1) + 1 \cdot IOE(64) + 2 \cdot IOE(1) + 1 \cdot IOE(64) = 8.64 \\ XORO &= 18 \\ MBWC &= 1 + 2 \cdot 1 + 1 \cdot 64 + 18 \cdot 1 + 2 \cdot 1 + 1 \cdot 64 = 151. \end{aligned}$$

Note that this computation is the same independent of the state of the strips S_0 and S_2 (because we do not touch them in this algorithm).

For parity compute, we require reading all the first 4KB chunks of each of the elements needed to compute all the parity (the first 4KB chunk of the parity elements). This includes for parity element P_3 the elements 3,B and F, but elements 3 and B are on lost strips, so we cannot use parity compute with these strips down. [However, we do observe that if the stripe were in Normal mode, we would need to read (for all the parity elements), multiple elements from each of strips S_0, S_2 and S_3, so $rd_2 = 3$ and this will clearly exceed the IOE costs of parity increment above. That is, even in Normal mode, this element should be updated by parity increment to minimize IOE costs.]

So, the costs in the above equations reflect the costs of a Short Write to any 4KB chunk of the "good" element 7.

A.1.2. Short Write to "lost" element Now suppose that the host issues a Short Write to the first 4KB chunk of element B. This only touches $m = 2$ parity elements P_3 and Q_0. Since this element B is lost, we have $wr_1 = 2$ and $wr_2 = 0$ (there is no write to update the lost element). For parity increment, since element B is lost, we need the reconstruction formula for this element when strips S_0 and S_2 are lost. The methodology in [5] explains how to find such a formula in this case. Find the formula for S, use it to find A, then 2,8,0, then B via the sequence

$$\begin{aligned} \underline{\mathrm{S}} &= P_0 \oplus P_1 \oplus P_2 \oplus P_3 \oplus Q_0 \oplus Q_1 \oplus Q_2 \oplus Q_3 \\ \underline{\mathrm{A}} &= 7 \oplus D \oplus \underline{\mathrm{S}} \\ \underline{2} &= 6 \oplus \underline{\mathrm{A}} \oplus E \oplus P_2 \\ \underline{8} &= \underline{2} \oplus 5 \oplus Q_2 \oplus \underline{\mathrm{S}} \\ \underline{0} &= 4 \oplus \underline{8} \oplus C \oplus P_0 \\ B &= \underline{0} \oplus E \oplus Q_0 \oplus \underline{\mathrm{S}}. \end{aligned} \tag{A.1}$$

The underlined terms are not explicitly computed; we combine these equations, collapse terms and yield the shorter formula:

$$B = 4 \oplus 5 \oplus 6 \oplus 7 \oplus C \oplus D \oplus P_1 \oplus P_3 \oplus Q_1 \oplus Q_3, \tag{A.2}$$

which has $L = 10$ input or source terms for the XOR computation. Note that we need parity elements for reconstruction that are not being updated and that Q_0 that is being updated is not used in this formula. So, we need to read every element (or part of) that is represented in this formula as well as Q_0. Hence, $rd_1 = 0$ and $rd_2 = 4$ (to read all the other elements) for the parity increment algorithm.

As above, by equations (4.4), (4.5) and (4.6) with $er = ew = 1$, $sr = sw = 64$, and $|c| = 1$, we find the costs of parity increment is:

$$\begin{aligned} IOC &= rd_1 + rd_2 + wr_1 + wr_2 = 0 + 4 + 2 + 0 = 6 \\ IOE &= 0 \cdot IOE(1) + 4 \cdot IOE(64) + 2 \cdot IOE(1) + 0 \cdot IOE(64) = 11.16 \\ XORO &= 10 + 2 + 3 \cdot 2 = 18 \\ MBWC &= 1 + 0 \cdot 1 + 4 \cdot 64 + 18 \cdot 1 + 2 \cdot 1 + 0 \cdot 64 = 277. \end{aligned}$$

As for element 7, the parity compute algorithm requires reading data from lost strips (e.g., from element 3 on strip S_0 to update parity element P_3). Consequently, we choose not to use parity increment in this case.

A.2. Short Read Cases

Suppose, as before, that strips S_0 and S_2 are lost and we get a host read for the first 4KB chunk in elements 7 and B.

A.2.1. Short Read from "good" element For the read from element 7, because it is good, we have simply to read the 4KB chunk itself; there are no XOR or write costs:

$$\begin{aligned} IOC &= rd_1 + rd_2 + wr_1 + wr_2 = 1 + 0 + 0 + 0 = 1 \\ IOE &= 1 \cdot IOE(1) + 0 \cdot IOE(64) + 0 \cdot IOE(1) + 0 \cdot IOE(64) = 1.02 \\ XORO &= 0 \\ MBWC &= 1 + 1 \cdot 1 + 0 \cdot 64 + 0 \cdot 1 + 0 \cdot 1 + 0 \cdot 64 = 2. \end{aligned}$$

A.2.2. Short Read from "lost" element When the host requests a read from element B, we need to reconstruct B using the formula in (A.2). This also implies that we need to

read from multiple elements from four strips ($rd_2 = 4$). So, we find

$$\begin{aligned} IOC &= rd_1 + rd_2 + wr_1 + wr_2 = 0 + 4 + 0 + 0 = 4 \\ IOE &= 0 \cdot IOE(1) + 4 \cdot IOE(64) + 0 \cdot IOE(1) + 0 \cdot IOE(64) = 9.12 \\ XORO &= L + 1 = 11 \\ MBWC &= 1 + 0 \cdot 1 + 4 \cdot 64 + 11 \cdot 1 + 0 \cdot 1 + 0 \cdot 64 = 268. \end{aligned}$$

A.3. Strip Write Cases

In this section, we describe the computations for two Strip Write cases, first to strip S_1 which is "good" and then to strip S_2 which (along with S_0) is assumed lost.

A.3.1. Strip Write to "good" strip We see from the generator matrix that the four elements (4–7) on strip S_1 touch all the parity elements on strips S_4 and S_5, for a total of $m = 8$ parity elements that need updating. Also, we have $wr_1 = 0$ and $wr_2 = 3$ to update the two full parity strips as well as the strip S_1. For the parity increment algorithm, we also need to read these same strips so $rd_1 = 0$ and $rd_2 = 3$. For the XOR overhead costs, we see that each of the parity elements P_i (for $i = 0, \ldots, 3$) is touched by exactly one of the updating elements, so $t_j = 1$ for each of these parity elements. For the parity element Q_0, we have $t_j = 1$ and for the other Q-parity, $t_j = 2$. Consequently, we have $T_I = 11$ for a total XOR overhead cost of

$$XORO = T_I + 2 \cdot 8 + 3 \cdot 4 = 11 + 16 + 12 = 39,$$

according to the formula in (5.4) with $r_D = 4$. The alternative formula (5.5) gives a total of

$$XORO = 2T_I + 2 \cdot 8 = 22 + 16 = 38,$$

so we select the computation algorithm from which this formula is derived for our parity increment.

We always read/write full elements so $er = ew = 64/4 = 16$ (that is, each element is 16 units of 4KB each because we have $r = 4$). We also are receiving $|c| = 64$ units from the host. In summary, we have the following costs to update strip S_1:

$$\begin{aligned} IOC &= rd_1 + rd_2 + wr_1 + wr_2 = 0 + 3 + 0 + 3 = 6 \\ IOE &= 0 \cdot IOE(16) + 3 \cdot IOE(64) + 0 \cdot IOE(16) + 3 \cdot IOE(64) = 13.68 \\ XORO &= 38 \\ MBWC &= 64 + 0 \cdot 16 + 3 \cdot 64 + 38 \cdot 16 + 0 \cdot 16 + 3 \cdot 64 = 1056. \end{aligned}$$

As for Short Write, the parity compute algorithm will require reading data from lost strips, so we skip this algorithm in this case.

A.3.2. Strip Write to "lost" strip The strip S_2 has elements that together touch all the parity elements so $m = 8$ in this case. Since this strip is lost, we have $wr_1 = 0$ and $wr_2 = 2$ (only update the two parity strips).

Next we need to find reconstruction formulas for the old data in strip S_1. Following the steps in (A.1) according to the method in [5] and adding the next two steps for reconstructing the last element 9, we have the new sequence of collapsed equations:

$$\begin{aligned} S &= P_0 \oplus P_1 \oplus P_2 \oplus P_3 \oplus Q_0 \oplus Q_1 \oplus Q_2 \oplus Q_3 \\ A &= 7 \oplus D \oplus S \\ 8 &= 6 \oplus A \oplus E \oplus P_2 \oplus 5 \oplus Q_2 \oplus S \\ B &= 4 \oplus 8 \oplus C \oplus P_0 \oplus E \oplus Q_0 \oplus S \\ 9 &= 6 \oplus 7 \oplus B \oplus C \oplus F \oplus P_3 \oplus Q_3 \oplus S. \end{aligned}$$

Note that this methodology saves an intermediate result (namely S). Summing, we find the total XOR overhead cost of using these formulas to compute the dataDeltas according to (5.6) is

$$L + 1 + 2 \cdot 4 = (8 + 3 + 7 + 7 + 8) + 1 + 8 = 42$$

(the extra +1 is for the XOR output used in saving the intermediate computation of S).

However, the algorithm in Section 5.3 rearranges the equations in a different order. In particular, it finds that element 8 can be constructed with 7 inputs, whereas each of the other elements requires 10 inputs (e.g., A can be generated by the composition of the first two formulas above). So this algorithm computes element 8 first. Continuing this approach, we generate the following sequence that also regenerates the required elements but at a slightly lower cost:

$$\begin{aligned} 8 &= 5 \oplus 6 \oplus 7 \oplus D \oplus E \oplus P_2 \oplus Q_2 \\ 9 &= 4 \oplus 6 \oplus 7 \oplus 8 \oplus E \oplus F \oplus P_0 \oplus P_3 \oplus Q_0 \oplus Q_3 \\ A &= 4 \oplus 5 \oplus 7 \oplus 9 \oplus F \oplus P_1 \oplus Q_1 \\ B &= 6 \oplus 9 \oplus A \oplus C \oplus D \oplus F \oplus P_3 \oplus Q_3. \end{aligned}$$

Here, the dataDelta overhead is

$$L + 2 \cdot 4 = (7 + 10 + 7 + 8) + 8 = 40,$$

for a total cost as in (5.7) of

$$T_I + 2 \cdot 8 + L + 2 \cdot 4 = 11 + 16 + 40 = 67.$$

(We have $T_I = 11$ again because five of the eight parity elements are touched by only one updating element and the other three are touched by only two.)

We see from the reconstruction formulas that we need to read at least two elements from all the good data strips as well as the old parity elements, so we have $rd_1 = 0$ and $rd_2 = 4$.

In summary, our costs for parity increment are:

$$\begin{aligned} IOC &= rd_1 + rd_2 + wr_1 + wr_2 = 0 + 4 + 0 + 2 = 6 \\ IOE &= 0 \cdot IOE(16) + 4 \cdot IOE(64) + 0 \cdot IOE(16) + 2 \cdot IOE(64) = 13.68 \\ XORO &= 67 \\ MBWC &= 64 + 0 \cdot 16 + 4 \cdot 64 + 0 \cdot 16 + 67 \cdot 16 + 2 \cdot 64 = 1520. \end{aligned}$$

Again, the parity compute algorithm depends on data from the lost strip S_0, so we must use the parity increment algorithm with the costs given above.

A.4. Strip Read Cases

In this section, we describe the computations for two Strip Read cases paralleling the Strip Write cases of the previous section.

A.4.1. Strip Read from "good" strip This case is analogous to the Short Read good example, with the exceptions that $|c| = 64$ (as for the Strip Write case, but now sent back to the host):

$$\begin{aligned} IOC &= rd_1 + rd_2 + wr_1 + wr_2 = 0 + 1 + 0 + 0 = 1 \\ IOE &= 0 \cdot IOE(16) + 1 \cdot IOE(64) + 0 \cdot IOE(16) + 0 \cdot IOE(64) = 2.28 \\ XORO &= 0 \\ MBWC &= 64 + 0 \cdot 16 + 1 \cdot 64 + 0 \cdot 1 + 0 \cdot 16 + 0 \cdot 64 = 128. \end{aligned}$$

A.4.2. Strip Read from "lost" strip From Section 5.4, we have for a lost strip XOR overhead computed from the recursive reconstruction formulas (as we did in the section above for lost Strip Write) but is simply $L + r_D = 32 + 4 = 36$. We only need to read data so $wr_1 = wr_2 = 0$, but we need to read the same set of data and parity elements (excluding parity elements we do not need for reconstruction). We see from the from above that $rd_1 = 0$ and $rd_2 = 4$. Thus, our lost Strip Read costs are:

$$\begin{aligned} IOC &= rd_1 + rd_2 + wr_1 + wr_2 = 0 + 4 + 0 + 0 = 4 \\ IOE &= 0 \cdot IOE(16) + 4 \cdot IOE(64) + 0 \cdot IOE(16) + 0 \cdot IOE(64) = 9.12 \\ XORO &= 32 \\ MBWC &= 64 + 0 \cdot 16 + 4 \cdot 64 + 0 \cdot 16 + 36 \cdot 16 + 0 \cdot 64 = 896. \end{aligned}$$

B. Metrics For All Usecases

In this section, in Tables B.1 to B.12, we present the raw numbers for all metrics and all codes at the two selected array sizes: $N = 8$, strip size 240KB and $N = 16$, strip size 256KB (with the appropriate caveats for the X-Code).

B.1. Strip Count $N = 8$

Table B.1: **Short Write,** $N = 8$

Stripe State	Code	*IOC*	*IOE*	*XORO*	*MBWC*	Eff. IO Cap.
Normal	IDEAL$\langle 8\rangle$	6.00	6.12	8.00	15.00	100.00
	BR$\langle 7,6\rangle$	6.00	6.45	8.56	31.94	
	EO$\langle 7,6\rangle$	6.00	6.45	10.22	33.61	
	RDP$\langle 7,6\rangle$	6.00	7.76	10.78	99.72	
	BCP$\langle 8\rangle$	6.00	6.12	8.00	15.00	
	X$\langle 7\rangle$	6.00	6.12	8.00	15.00	
	LSI$\langle 4\rangle$	5.00	5.10	6.00	12.00	
Degr.	IDEAL$\langle 8\rangle$	5.62	5.74	7.75	14.38	106.67
	BR$\langle 7,6\rangle$	5.88	6.28	8.22	29.43	102.68
	EO$\langle 7,6\rangle$	5.88	6.28	9.69	30.91	102.68
	RDP$\langle 7,6\rangle$	5.88	7.43	10.09	88.67	104.48
	BCP$\langle 8\rangle$	5.77	6.01	7.75	20.97	101.83
	X$\langle 7\rangle$	5.60	5.87	7.57	22.06	104.26
	LSI$\langle 4\rangle$	4.62	4.72	5.75	11.38	108.11
Crit.	IDEAL$\langle 8\rangle$	5.46	5.57	7.29	13.75	109.80
	BR$\langle 7,6\rangle$	5.47	7.06	10.20	90.43	91.34
	EO$\langle 7,6\rangle$	5.49	7.14	11.33	94.79	90.26
	RDP$\langle 7,6\rangle$	5.50	7.99	11.65	137.49	97.05
	BCP$\langle 8\rangle$	5.52	6.82	8.71	74.58	89.72
	X$\langle 7\rangle$	5.17	6.59	8.23	80.11	92.88
	LSI$\langle 4\rangle$	4.14	4.23	5.36	10.50	120.69

Table B.2: **Short Read,** $N = 8$

Stripe State	Code	*IOC*	*IOE*	*XORO*	*MBWC*	Eff. IO Cap.
Normal	IDEAL⟨8⟩	1.00	1.02	0.00	2.00	100.00
	BR⟨7, 6⟩	1.00	1.02	0.00	2.00	
	EO⟨7, 6⟩	1.00	1.02	0.00	2.00	
	RDP⟨7, 6⟩	1.00	1.02	0.00	2.00	
	BCP⟨8⟩	1.00	1.02	0.00	2.00	
	X⟨7⟩	1.00	1.02	0.00	2.00	
	LSI⟨4⟩	1.00	1.02	0.00	2.00	
Degr.	IDEAL⟨8⟩	1.62	1.66	0.88	3.50	61.54
	BR⟨7, 6⟩	1.62	1.66	0.88	3.50	61.54
	EO⟨7, 6⟩	1.62	1.66	0.88	3.50	61.54
	RDP⟨7, 6⟩	1.62	1.66	0.88	3.50	61.54
	BCP⟨8⟩	1.62	1.66	0.88	3.50	61.54
	X⟨7⟩	1.57	1.60	0.86	3.43	63.64
	LSI⟨4⟩	1.12	1.15	0.38	2.50	88.89
Crit.	IDEAL⟨8⟩	2.25	2.30	1.75	5.00	44.44
	BR⟨7, 6⟩	2.25	3.52	4.26	68.79	28.97
	EO⟨7, 6⟩	2.25	3.57	4.11	70.99	28.59
	RDP⟨7, 6⟩	2.25	3.53	4.08	69.08	28.89
	BCP⟨8⟩	2.25	3.38	3.17	60.85	30.14
	X⟨7⟩	2.14	3.34	3.09	64.06	30.52
	LSI⟨4⟩	1.25	1.27	0.75	3.00	80.00

Table B.3: **Strip Write,** $N = 8$

Stripe State	Code	*IOC*	*IOE*	*XORO*	*MBWC*	Eff. IO Cap.
Normal	IDEAL⟨8⟩	6.00	13.20	8.00	900.00	100.00
	BR⟨7, 6⟩	6.00	13.20	49.67	916.67	
	EO⟨7, 6⟩	6.00	13.20	56.33	983.33	
	RDP⟨7, 6⟩	6.00	13.20	56.33	983.33	
	BCP⟨8⟩	14.00	19.40	24.00	675.00	
	X⟨7⟩	14.00	20.00	40.00	750.00	
	LSI⟨4⟩	5.00	11.00	6.00	720.00	
Degr.	IDEAL⟨8⟩	5.62	12.38	7.75	862.50	106.67
	BR⟨7, 6⟩	5.62	12.38	48.38	881.25	106.67
	EO⟨7, 6⟩	5.62	12.38	55.88	956.25	106.67
	RDP⟨7, 6⟩	5.62	12.38	55.88	956.25	106.67
	BCP⟨8⟩	12.38	17.66	23.25	658.12	109.84
	X⟨7⟩	12.00	17.71	37.14	707.14	112.90
	LSI⟨4⟩	4.62	10.18	5.75	682.50	108.11
Crit.	IDEAL⟨8⟩	5.46	12.02	7.29	825.00	109.80
	BR⟨7, 6⟩	5.46	12.02	51.68	904.70	109.80
	EO⟨7, 6⟩	5.46	12.02	58.51	972.92	109.80
	RDP⟨7, 6⟩	5.46	12.02	54.43	932.14	109.80
	BCP⟨8⟩	10.71	16.18	25.39	699.11	119.91
	X⟨7⟩	10.00	15.81	40.00	740.48	126.51
	LSI⟨4⟩	4.14	9.11	5.36	630.00	120.69

Table B.4: **Strip Read,** $N = 8$

Stripe State	Code	*IOC*	*IOE*	*XORO*	*MBWC*	Eff. IO Cap.
Normal	IDEAL⟨8⟩	1.00	2.20	0.00	120.00	100.00
	BR⟨7, 6⟩	1.00	2.20	0.00	120.00	
	EO⟨7, 6⟩	1.00	2.20	0.00	120.00	
	RDP⟨7, 6⟩	1.00	2.20	0.00	120.00	
	BCP⟨8⟩	1.00	1.90	0.00	90.00	
	X⟨7⟩	1.00	2.00	0.00	100.00	
	LSI⟨4⟩	1.00	2.20	0.00	120.00	
Degr.	IDEAL⟨8⟩	1.62	3.58	0.88	210.00	61.54
	BR⟨7, 6⟩	1.62	3.57	5.25	210.00	61.54
	EO⟨7, 6⟩	1.62	3.57	5.25	210.00	61.54
	RDP⟨7, 6⟩	1.62	3.57	5.25	210.00	61.54
	BCP⟨8⟩	1.75	3.55	2.62	174.14	53.59
	X⟨7⟩	1.71	3.77	4.29	195.71	53.03
	LSI⟨4⟩	1.12	2.47	0.38	150.00	88.89
Crit.	IDEAL⟨8⟩	2.25	4.95	1.75	300.00	44.44
	BR⟨7, 6⟩	2.25	4.95	17.43	369.29	44.44
	EO⟨7, 6⟩	2.25	4.95	20.08	395.83	44.44
	RDP⟨7, 6⟩	2.25	4.95	16.01	355.06	44.44
	BCP⟨8⟩	2.25	4.73	7.93	287.68	40.21
	X⟨7⟩	2.14	4.86	12.38	309.52	41.18
	LSI⟨4⟩	1.25	2.75	0.75	180.00	80.00

Table B.5: **Full Stripe Write,** $N = 8$

Stripe State	Code	*IOC*	*IOE*	*XORO*	*MBWC*	Eff. IO Cap.
Normal	IDEAL⟨8⟩	8.00	17.60	14.00	1680.00	100.00
	BR⟨7, 6⟩	8.00	17.60	89.00	1730.00	
	EO⟨7, 6⟩	8.00	17.60	109.00	1930.00	
	RDP⟨7, 6⟩	8.00	17.60	109.00	1930.00	
	BCP⟨8⟩	8.00	17.60	56.00	1680.00	
	X⟨7⟩	7.00	16.80	84.00	1680.00	
	LSI⟨4⟩	8.00	17.60	12.00	1440.00	
Degr.	IDEAL⟨8⟩	7.00	15.40	12.25	1515.00	114.29
	BR⟨7, 6⟩	7.00	15.40	77.88	1558.75	114.29
	EO⟨7, 6⟩	7.00	15.40	95.38	1733.75	114.29
	RDP⟨7, 6⟩	7.00	15.40	95.38	1733.75	114.29
	BCP⟨8⟩	7.00	15.40	49.00	1515.00	114.29
	X⟨7⟩	6.00	14.40	72.00	1490.00	116.67
	LSI⟨4⟩	7.00	15.40	10.50	1290.00	114.29
Crit.	IDEAL⟨8⟩	6.00	13.20	10.50	1350.00	133.33
	BR⟨7, 6⟩	6.00	13.20	66.75	1387.50	133.33
	EO⟨7, 6⟩	6.00	13.20	81.75	1537.50	133.33
	RDP⟨7, 6⟩	6.00	13.20	81.75	1537.50	133.33
	BCP⟨8⟩	6.00	13.20	42.00	1350.00	133.33
	X⟨7⟩	5.00	12.00	60.00	1300.00	140.00
	LSI⟨4⟩	6.00	13.20	9.00	1140.00	133.33

Table B.6: **Rebuild,** $N = 8$

Stripe State	Code	*IOC*	*IOE*	*XORO*	*MBWC*
Degr.	IDEAL⟨8⟩	7.00	15.40	7.00	840.00
	BR⟨7, 6⟩	7.00	15.40	42.62	846.25
	EO⟨7, 6⟩	7.00	15.40	45.12	871.25
	RDP⟨7, 6⟩	7.00	15.40	45.12	871.25
	BCP⟨8⟩	8.00	17.60	28.00	900.00
	X⟨7⟩	7.00	16.80	42.00	910.00
	LSI⟨4⟩	3.00	6.60	3.00	360.00
Crit.	IDEAL⟨8⟩	8.00	17.60	14.00	1320.00
	BR⟨7, 6⟩	8.00	17.60	93.75	1417.50
	EO⟨7, 6⟩	8.00	17.60	114.89	1628.93
	RDP⟨7, 6⟩	8.00	17.60	95.61	1436.07
	BCP⟨8⟩	8.00	17.60	56.00	1320.00
	X⟨7⟩	7.00	16.80	84.00	1330.00
	LSI⟨4⟩	5.29	11.63	6.00	677.14

B.2. Strip Count $N = 16$

Table B.7: **Short Write,** $N = 16$

Stripe State	Code	*IOC*	*IOE*	*XORO*	*MBWC*	Eff. IO Cap.
Normal	IDEAL⟨16⟩	6.00	6.12	8.00	15.00	100.00
	BR⟨17, 14⟩	6.00	6.27	8.23	22.54	
	EO⟨17, 14⟩	6.00	6.27	10.67	24.98	
	RDP⟨17, 14⟩	6.00	8.34	11.52	129.33	
	BCP⟨16⟩	6.00	6.12	8.00	15.00	
	X⟨17⟩	6.00	6.12	8.00	15.00	
	LSI⟨8⟩	5.00	5.10	6.00	12.00	
Degr.	IDEAL⟨16⟩	6.31	6.44	8.88	16.19	95.05
	BR⟨17, 14⟩	6.44	6.70	8.59	22.88	93.48
	EO⟨17, 14⟩	6.44	6.70	10.88	25.17	93.48
	RDP⟨17, 14⟩	6.44	8.64	11.62	122.94	96.44
	BCP⟨16⟩	6.38	6.58	8.38	19.20	93.02
	X⟨17⟩	6.42	6.62	8.41	19.51	92.48
	LSI⟨8⟩	4.81	4.91	5.88	11.69	103.90
Crit.	IDEAL⟨16⟩	6.74	6.88	8.70	16.44	89.00
	BR⟨17, 14⟩	6.74	8.91	18.90	128.17	70.34
	EO⟨17, 14⟩	6.75	8.95	19.49	130.50	70.02
	RDP⟨17, 14⟩	6.75	10.64	20.13	215.70	78.34
	BCP⟨16⟩	6.76	8.71	12.76	111.49	70.26
	X⟨17⟩	6.83	8.99	14.93	123.92	68.07
	LSI⟨8⟩	4.60	4.69	5.72	11.32	108.69

Table B.8: **Short Read,** $N = 16$

Stripe State	Code	*IOC*	*IOE*	*XORO*	*MBWC*	Eff. IO Cap.
Normal	IDEAL⟨16⟩	1.00	1.02	0.00	2.00	100.00
	BR⟨17, 14⟩	1.00	1.02	0.00	2.00	
	EO⟨17, 14⟩	1.00	1.02	0.00	2.00	
	RDP⟨17, 14⟩	1.00	1.02	0.00	2.00	
	BCP⟨16⟩	1.00	1.02	0.00	2.00	
	X⟨17⟩	1.00	1.02	0.00	2.00	
	LSI⟨8⟩	1.00	1.02	0.00	2.00	
Degr.	IDEAL⟨16⟩	1.81	1.85	0.94	3.75	55.17
	BR⟨17, 14⟩	1.81	1.85	0.94	3.75	55.17
	EO⟨17, 14⟩	1.81	1.85	0.94	3.75	55.17
	RDP⟨17, 14⟩	1.81	1.85	0.94	3.75	55.17
	BCP⟨16⟩	1.81	1.85	0.94	3.75	55.17
	X⟨17⟩	1.82	1.86	0.94	3.76	54.84
	LSI⟨8⟩	1.06	1.08	0.19	2.25	94.12
Crit.	IDEAL⟨16⟩	2.62	2.68	1.88	5.50	38.10
	BR⟨17, 14⟩	2.62	4.59	11.88	110.84	22.24
	EO⟨17, 14⟩	2.62	4.61	10.33	110.70	22.11
	RDP⟨17, 14⟩	2.62	4.60	10.30	110.00	22.17
	BCP⟨16⟩	2.62	4.46	5.93	98.73	22.87
	X⟨17⟩	2.65	4.70	8.03	111.46	21.72
	LSI⟨8⟩	1.12	1.15	0.38	2.50	88.89

Table B.9: **Strip Write,** $N = 16$

Stripe State	Code	*IOC*	*IOE*	*XORO*	*MBWC*	Eff. IO Cap.
Normal	IDEAL⟨16⟩	6.00	13.68	8.00	960.00	100.00
	BR⟨17, 14⟩	6.00	13.68	129.86	967.43	
	EO⟨17, 14⟩	6.00	13.68	155.86	1071.43	
	RDP⟨17, 14⟩	6.00	13.68	156.14	1072.57	
	BCP⟨16⟩	30.00	36.72	56.00	840.00	
	X⟨17⟩	34.00	41.20	120.00	900.00	
	LSI⟨8⟩	5.00	11.40	6.00	768.00	
Degr.	IDEAL⟨16⟩	6.31	14.39	8.88	1036.00	95.05
	BR⟨17, 14⟩	6.31	14.39	144.44	1045.75	95.05
	EO⟨17, 14⟩	6.31	14.39	178.56	1182.25	95.05
	RDP⟨17, 14⟩	6.31	14.39	178.94	1183.75	95.05
	BCP⟨16⟩	28.19	35.40	62.12	913.50	103.74
	X⟨17⟩	32.00	39.76	134.12	984.71	103.61
	LSI⟨8⟩	4.81	10.97	5.88	748.00	103.90
Crit.	IDEAL⟨16⟩	6.74	15.37	8.70	1052.27	89.00
	BR⟨17, 14⟩	6.74	15.37	169.81	1174.72	89.00
	EO⟨17, 14⟩	6.74	15.37	206.13	1320.00	89.00
	RDP⟨17, 14⟩	6.74	15.37	185.15	1236.07	89.00
	BCP⟨16⟩	26.37	34.22	71.22	1018.73	107.29
	X⟨17⟩	30.00	38.49	154.19	1101.18	107.05
	LSI⟨8⟩	4.60	10.49	5.72	724.27	108.70

Table B.10: **Strip Read,** $N = 16$

Stripe State	Code	*IOC*	*IOE*	*XORO*	*MBWC*	Eff. IO Cap.
Normal	IDEAL⟨16⟩	1.00	2.28	0.00	128.00	100.00
	BR⟨17, 14⟩	1.00	2.28	0.00	128.00	
	EO⟨17, 14⟩	1.00	2.28	0.00	128.00	
	RDP⟨17, 14⟩	1.00	2.28	0.00	128.00	
	BCP⟨16⟩	1.00	2.12	0.00	112.00	
	X⟨17⟩	1.00	2.20	0.00	120.00	
	LSI⟨8⟩	1.00	2.28	0.00	128.00	
Degr.	IDEAL⟨16⟩	1.81	4.13	0.94	240.00	55.17
	BR⟨17, 14⟩	1.81	4.13	15.00	240.00	55.17
	EO⟨17, 14⟩	1.81	4.13	15.00	240.00	55.17
	RDP⟨17, 14⟩	1.81	4.13	15.00	240.00	55.17
	BCP⟨16⟩	1.88	4.12	6.56	221.00	51.39
	X⟨17⟩	1.88	4.29	14.12	236.94	51.26
	LSI⟨8⟩	1.06	2.42	0.19	144.00	94.12
Crit.	IDEAL⟨16⟩	2.62	5.99	1.88	352.00	38.10
	BR⟨17, 14⟩	2.62	5.98	59.10	468.42	38.10
	EO⟨17, 14⟩	2.62	5.98	74.30	529.20	38.10
	RDP⟨17, 14⟩	2.62	5.98	53.08	444.34	38.10
	BCP⟨16⟩	2.62	5.85	22.98	400.87	36.27
	X⟨17⟩	2.65	6.11	49.85	432.35	36.03
	LSI⟨8⟩	1.12	2.57	0.38	160.00	88.89

Table B.11: **Full Stripe Write,** $N = 16$

Stripe State	Code	*IOC*	*IOE*	*XORO*	*MBWC*	Eff. IO Cap.
Normal	IDEAL⟨16⟩	16.00	36.48	30.00	3840.00	100.00
	BR⟨17, 14⟩	16.00	36.48	493.00	3892.00	
	EO⟨17, 14⟩	16.00	36.48	675.00	4620.00	
	RDP⟨17, 14⟩	16.00	36.48	677.00	4628.00	
	BCP⟨16⟩	16.00	36.48	240.00	3840.00	
	X⟨17⟩	17.00	40.12	544.00	4352.00	
	LSI⟨8⟩	16.00	36.48	24.00	3072.00	
Degr.	IDEAL⟨16⟩	15.00	34.20	28.12	3656.00	106.67
	BR⟨17, 14⟩	15.00	34.20	462.19	3704.75	106.67
	EO⟨17, 14⟩	15.00	34.20	632.81	4387.25	106.67
	RDP⟨17, 14⟩	15.00	34.20	634.69	4394.75	106.67
	BCP⟨16⟩	15.00	34.20	225.00	3656.00	106.67
	X⟨17⟩	16.00	37.76	512.00	4156.00	106.25
	LSI⟨8⟩	15.00	34.20	22.50	2912.00	106.67
Crit.	IDEAL⟨16⟩	14.00	31.92	26.25	3472.00	114.29
	BR⟨17, 14⟩	14.00	31.92	431.38	3517.50	114.28
	EO⟨17, 14⟩	14.00	31.92	590.62	4154.50	114.28
	RDP⟨17, 14⟩	14.00	31.92	592.38	4161.50	114.28
	BCP⟨16⟩	14.00	31.92	210.00	3472.00	114.28
	X⟨17⟩	15.00	35.40	480.00	3960.00	113.33
	LSI⟨8⟩	14.00	31.92	21.00	2752.00	114.28

Table B.12: **Rebuild,** $N = 16$

Stripe State	Code	*IOC*	*IOE*	*XORO*	*MBWC*
Degr.	IDEAL$\langle 16 \rangle$	15.00	34.20	15.00	1920.00
	BR$\langle 17, 14 \rangle$	15.00	34.20	240.81	1923.25
	EO$\langle 17, 14 \rangle$	15.00	34.20	252.19	1968.75
	RDP$\langle 17, 14 \rangle$	15.00	34.20	252.31	1969.25
	BCP$\langle 16 \rangle$	16.00	36.48	120.00	1984.00
	X$\langle 17 \rangle$	17.00	40.12	272.00	2244.00
	LSI$\langle 8 \rangle$	3.00	6.84	3.00	384.00
Crit.	IDEAL$\langle 16 \rangle$	16.00	36.48	30.00	2944.00
	BR$\langle 17, 14 \rangle$	16.00	36.48	542.96	3195.83
	EO$\langle 17, 14 \rangle$	16.00	36.48	747.09	4012.37
	RDP$\langle 17, 14 \rangle$	16.00	36.48	529.12	3140.50
	BCP$\langle 16 \rangle$	16.00	36.48	240.00	2944.00
	X$\langle 17 \rangle$	17.00	40.12	544.00	3332.00
	LSI$\langle 8 \rangle$	5.67	12.92	6.00	746.67

C. Average Numbers of Elements of Each Type

In the tables of Appendix B, we provided the average costs over all random operations to a stripe. As noted in Section 4.4, more detailed distributional views of the metrics can also be studied. In this section, we give the distribution of good and lost elements for each of the codes in each of the failure modes. The lost elements are divided into subclasses, labeled lost$[rd_2]$, depending on the number rd_2 of multi-element strip reads required for simple reconstruction (as if for a Short Read use case, see Section 5.2). This sort of distribution provides a closer look at where the costs, in particular *IOE* costs, are derived and can be used for other analysis such as worst-case costs, etc..

Table C.13 has the average number of elements of each type for strip counts $N = 8$ (with the exception that for the X-Code, we use only a $N = 7$). Table C.14 gives the same values for $N = 16$ (and $N = 17$ for the X-Code). The averages are computed over the failure cases within each failure mode.

Table C.13: **Element Type Counts (avg),** $N = 8$

Stripe State	Element Type	IDEAL[8]	BR[7,6]	EO[7,6]	RDP[7,6]	BCP[8]	X[7]	LSI[4]
Normal	good	6	36	36	36	24	35	4
Degr.	good	5.25	31.5	31.5	31.5	21	30	3.5
	lost[0]	0.75	4.5	4.5	4.5	3	5	0.5
Crit.	good	4.50	27	27	27	18	25	3
	lost[0]	1.50	1.50	1.29	2.39	1.71	3.33	1
	lost[1]	0	0.89	0	0.18	0	0	0
	lost[2]	0	0.18	1.07	0	0	0	0
	lost[3]	0	0	0	0	0.11	0	0
	lost[4]	0	0.71	0.68	0	0	4.00	0
	lost[5]	0	1.00	1.82	1.07	3.25	2.67	0
	lost[6]	0	4.71	4.14	5.36	0.93	0	0

Table C.14: **Element Type Counts (avg),** $N = 16$

Stripe State	Element Type	IDEAL[16]	BR[17,14]	EO[17,14]	RDP[17,14]	BCP[16]	X[17]	LSI[8]
Normal	good	14	224	224	224	112	255	8
Degr.	good	13.12	210	210	210	105	240	7.5
	lost[0]	0.88	14	14	14	7	15	0.5
Crit.	good	12.25	196	196	196	98	225	7
	lost[0]	1.75	2.22	1.87	3.39	1.87	3.75	1
	lost[1]	0	1.41	0	0.11	0	0	0
	lost[2]	0	0.11	1.52	0	0	0	0
	lost[3-11]	0	0	0	0	0	0	0
	lost[12]	0	0.48	0.42	0	0	0	0
	lost[13]	0	1.42	2.84	1.52	11.33	0	0
	lost[14]	0	22.37	21.36	22.98	0.80	14.00	0
	lost[15]	0	0	0	0	0	12.25	0