Efficient Data Mappings for Parity-Declustered Data Layouts

Eric J. Schwabe [*]
School of CTI
DePaul University
243 S. Wabash Ave
Chicago, IL 60604

Ian M. Sutherland [†]
Oracle Corporation
233 S. Wacker Drive
Chicago, IL 60606

July 17, 2002

Abstract

The joint demands of high performance and fault tolerance in a large array of disks can be satisfied by a parity-declustered data layout – an arrangement of data and redundant information that allows the rapid reconstruction of lost data while the array continues to operate. A data layout is typically generated by partitioning the data units on the disks into stripes and choosing one or more units per stripe to hold redundant information. Such a data layout can be represented as a table of stripes. The data mapping problem is the problem of translating a data address in a linear address space (the file system's view) into a disk identifier and an offset on the disk where the data is stored. Typically, the disk and offset are obtained from the data layout using table lookups, but recent work has yielded mappings that compute (disk, offset) pairs directly from data addresses without the need to store tables. In this paper, we show that parity-declustered data layouts based on commutative rings yield mappings with improved computational efficiency. These layouts also apply to a wider range of array configurations than other known layouts that do not use table lookup.

1 Introduction

1.1 Data Layouts for Disk Arrays

Disk arrays provide increased I/O throughput for large data sets by distributing the data over a collection of smaller disks (instead of a single larger disk) and allowing parallel access [8]. Since each disk in the array may fail independently with some probability per unit time, the probability that some disk in a large array will fail in unit time is greatly increased. Thus, the ability to reconstruct the contents of a failed disk is important to the feasibility of large disk arrays.

One technique to achieve fault tolerance in an array of v disks is called RAID5 (thus named by Patterson, Gibson, and Katz [8]). This technique is illustrated in Figure 1. Each disk is divided into *units*, and in each

[*] `eschwabe@cs.depaul.edu`. Supported in part by NSF Grant CCR–9996375.
[†] `ian.sutherland@oracle.com`.

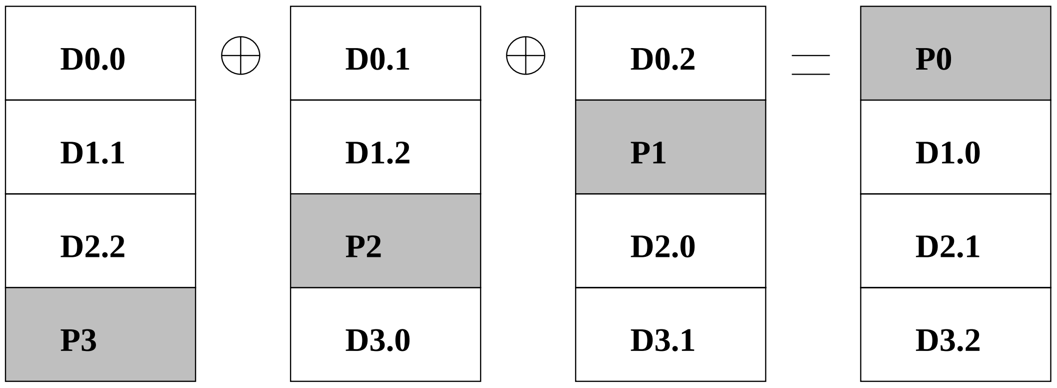

Figure 1: RAID5 on a four-disk array. In each row **i**, the parity unit **Pi** is the bitwise exclusive "or" of the three data units **Di.0**, **Di.1**, and **Di.2**.

row, one of the units holds the bitwise exclusive "or" (i.e., parity) of the remaining $v-1$ units. This allows the disk array to recover from a single disk failure, as the contents of each unit on the failed disk can be reconstructed by taking the bitwise exclusive "or" of the $v-1$ surviving units from that row. Thus, by dedicating $1/v$ of the total space in the array to redundant information, the array can recover from any single disk failure by reading the entire contents of each of the surviving disks.

In general, we can achieve fault tolerance by constructing a *data layout* — an arrangement of data and redundant information that allows the array to reconstruct the contents of one or more failed disks. A data layout is created by partitioning the units in the array into a collection of non-overlapping *stripes*. (In the RAID5 example, the stripes used are precisely the rows.) The number of units in each stripe is called the *stripe size*. Some subset of the units in each stripe will hold users' data; however, one or more units per stripe will instead hold redundant information computed from the data stored in the other units of the stripe. (In the RAID5 example, the stripe size is v, and one unit per stripe stores the parity of the remaining units.) This redundant information stored for each stripe enables the array to recover from disk failures.

Clients using the disk array to store data need not be concerned with the details of the data layout. In particular, they need not know which units contain data or redundant information, or even on which disks and at which offsets their data are stored. To such a client, all of the data will appear to reside on a single logical disk, consisting entirely of data units and organized into a linear address space.

If an array must remain available during the reconstruction of lost data, or must be taken off-line for as little time as possible for failure recovery, we may wish to reduce the time spent on failure recovery at the cost of dedicating more space to redundant information. This tradeoff of additional redundant space for reduced recovery time can be achieved using a technique called *parity declustering*, in which the stripe size k is smaller than the array size v. Parity-declustered data layouts have been considered by, among others, Holland and Gibson [4], Muntz and Lui [7], Schwabe and Sutherland [9], Stockmeyer [10], and Alvarez, Burkhard, and Cristian [1]. Holland and Gibson [4] described the following conditions that data layouts should satisfy:

1. Fault tolerance: When one disk fails (or more, if we wish to consider multiple disk failures), the data that resided on that disk must be computable from the data residing on the surviving disks.

2. Even distribution of redundant information: The redundant information that is stored in the array to effect the desired level of fault tolerance must be evenly distributed over the disks of the array.

3. Even distribution of reconstruction workload: When disks fail, the additional workload on the array generated by the reconstruction of their lost data must be evenly distributed over the surviving disks.

$\{0,1,2\}$	$\{0,3,6\}$	$\{1,4,6\}$
$\{0,1,4\}$	$\{0,5,6\}$	$\{2,3,4\}$
$\{0,1,6\}$	$\{1,2,3\}$	$\{2,3,6\}$
$\{0,2,4\}$	$\{1,2,5\}$	$\{2,4,6\}$
$\{0,2,5\}$	$\{1,3,5\}$	$\{2,5,6\}$
$\{0,3,4\}$	$\{1,3,6\}$	$\{3,4,5\}$
$\{0,3,5\}$	$\{1,4,5\}$	$\{4,5,6\}$

Figure 2: A BIBD for $v = 7$, $k = 3$.

4. Large write optimization: The addresses corresponding to the data units of a single stripe must be contiguous in the linear address space. (If this holds, then when large amounts of data are written, many stripes will have all of their data units written, so the redundant information for those stripes can be updated without reading the existing values of any of the data units.)

5. Maximal parallelism: If v contiguous units of data in the linear address space are read, the resulting disk accesses must be evenly distributed over the v disks in the array.

6. Efficient data mapping: The mapping of addresses in the linear address space to the corresponding physical disks and offsets must be efficiently computable.

All of the data layouts discussed in this paper satisfy Conditions 1, 2, and 3. Conditions 4, 5, and 6 are not exclusively properties of a data layout, but rather of a data layout together with a mapping of the linear address space onto its data units. In fact, for all of the data layouts considered in this paper, we will demonstrate data mappings that satisfy Condition 4. We will not address Condition 5, as Alvarez, Burkhard, Stockmeyer, and Cristian [2] showed that Conditions 4 and 5 can only be realized simultaneously when either k is in the set $\{v, v-1, 2\}$ or $(v, k) = (3,5), (3,7), \text{or } (4,7)$. Our focus will be on efficiently mapping linear address spaces to disk identifiers and offsets, to satisfy Condition 6.

Many constructions of parity-declustered layouts use *balanced incomplete block designs* (BIBDs). A BIBD is a collection of b subsets (called *tuples*) of k elements, each drawn from a set of v elements, that satisfies the following two properties (see, e.g., Hanani [3]):

- Each element appears the same number of times (called r) among the b tuples;
- Each pair of elements appears the same number of times (called λ) among the b tuples.

In fact, as long as $k \geq 2$, the second property implies the first, since $r = \lambda \cdot \frac{v-1}{k-1}$.

For example, consider the set $\{0,1,2,3,4,5,6\}$. For $v = 7$ and $k = 3$, the collection of tuples in Figure 2 forms a BIBD with $b = 21$, $r = 9$, and $\lambda = 3$. That is, every number appears in exactly nine tuples and every pair of numbers appears in exactly three tuples.

In order to construct a data layout from a BIBD, we consider the v elements to be the disks in the array. Each tuple in the BIBD corresponds to a stripe containing one unit from each of the disks that appear in that tuple. Therefore each stripe will contain units from exactly k disks, and each disk will contain exactly r units. (We call r the *size* of the layout.) For each pair of disks, there are exactly λ stripes that contain a unit from both disks. If each stripe contains $k-1$ units of data and one of redundant information, then when one disk fails, exactly λ units from each of the remaining disks (i.e., a $\frac{k-1}{v-1}$ fraction of their contents) will have to be read to reconstruct the lost data.

In order to achieve f-fault tolerance, it is sufficient to choose f units from each stripe to hold redundant information. Schwabe and Sutherland [9] gave a general method for choosing the f units from each stripe so that they will be as evenly distributed as possible among the v disks. Alvarez, Burkhard, and Cristian [1] demonstrated a method for computing the values to be stored in each of the f chosen redundancy units from

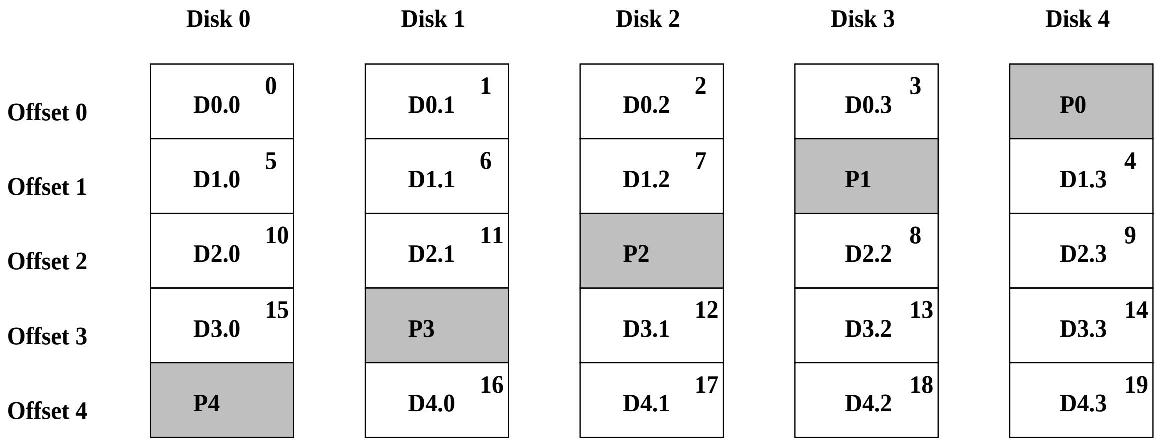

Figure 3: A mapping of addresses to data units in a RAID5 data layout with $v = 5$.

the contents of the $k - f$ data units so as to achieve f-fault tolerance within each stripe (and therefore over the entire array). Since these two techniques can be applied to any set of stripes, from this point forward we will only be concerned with choosing an appropriate division of the units into stripes.

1.2 The Data Mapping Problem

Recall that a disk array appears to its clients as a single logical disk with a linear address space. Data addresses in this space are mapped to disks and offsets on those disks.

For the RAID5 data layout illustrated in Figure 3, with $v = 5$, there are a total of 25 units, where 20 are data units and five contain redundant information. Assigning the addresses 0 through 19 to the data units as illustrated, it is clear that the disk and offset for a particular address X can easily be computed. The offset of address X is $\lfloor X/4 \rfloor$. The disk containing address X is $((X \bmod 4) - \lfloor X/4 \rfloor) \bmod 5$.

For parity-declustered data layouts, the problem of computing disks and offsets is more complicated. One way to do this is to use a table derived from a BIBD. The tuples of the BIBD make up the rows of the table, and each entry in a row is an element of that tuple. In this table, each row will represent one stripe in the layout, and each entry in a row will represent a disk from which that stripe contains a unit. Addresses from the linear address space can be assigned in row-major order to the entries of the table, ignoring the last f entries in each row, which correspond to redundancy units. (Thus $k - f$ data units are assigned to each row in the table.) This associates a disk identifier with each address. The offset for an address is the number of times the corresponding disk identifier appears in rows above the row where that address appears. This mapping is due to Holland and Gibson [4], and is illustrated in Figure 4 for the complete block design with $v = 5$ and $k = 4$. To compute a disk and offset from a given address, we first map it to a row and column in the table, setting $\mathit{row} = \mathit{address}\ /\ (k - f)$ and $\mathit{column} = \mathit{address} \bmod k - f$. Next, we use the contents of the table to determine the disk and offset where that address is located.

The disk number can be obtained with a single lookup in the table of stripes, since it is the value stored in the computed row and column. The offset is a bit more difficult to compute, as it depends on the number of occurrences of the discovered disk number that occur in rows above the computed row. While it may take many table lookups to compute this offset directly, it is a relatively simple matter to precompute all of the offsets while the table is being constructed. This will take additional time that is only linear in the number of entries in the table, and will increase the space required to store the table by only a factor of two (each location that previously stored a disk number now stores both a disk number and an offset). If this precomputation is done, then the offset can also be determined with a single table lookup.

A:	**1** (0) $^{(0)}$	**2** (0) $^{(1)}$	**3** (0) $^{(2)}$	**0** (0)
B:	**0** (1) $^{(3)}$	**2** (1) $^{(4)}$	**4** (0) $^{(5)}$	**1** (1)
C:	**0** (2) $^{(6)}$	**1** (2) $^{(7)}$	**3** (1) $^{(8)}$	**4** (1)
D:	**0** (3) $^{(9)}$	**3** (2) $^{(10)}$	**4** (2) $^{(11)}$	**2** (2)
E:	**1** (3) $^{(12)}$	**2** (3) $^{(13)}$	**4** (3) $^{(14)}$	**3** (3)

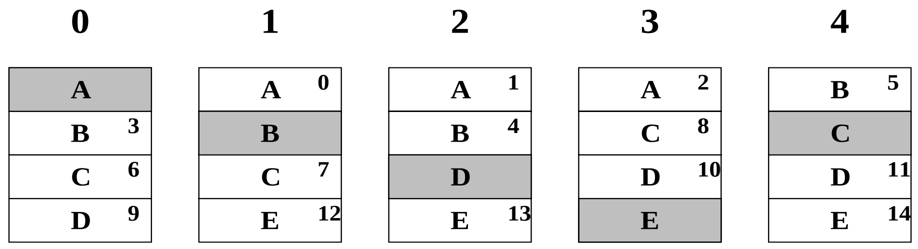

Figure 4: A table of strips derived from the complete block design with $v = 5$, $k = 4$, and the single-fault-tolerant parity-declustered data layout derived from it. Each table entry shows "**disk number** (offset) $^{\text{(address)}}$" for one unit, though only the disk number need be stored. The rightmost element in each row is the parity unit for that stripe, so it has no data address assigned to it.

However, the resulting table could be quite large. For instance, in the case of a data layout derived from a *complete* block design, which consists of all subsets of size k of the set of v disks, the table will have $\binom{v}{k}$ rows and k columns.

The remainder of this paper considers ways to reduce this space requirement by using data layouts that do not require the explicit storage of tables of stripes. In Section 2, we will give a brief description of the DATUM layouts of Alvarez, Burkhard, and Cristian [1], which are based on complete block designs, and analyze the computational complexity of their data mappings. In Sections 3 and 4 we will present an alternative: data layouts based on a class of BIBDs called ring-based block designs. Both DATUM and ring-based layouts take advantage of the mathematical structure of the layouts to compute disks and offsets directly rather than looking them up in tables. Ring-based layouts are applicable to a wider range of array configurations than DATUM layouts, and as we will discuss in Section 5, also have lesser computational requirements for their data mappings than DATUM layouts.

2 DATUM Layouts

Alvarez, Burkhard, and Cristian [1] developed the first parity-declustered data layouts, called DATUM layouts, for which mappings of data addresses to disks and offsets are not computed using table lookup. Instead, disks and offsets are computed directly from addresses, implicitly referring to the table of stripes without storing it in memory.

In the following, we describe their construction and analyze the efficiency of their data mappings.

2.1 Layout Construction

DATUM layouts are based on complete block designs, with a particular ordering of their tuples. The set of tuples in the complete block design is the set of all subsets of k of the v disks (for convenience, we assume that the v disks are labeled $\{0, 1, \ldots, v-1\}$). Within each tuple, disks appear in increasing order. The ordering of the tuples is as follows: $(X_1, \ldots, X_k)$ precedes $(Y_1, \ldots, Y_k)$ if and only if for some $j \leq k$, $X_j < Y_j$ and for all i satisfying $j < i \leq k$, $X_i = Y_i$. The number of tuples that precede a given tuple in this ordering is called the *rank* of that tuple.

Given this order, Alvarez et al. defined two functions: `loc`$(X_1, \ldots, X_k)$, which computes the rank of an input tuple, and its inverse, `invloc`$(rank)$, which generates the k elements of the tuple with a particular rank. The function `invloc` can be computed using the following algorithm:

```
invloc($rank$)
    for (int $i$ = $k$; $i$ >= 1; $i$--)
        $l$ = $i$
        while ( $\binom{l}{i}$ <= $rank$ )
            $l$ = $l$ + 1
        $X_i$ = $l$ - 1
        $rank$ = $rank - \binom{l-1}{i}$
    return $(X_1, X_2, \ldots, X_k)$
```

If we are given a row *rank* and column *col* in the table, we can compute the disk number stored at that location by taking X_{col}, the col^{th} element in the tuple returned by `invloc`$(rank)$. Once the tuple $(X_1, X_2, \ldots, X_k)$ in row *rank* has been found, the offset of the unit from that stripe on disk X_{col} can be computed using the formula

$$\#X_{col} = \sum_{i=1}^{col-1} \binom{X_i}{i} + \sum_{i=col+1}^{k} \binom{X_i - 1}{i - 1}.$$

Alvarez et al. established the correctness of the `invloc` function and the formula for $\#X_{col}$, but did not analyze their computation complexity.

2.2 Computational Complexity of Data Mappings

To determine the worst-case running time of `invloc`, we observe that the outer "for" loop has k iterations, and the inner "while" loop could have $v - k$ iterations in the worst case. A binomial coefficient is computed at the end of each iteration of the outer "for" loop, as well as in each iteration of the inner "while" loop. This yields a worst-case running time of $\Theta(vkC)$, where C is the time required to compute a binomial coefficient. The time required to compute $\#X_{col}$ is dominated by the time to compute k binomial coefficients, yielding a worst-case running time of $\Theta(kC)$.

The straightforward method to compute a binomial coefficient takes $C = \Theta(v)$ steps, yielding a total of $\Theta(kv^2)$ steps to compute the disk and offset. However, a closer inspection of the function `invloc` reveals that we do not have to compute an entirely new binomial coefficient in each iteration of the inner loop, but rather we start by computing $\binom{i}{i} = 1$, and in each iteration can go from $\binom{l}{i}$ to $\binom{l+1}{i}$ using the fact that $\binom{l+1}{i} = \frac{l+1}{l-i+1} \cdot \binom{l}{i}$. This incremental computation of binomial coefficients takes only $\Theta(1)$ steps for each iteration of the inner loop after the first (which still takes $\Theta(v)$ steps). This reduces the total time requirements for `invloc`, and thus for the entire process of computing the disk and offset from $\Theta(kv^2)$ to $\Theta(kv)$.

2.3 Usability of DATUM Layouts for Large Disk Arrays

DATUM layouts eliminate the need to store a table of size that is polynomial in v in exchange for enough space to store the tuple $(X_1, X_2, \ldots, X_k)$ and $\Theta(kv)$ time to compute disks and offsets. These layouts can be constructed for all possible values of v and k, but since DATUM layouts will always contain $\binom{v}{k}$ stripes, they may be too large to use. In general, if a layout contains b stripes, then its size is bk/v, so each disk in the array must contain at least bk/v units in order for the layout to be used on that array. For DATUM layouts,

v	k	% usable
8	all	100
16	all	100
32	1 ... 9, 24 ... 32	56.25
64	1 ... 6, 59 ... 64	18.75
128	1 ... 4, 125 ... 128	6.25
256	1 ... 4, 253 ... 256	3.125

Figure 5: DATUM layouts with size at most 10 million.

v	k	% usable:
8	all	100
16	all	100
32	all	100
64	1 ... 9, 56 ... 64	28.125
128	1 ... 7, 122 ... 128	10.9375
256	1 ... 6, 251 ... 256	4.6875

Figure 6: DATUM layouts with size at most 10 billion.

$bk/v = \binom{v-1}{k-1}$. In fact, in most cases it is even larger due to replication of the layout for parity balancing, but the size is always at least this amount.

To illustrate this drawback, consider a disk array consisting of 10GB disks that are divided into units of 4KB each. For this array, the number of units on each disk is 2,621,440 (disk size divided by unit size), so any layout with size greater than this amount cannot be used. Since disk and unit sizes can vary and we are only looking for a rough guideline for usability, we will consider any layout with size at most 10 million to be usable.

In Figure 5, we give values of k for which the DATUM layout is usable, for various array sizes v. We observe that even for moderately sized arrays that are already commercially available (e.g., 64 disks), layouts based on complete block designs are too large to be usable for more than 80% of the possible values of k. For arrays of 128 and 256 disks, the percentage of k values that are ruled out rises to more than 93% and 96% respectively; there, only the few smallest and few largest values of k yield usable DATUM layouts.

This is admittedly a very rough guideline for usability, but the same pattern of rapidly decreasing usability applies even for much more generous definitions. For 10GB disks with units consisting of a single byte, which would yield a layout size bound of 10,737,418,240 ($10 \cdot 2^{30}$ — roughly 10 billion), the results would be as indicated in Figure 6. Increasing the bound on the number of units per disk by a factor of 1000 only makes a few more values of k feasible once the array size reaches 64. (We would obtain the same table if we left the unit size as 4KB but increased the disk size to 10,000GB – well beyond the sizes currently in use.)

Furthermore, even if the disks in an array are sufficiently large to use a particular layout, using a smaller layout may still improve performance, by leading to better local load balancing across the array. Also, if the disk size is greater than the layout size but is not an integral multiple of the layout size, then some amount of space on each disk will be unusable when we cover the array with as many copies of the layout as fit. This amount of wasted space could be as much as $r - 1$ units per disk. Thus, using a smaller layout would reduce the potential amount of wasted space.

3 Ring-Based Layouts

In this section, we extend the ring-based data layouts of Schwabe and Sutherland [9] to eliminate the need to store tables to describe their stripes. We use the algebraic structure of a ring-based block design to develop functions to map data addresses to the corresponding disks and offsets.

These ring-based data layouts have two advantages over DATUM layouts:

1. They are smaller, and therefore applicable to a wider range of arrays – they contain only $v(v-1)$ stripes rather than $\binom{v}{k}$;
2. The functions to compute disks and offsets from data addresses are more efficiently computable – they have worst-case running time $O(k \log v \log \log v)$ rather than $\Theta(kv)$.

In the following, we review the ring-based data layout construction of Schwabe and Sutherland [9], and present algorithms to compute disks and offsets without explicitly storing tables of stripes.

3.1 Layout Construction

Ring-based data layouts are derived from a class of block designs called *ring-based block designs*. The elements of a ring-based block design are taken from a *commutative ring with a unit* (hereafter referred to as simply a "ring"). A ring is an algebraic object consisting of a set of elements, an addition operation (which is associative, commutative, and has an identity element 0 and additive inverses), and a multiplication operation (which is associative, commutative, and has an identity element $1 \neq 0$) that distributes over addition. The *order* of a ring R is the number of elements in R.

A set of elements $\{g_0, \ldots, g_{k-1}\}$ of a ring R are called *generators* of a ring-based block design if, whenever $i \neq j$, $g_i - g_j$ has a multiplicative inverse (denoted by $(g_i - g_j)^{-1}$). The tuples of a ring-based block design are indexed by pairs (y, x), where x is an arbitrary ring element and y is an arbitrary non-zero ring element. Given a ring R of finite order v and a set of generators as above, the tuple indexed by (y, x) is the set

$$T_{(y,x)} = \{y(g_i - g_0) + x \mid i = 0, \ldots, k-1\}.$$

The ring-based block design for R and a set of k generators is

$$\left\{T_{(y,x)} \mid x \in R, y \in R - \{0\}\right\}.$$

This set of tuples is a BIBD with $v(v-1)$ tuples [9]. If $v = \prod_{i=1}^{m} p_i^{n_i}$, where $p_1, p_2, \ldots, p_m$ are distinct primes, there exists a ring R of order v and a set of k generators in R if and only if $k \leq \min\{p_i^{n_i} \mid i = 1, \ldots, m\}$ [9]. Schwabe and Sutherland showed that R can be taken to be the cross product of finite fields $\mathrm{GF}(p_1^{n_1}) \times \mathrm{GF}(p_2^{n_2}) \times \ldots \times \mathrm{GF}(p_m^{n_m})$, with operations (addition and multiplication) defined component-wise. This ring will contain k generators for every k meeting the above condition. From this point forward, R will denote such a ring.

A ring-based data layout is obtained from a ring-based block design by ordering the tuples of the block design. Schwabe and Sutherland did not consider the problem of ordering the tuples or computing the disk and offset corresponding to a given address in a ring-based data layout. By default, disks and offsets could be computed by a series of lookups in a table consisting of the tuples arranged in some linear order, as discussed earlier. However, ring-based block designs have enough structure that the table of tuples does not have to be stored explicitly; the disks and offsets can be efficiently computed without such a table.

In the following, we will define an ordering of the tuples that associates each tuple with a rank from the set $\{0, 1, \ldots b-1\}$. This order allows us to compute table entries directly as needed rather than storing them.

To order the tuples of the ring-based block design, we will first define a bijection f from the ring R to the set $\{0, 1, \ldots, v-1\}$ that will identify each ring element with a unique integer. This will allow us to associate the index (y, x) of a tuple with the pair of integers $(f(y), f(x))$, so we can regard the tuples as being indexed by integers (j, i) where $i \in \{0, 1, \ldots, v-1\}$ and $j \in \{1, 2, \ldots, v-1\}$. To avoid confusion, when we are using such

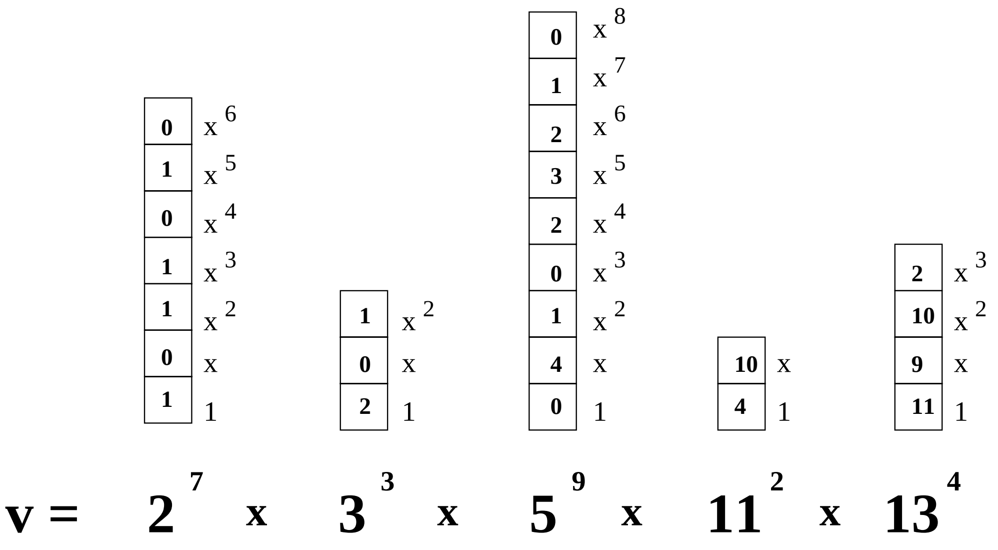

Figure 7: Representation of a ring element as a collection of polynomials, for $v = 2^7 \times 3^3 \times 5^9 \times 11^2 \times 13^4$.

a pair of integers as a tuple index, we will write the pair as $\langle j, i\rangle$ rather than (j, i). We then order the tuples by their indices $\langle 1,0\rangle, \langle 1,1\rangle, \ldots, \langle 1, v-1\rangle, \langle 2,0\rangle, \langle 2,1\rangle, \ldots, \langle 2, v-1\rangle, \ldots, \langle v-1,0\rangle, \langle v-1,1\rangle, \ldots, \langle v-1, v-1\rangle$.

The bijection f will use the following representation of the ring elements. The elements of the field $\mathrm{GF}(p^n)$ can be represented as polynomials of degree at most $n-1$ in a variable x with coefficients being integers mod p. Thus, a ring element is represented as an m–tuple of polynomials $(P_1(\mathsf{x}), \ldots, P_m(\mathsf{x}))$, where $P_i(\mathsf{x})$ has degree at most n_i and coefficients that are integers mod p_i, as illustrated in Figure 7.

The bijection f is defined as follows: If we evaluate P_i at p_i, the result will be an integer (denoted by $P_i(p_i)$) between 0 and $p_i^{n_i}-1$. (In this case, we are evaluating $P_i(p_i)$ over the integers, rather than the integers modulo p_i.) Repeating this for each of the polynomials, the result will be an m-tuple $(P_1(p_1), \ldots, P_m(p_m))$ of non-negative integers. The value of $f(P_1(\mathsf{x}), \ldots, P_m(\mathsf{x}))$ will be the rank of $(P_1(p_1), \ldots, P_m(p_m))$ under the lexicographic order. (Clearly, this yields a bijective mapping f between the ring elements and the integers from 0 to $v-1$). This rank is given by the expression $\sum_{i=1}^{m}\left(P_i(p_i)\cdot\prod_{j=i+1}^{m} p_j^{n_j}\right)$, which can be computed by the following algorithm `f`:

```
f($P_1$(x), ..., $P_m$(x))
    total = 0
    for $i$ = 1 to $m$
        total = total * $p_i^{n_i}$
        total = total + $P_i(p_i)$
    return total
```

Evaluating $P_i(p_i)$ takes time $O(n_i)$, and the remaining statements in each iteration of the "for" loop take constant time, so the total time for the m loop iterations is $\sum_{i=1}^{m} O(n_i) = O(\sum_{i=1}^{m} n_i) = O(\log v)$. Therefore the time to compute f is $O(\log v)$. (We have used the fact that $\sum_{i=1}^{m} n_i = O(\log v)$, since each p_i is at least two. Also, $m = O(\log v)$, since $m \le \sum_{i=1}^{m} n_i$.)

To compute the inverse of f, we must take an integer x and determine the m–tuple of polynomials $(P_1(\mathsf{x}), \ldots, P_m(\mathsf{x}))$ for which $(P_1(p_1), \ldots, P_m(p_m))$ will have rank x in the lexicographic order. The outer loop of the following algorithm `invf` computes the values $P_m(p_m)$, $P_{m-1}(p_{m-1})$, $\ldots$, $P_1(p_1)$, and the inner loop computes the coefficients of each P_i from $P_i(p_i)$:

```
invf($x$)
    for $i$ = $m$ to 1
        // $x_i$ is $P_i(p_i)$
        $x_i = x \bmod p_i^{n_i}$
        $x = x \text{ div } p_i^{n_i}$
        // the coefficients of $P_i$ are stored in $a(i,0),\ldots,a(i,n_i-1)$
        for $j$ = 0 to $n_i - 2$
            $a(i,j)$ = $x_i \bmod p^{j+1}$
            $x_i$ = $x_i$ div $p^{j+1}$
        $a(i,n_i-1)$ = $x_i$
    // the array $a$ stores the coefficients of the polynomials ($P_1$(x), ..., $P_m$(x))
    return $a$
```

The i^{th} iteration of the outer loop takes constant time, plus the time required for the inner loop, which is $\Theta(n_i)$. This yields a total of $\sum_{i=1}^{m}\Theta(n_i) = \Theta(\sum_{i=1}^{m} n_i) = O(\log v)$ steps to compute the inverse of f. Therefore we can convert ring elements into integers in $\{0,1,\ldots,v-1\}$ and vice versa in $O(\log v)$ steps.

The ordering of the tuples in the ring-based block design by their indices $\langle 1,0\rangle$, $\langle 1,1\rangle$, ..., $\langle 1,v-1\rangle$, $\langle 2,0\rangle$, $\langle 2,1\rangle$, ..., $\langle 2,v-1\rangle$, ..., $\langle v-1,0\rangle$, $\langle v-1,1\rangle$, ..., $\langle v-1,v-1\rangle$ defines the ring-based data layout. A table of these tuples would consist of $v(v-1)$ rows and k columns. Disks and offsets could be calculated using lookups in this table as described earlier. We now describe how use this order to compute disks and offsets without using table lookups.

3.2 Computational Complexity of Data Mappings

In order to compute a disk and offset for a particular data address, we must do the following:

1. Convert the address to a rank (row) and position (column) in the table;
2. Compute the numerical values of $f(y)$ (between 1 and $v-1$) and $f(x)$ (between 0 and $v-1$) corresponding to that rank;
3. Compute the ring elements y and x that index the tuple of that rank;
4. Compute the ring element in the desired position of that tuple;
5. Convert that ring element (which represents the disk identifier) to its numerical label;
6. Compute the offset of the desired address on that disk.

As was discussed earlier, Step 1 can be done in constant time with simple arithmetic operations. The values of $f(y)$ and $f(x)$ in Step 2 can be computed in constant time from the row as $f(y) = (row/v) + 1$ and $f(x) = (row \bmod v)$. The conversion to ring elements in Step 3 and back to numerical values in Step 5 both require a constant number of applications of the function f and its inverse, which take a total of $O(\log v)$ steps.

Step 4 must compute the element in the given position of the tuple indexed by ring elements y and x. This element is given by $y(g_j - g_0) + x$, where j is the given position. Computing this element from y, x, and the two generators g_j and g_0 requires one subtraction, one multiplication, and one addition of ring elements.

Recall that each ring element can be represented as an m–tuple of polynomials $(P_1(\mathsf{x}),\ldots,P_m(\mathsf{x}))$, where $P_i(\mathsf{x})$ has degree at most n_i and coefficients that are integers mod p_i. Each each polynomial represents an element from the field $\mathrm{GF}({p_i}^{n_i})$.

Addition in the field is polynomial addition, with coefficients added mod p_i. Clearly, adding two field elements will take $O(n_i)$ steps, as will subtracting two field elements. Multiplication in the field is polynomial multiplication, where the product is taken modulo some fixed irreducible polynomial of degree n_i (which much be stored), and all coefficients are computed mod p_i. Multiplying two field elements will therefore take $O(n_i \log n_i)$ steps for the initial multiplication (using, e.g., a Discrete Fourier Transform); evaluating the resulting product modulo an irreducible polynomial adds only $O(n_i \log n_i)$ more steps (see, e.g., von zur Gathen and Gerhard [11]).

Therefore, addition of ring elements takes $O(\sum_{i=1}^{m} n_i) = O(\log v)$ steps, and multiplication of ring elements takes $O(\sum_{i=1}^{m} n_i \log n_i) = O(\log v \log\log v)$ steps. Thus, Step 4 takes a total of $O(\log v) + O(\log v \log\log v) + O(\log v) = O(\log v \log\log v)$ steps.

Once the disk in the given position j of the tuple (y, x) is computed, its offset must be computed in Step 6. The offset is given by the number of occurrences of that disk in tuples with rank lower than the rank computed in Step 1. First we note that given the ordering of the tuples, each set $S_y = \{T_{(y,x)} \mid x \in R\}$ of tuples contains exactly k occurrences of each disk (once in each possible position in a tuple), so that the number of occurrences of disk d in tuples of rank lower than r is $k\lfloor r/v \rfloor$ (which is $k \cdot f(y)$), plus the number of occurrences of d in tuples of the form $T_{(y,x')}$, where $f(x') < f(x)$.

To compute this last term, we note that there are at most $k-1$ possible positions in which d can appear: $i \in \{0, \ldots, j-1, j+1, \ldots, k-1\}$. In each case, we must have $d = y(g_i - g_0) + x'$, or solving for x', $x' = d - y(g_i - g_0)$. If $f(x') < f(x)$, then $T_{(y,x')}$ contains disk d and has rank lower than that of $T_{(y,x)}$. So we compute x' for each of the $k-1$ positions other than j, and compare $f(x')$ to $f(x)$, keeping track of the number of positions for which the result is smaller than $f(x)$. This amount is added to $k \cdot f(y)$ to obtain the offset. This takes a total of $(k-1)(O(\log v) + O(\log v \log\log v) + O(\log v) + O(\log v)) = O(k \log v \log\log v)$ steps. Therefore, computing the disk and offset for a particular data address takes a total of $O(k \log v \log\log v)$ steps.

Since a polynomial of degree n_i can be stored in $O(n_i)$ space, the space required to store the m polynomials that make up a ring element is $O(\sum_{i=1}^{m} n_i) = O(\log v)$. The computation of the disk and offset requires $O(\log v)$ space for the various ring elements involved. In addition, $O(k \log v)$ space is needed to store the k generators, and $O(\log v)$ space to store the m irreducible polynomials, n_i's, and p_i's. This yields a total of $O(k \log v)$ space.

Alternately, to save the $O(k \log v)$ space for the generators, each generator could be constructed whenever needed in $O(\log v)$ steps, assuming that we use a particular canonical set of generators for each field. In particuar, to construct generator g_j: For each i from 1 to m, convert j into a base-p_i integer and use its digits as the coefficients of the i^{th} polynomial. These m polynomials together constitute g_j. This reduces the space requirements to $O(\log v)$ while keeping the same asymptotic running time of $O(k \log v \log\log v)$.

4 Reducing the Computational Complexity of Ring-Based Layouts

We can improve the $O(k \log v \log\log v)$ running time that we obtained using a polynomial representation of ring elements by using a different representation of ring elements and storing at most an additional $v-1$ integers. As before, each ring element is an m–tuple of field elements, with addition and multiplication defined component–wise. However, the individual field elements are represented differently.

By definition, the non-zero elements of a field form a group under multiplication. In a finite field GF(p^n), this group is actually a *cyclic* group of order $p^n - 1$. That is, the elements of the group are exactly $\{1, \alpha, \alpha^2, \ldots, \alpha^{p^n-2}\}$ for some non-zero field element α. (See Koblitz [5] or Lang [6] for further discussion of these concepts.) We can define a bijection $i \mapsto \bar{i}$ from $\{0, \ldots, p^n - 1\}$ onto GF(p^n) as follows:

$$\bar{i} = \begin{cases} 0 & \text{if } i = 0 \\ \alpha^{i-1} & \text{otherwise} \end{cases}$$

This bijection allows us to represent field elements as integers from 0 to $p^n - 1$; we will call this representation the *exponent* representation. Using the exponent representation of field elements, multiplication in the field can easily be performed in constant time, as follows:

$$\overline{i} \cdot \overline{j} = \begin{cases} \overline{0} & \text{if either } i \text{ or } j \text{ is } 0 \\ \overline{((i + j - 2) \bmod p^n - 1) + 1} & \text{otherwise} \end{cases}$$

Field addition can also be performed in constant time, but this requires a list of $p^n - 1$ precomputed integers to be stored. Note that if $i = 0$, then $\overline{i} + \overline{j} = \overline{j}$, and similarly if $j = 0$. However, suppose that $0 < i \leq j < p^n$ (i.e., neither i nor j is zero). Then

$$\begin{aligned} \overline{i} + \overline{j} &= \alpha^{i-1} + \alpha^{j-1} \\ &= \alpha^{i-1} \cdot (1 + \alpha^{j-i}) \end{aligned}$$

where $j - i \in \{0, \ldots, p^n - 2\}$. If we precompute a list of integers $e_0, \ldots, e_{p^n-2}$ where $\overline{e_l} = 1 + \alpha^l$ for each $l \in \{0, \ldots, p^n - 2\}$, then we can compute $\overline{i} + \overline{j}$ as $\overline{i} \cdot \overline{e_{j-i}}$. Thus, by using the precomputed list, field addition of non-zero elements can be reduced to field multiplication in constant time, and therefore can be performed in constant time.

Field negation (and therefore, subtraction) can also be performed in constant time. For fields GF(2^n), $-x = x$ for all field elements x, so negation is the identity mapping. For fields GF(p^n) with $p \neq 2$, $-1 = \alpha^{\frac{p^n-1}{2}}$, so negation is multiplication by the field element $\overline{\frac{p^n-1}{2}}$, which can be performed in constant time.

Thus, using the exponent representation for each of the m fields whose cross product makes up the ring of v elements, we can perform each ring addition, negation, and multiplication in $\Theta(m)$ steps, which is $O(\log v)$. Using the exponent representation for the ring, the time required for the calculation of disks and offsets is reduced from $O(\log v \log \log v)$ to $O(\log v)$.

The space required in this representation to store a ring element is m, which is $O(\log v)$. Computing generators as needed is simple in this representation, since we can just use $g_j = (\overline{j}, \overline{j}, \ldots \overline{j})$, for each j from 0 to $k - 1$. The exponent representation requires $\sum_{i=1}^{m} p_i^{n_i}$ space beyond that required to store ring elements; this quantity represents the total number of e_i's that must be stored. It is at most v, but will be smaller than v whenever $m > 1$. The total space required for the exponent representation is therefore $O(v)$.

Note that the $O(\log v)$ upper bound on the cost of ring operations and the $O(v)$ upper bound on the storage requirements for the exponent representation cannot be simultaneously realized. For example, if the number of integers stored is $\Omega(v)$, m must be $O(1)$, in which case the ring operations take only $\Theta(1)$ time each. (This is just a single example of a more general, but rather complicated, tradeoff.)

5 Comparisons between DATUM Layouts and Ring-Based Layouts

DATUM layouts require $\Theta(kv)$ time to compute disk numbers and offsets with space requirements of $\Theta(k)$. The implementation of ring-based layouts using the polynomial representation of ring elements requires less time – only $O(k \log v \log \log v)$ – to compute disk numbers and offsets, and the space requirements are $O(\log v)$. If $k = \Omega(\log v)$, then the space requirements for ring-based layouts are no greater than those of DATUM layouts. The implementation of ring-based layouts using the exponent representation of ring elements requires even less time, $O(k \log v)$, but more space, $O(v)$. Here, we must have $k = \Omega(v)$ for the space requirements not to exceed those of DATUM layouts. This comparison of the time and space requirements is summarized in Figure 8.

Recall that a DATUM layout has size $\binom{v-1}{k-1}$ for an array of v disks and stripe size k. The number of units on the type of disk being used must exceed this value for the layout to be usable. As we discussed earlier,

Layout	Time required	Space required
DATUM	$\Theta(kv)$	$\Theta(k)$
Ring-based (polynomial representation)	$O(k \log v \log\log v)$	$O(\log v)$
Ring-based (exponent representation)	$O(k \log v)$	$O(v)$

Figure 8: Comparisons of data mapping complexities.

this rules out the use of these layouts for many values of v and k that include commercially available array configurations. As arrays grow larger (and/or their constituent disks smaller), DATUM layouts will work for even fewer values of v and k.

On the other hand, when they exist for a particular v and k, ring-based layouts have size $k(v-1)$. Thus, for any v smaller than $\sqrt{10,000,000}$ (roughly 3,000), all existing ring-based layouts are usable. (Here we are using our rough definition that a layout is usable if it has size at most 10,000,000.) For larger v, existing layouts are usable as long as k is at most $10,000,000/(v-1)$. Thus, the usability of ring-based layouts will not be limited by disk sizes until array sizes grow by at least an order of magnitude. It is also more efficient to compute disks and offsets from data addresses in ring-based layouts than in DATUM layouts.

However, ring-based layouts do not exist for some values of v and k. In particular, as we stated earlier, if $v = \prod_{i=1}^{m} p_i^{n_i}$ for distinct primes p_i, then a ring-based layout for v and k exists if and only if k is less than or equal to $\min\{p_i^{n_i} \mid i = 1, \ldots, m\}$. When v is a prime number or a power of a prime, ring-based block designs (and therefore ring-based layouts) exist for all $k \leq v$. So, for example, if v is a power of 2, then we can construct a ring-based layout for any k.

6 Conclusions and Future Work

In this paper, we have demonstrated that the ring-based data layouts of Schwabe and Sutherland [9] can be implemented without using large tables to represent the collections of stripes in the layouts. In particular, we have taken advantage of the mathematical structure of ring-based block designs to directly compute the table entries as needed, rather than store the entire table. This work follows the development of DATUM layouts by Alvarez et al. [1], who developed explicit data mappings for their layouts. The advantage of ring-based layouts is that they are smaller and therefore more widely usable, and their data mappings are computationally more efficient.

In both this paper and the earlier work of Alvarez et al. on DATUM layouts, the data mappings took advantage of the mathematical structure of a collection of stripes to reduce the storage requirements of the layouts. This paper uses these structures to analyze and reduce the computational complexity of the mappings. However, there are many other BIBDs in the literature, which use a variety of mathematical techniques in their construction (see, e.g., Hanani [3]). In future work, we will investigate whether other BIBDs yield data layouts whose data mappings have small time and space requirements.

The particular orderings of tuples used in the construction of ring-based layouts were chosen to enable efficient algorithms for the direct computation of disks and offsets without using table lookups. However, the ordering of the tuples affects another property of the layout that we did not consider in this paper: the parallelism of the layout, defined as how evenly large reads that span many stripe units are distributed among the disks in the array.

Ring-based layouts have a high degree of parallelism for certain large reads that span v entire contiguous stripes (and thus a total of $(k-f)v$ contiguous stripe units). However, these reads are much larger than the reads of v contiguous stripe units for which the maximal parallelism condition is defined. We will consider other possible orderings of the tuples to see whether we can achieve maximal or nearly maximal parallelism without unduly affecting the computational efficiency that we have demonstrated.

References

[1] G. A. Alvarez, W. A. Burkhard, and F. Cristian, "Tolerating Multiple Failures in RAID Architectures with Optimal Storage and Uniform Declustering," in *Proceedings of the 24th ACM/IEEE International Symposium on Computer Architecture*, pp. 62–71 (1997).

[2] G. A. Alvarez, W. A. Burkhard, L. J. Stockmeyer, and F. Cristian, "Declustered Disk Array Architectures with Optimal and Near-Optimal Parallelism," in *Proceedings of the 25th ACM/IEEE International Symposium on Computer Architecture*, pp. 109–120 (1998).

[3] H. Hanani,"Balanced Incomplete Block Designs and Related Designs," *Discrete Mathematics*, Vol. 11, pp. 255–369 (1975).

[4] M. Holland and G. A. Gibson, "Parity Declustering for Continuous Operation in Redundant Disk Arrays," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 23–35 (1992).

[5] N. Koblitz, "A Course in Number Theory and Cryptography," Springer-Verlag (1987).

[6] S. Lang, "Algebra (3rd edition)," Addison Wesley Longman, Inc. (1992).

[7] R. R. Muntz and J. C. S. Lui, "Performance Analysis of Disk Arrays Under Failure," in *Proceedings of the 16th Conference on Very Large Data Bases*, pp. 162–173 (1990).

[8] D. A. Patterson, G. A. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *Proceedings of the Conference on Management of Data*, pp. 162–173 (1990).

[9] E. J. Schwabe and I. M. Sutherland, "Improved Parity-Declustered Data Layouts for Disk Arrays," *Journal of Computer and System Sciences*, Vol. 53, No. 3, pp. 328–343 (1996).

[10] L. Stockmeyer, "Parallelism in Parity-Declustered Layouts for Disk Arrays," Technical Report RJ9915, IBM Almaden Research Center (1994).

[11] J. von zur Gathen and J. Gerhard, "Modern Computer Algebra," Cambridge University Press (1999).