Efficient data mappings for parity-declustered data layouts

Eric J. Schwabe*, 1, Ian M. Sutherland
School of CTI, DePaul University, 243 S. Wabash Avenue, Chicago, IL 60604, USA
Received 10 October 2003; accepted 20 February 2004

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. Such a data layout is generated by partitioning the data on the disks into stripes and choosing a part of each stripe to hold redundant information. Thus the data layout can be represented as a table of stripes. The data mapping problem is the problem of translating a data address into a disk identifier and an offset on that disk. Recent work has yielded mappings that compute disks and offsets 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 and wider applicability.

Keywords: Disk arrays; RAID; Data layouts; Parity-declustering

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 [7]. Since each disk in the array may fail independently with some probability per

* Corresponding author.
E-mail addresses: eschwabe@cs.depaul.edu (E.J. Schwabe), sans_peur2001@yahoo.com (I.M. Sutherland).

0304-3975/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
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 et al. [7]). This technique is illustrated in Fig. 1. Each disk is divided into units, and in each 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 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 a 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 [6], Schwabe and Sutherland [9], Stockmeyer [10], and Alvarez...
et al. [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 a disk fails, the additional workload on the array generated by the reconstruction of its lost data must be evenly distributed over the surviving disks.

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 data 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 1–3. Conditions 4–6 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 et al. [2] showed that Conditions 4 and 5 can only be realized simultaneously when either \( k \) is in the set \( \{ v, v - 1, 2 \} \) or \((k, v) = (3, 5), (3, 7), \) 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]): First, each element appears the same number of times (called \( r \)) among the \( b \) tuples. Second, each pair of elements appears the same number of times (called \( \lambda \)) 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 Fig. 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 in the layout. We call \( r \) the size of the layout. If the number of units on the actual disks is greater than \( r \), we simply cover the disks as completely as possible with copies of the data layout. For each pair of disks, there are exactly \( \lambda \) 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 \( \lambda \) 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 et al. [1] demonstrated a method for computing the values to be stored in each of the \( f \) chosen redundancy units from 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 primarily 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, consisting entirely of data units, with a linear address space. Data addresses in this space are mapped to disks and offsets on those disks.

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 entry in each row, which is a redundancy unit. (Thus \( k - 1 \) 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 Fig. 3 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 \( \text{row} = \lfloor \text{address} / (k - 1) \rfloor \) and \( \text{column} = \text{address} \mod(k - 1) \). Next, we use the contents of the table to determine the disk and offset where that address is located. (For an \( f \)-fault-tolerant disk array, just replace each “\( k - 1 \)” in the above discussion with a “\( k - f \)”.)

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 appear in rows above the computed row. (Furthermore, if the data layout has been replicated to fill an array of large disks, we must also take into account how many offsets are filled by other copies of the layout.) Offsets can be precomputed while the table is being constructed, requiring additional work proportional to the size of the table, and if this is done then the
Fig. 3. A table of stripes derived from the complete block design with \( v = 5 \) and \( k = 4 \), and the parity-declustered data layout derived from it. Each table entry shows “disk number (offset) (address)” for one unit. Shaded units store redundant information, and so have no data addresses assigned to them.

offset can also be determined with a single table lookup. 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.

1.3. Our results

This paper considers ways to reduce this space requirement by using data layouts that do not require the storage of tables of stripes. Alvarez et al. [1] proposed the DATUM layouts for this purpose, but did not consider the computational complexity of their data mappings nor the usability of the layouts for large arrays. We review these layouts in Section 2.

We present an alternative in Section 3: ring-based data layouts. Both DATUM layouts and ring-based layouts take advantage of their mathematical structure to compute disks and offsets directly. We analyze the computational complexity of the data mappings of ring-based layouts as well as those of DATUM, and show in Section 5 that ring-based layouts have smaller time complexity than DATUM layouts (\( O(k \log v) \) versus \( \Theta(kv) \) in the word model, and \( O(k^2 + \log^2 v \log \log v \log \log \log v) \) versus \( \Omega(k^2 \log (v - \log k)) \) in the bit model). Ring-based layouts are also applicable to a wider range of array configurations than DATUM layouts.

2. DATUM layouts

Alvarez et al. [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. In the following, we describe their construction and analyze the complexity of their data mappings.
2.1. Layout construction and data mapping complexity

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 (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: \( \text{loc} (X_1, \ldots, X_k) \), which computes the rank of an input tuple, and its inverse, \( \text{invloc} (\text{rank}) \), which computes the \( k \) elements of the tuple with a particular rank. The function \( \text{invloc} \) can be computed using the following algorithm:

\[
\text{invloc} (\text{rank})
\]

\[
\text{for } (\text{int } i = k; i >= 1; i--)
\]

\[
l = i
\]

\[
\text{while } ( (\binom{l}{i}) <= \text{rank} )
\]

\[
l = l + 1
\]

\[
X_i = l - 1
\]

\[
\text{rank} = \text{rank} - (\binom{l-1}{i})
\]

\[
\text{return } (X_1, X_2, \ldots, X_k)
\]

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

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

Alvarez et al. established the correctness of their algorithms and formulas, but did not analyze their computational complexity.

2.2. Computational complexity of data mappings

To determine the worst-case running time of \( \text{invloc} \), 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 \( \#X_{\text{col}} \) is dominated by the time to compute \( k - 1 \) 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 \( \text{invloc} \) 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 \( \binom{l}{i} = 1 \),
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 $O(1)$ steps for each iteration of the inner loop after the first (which still takes $O(v)$ steps). This reduces the total time requirements for $\text{invloc}$, and thus for the entire process of computing the disk and offset from $\Theta(kv^2)$ to $\Theta(kv)$ in the word model.

### 2.3. Usability of layouts for large disk arrays

DATUM layouts eliminate the need to store a table of size polynomial in $v$ in exchange for enough space to store a tuple $(X_1, X_2, \ldots, X_k)$ and $O(kv)$ time to compute disks and offsets. These layouts can be constructed for all possible values of $v$ and $k$, but they may be too large to use. DATUM layouts contain $b = \binom{v}{k}$ stripes, so each disk in the array must contain $bk/v = \binom{v-1}{k-1}$ units in order for the layout to be used.

Consider an array of 10 GB disks where each unit contains 4 KB. The number of units on each disk is therefore $\frac{10 \cdot 2^{30}}{4 \cdot 2^{10}} = 2.5 \cdot 2^{20} = 2,621,440$, so any layout with size greater than this amount cannot be used. We are only looking for a rough guideline for usability, so we will consider any layout with size at most 10 million to be usable.

In Fig. 4, we give values of $k$ for which the DATUM layout is usable, for various array sizes $v$. Even for moderately sized arrays that are commercially available (e.g., $v = 64$ disks), DATUM layouts are too large to be usable for more than 80% of the 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. In those cases, 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 usability guidelines. For 10 TB disks with units of size 4 KB, which would yield a layout size of 2,684,354,560 (which we will round up to 10 billion), the results would be as indicated in Fig. 5. Increasing the bound on the number of units per disk by a factor of over 1000 only makes a few more values of $k$ feasible once the array size reaches 64.

<table>
<thead>
<tr>
<th>$v$</th>
<th>$k$</th>
<th>% usable</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>all</td>
<td>100.000</td>
</tr>
<tr>
<td>16</td>
<td>all</td>
<td>100.000</td>
</tr>
<tr>
<td>32</td>
<td>1 ... 9, 24 ... 32</td>
<td>56.250</td>
</tr>
<tr>
<td>64</td>
<td>1 ... 6, 59 ... 64</td>
<td>18.750</td>
</tr>
<tr>
<td>128</td>
<td>1 ... 4, 125 ... 128</td>
<td>6.250</td>
</tr>
<tr>
<td>256</td>
<td>1 ... 4, 253 ... 256</td>
<td>3.125</td>
</tr>
</tbody>
</table>

Fig. 4. DATUM layouts with size at most 10 million.
Parity declustering was introduced to reduce the time to reconstruct lost data without dedicating excessive amounts of space to redundant information, by using an intermediate value of $k$ between 2 and $v$. DATUM layouts for large disk arrays can only be used with values of $k$ that are close to 2 or close to $v$. If $k$ is close to $v$, parity declustering will yield little improvement in the reconstruction time. If $k$ is close to 2, a large fraction of the array will be dedicated to redundant information. Thus, for large disk arrays, using DATUM layouts forces a choice between two undesirable options, rather than allowing values of $k$ that balance the need for fast reconstruction against the need for less redundant storage.

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 and a smaller amount of wasted space when the disk size is not an integral multiple of the layout size.

3. Ring-based layouts

In this section, we define data mappings for the ring-based data layouts of Schwabe and Sutherland [9] that 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 $O(k \log v)$ rather than $O(kv)$ in the word model and $O(k \log^2 v + \log^2 v \log \log v \log \log v \log v)$ rather than $O(k^2 v (\log v - \log k))$ in the bit model.

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 (associative, commutative, and having an identity element 0 and additive inverses), and a multiplication operation (associative, commutative, and having an identity element 1 ≠ 0) that distributes over addition. The order of a ring \( R \) is the number of elements in \( R \).

The elements of a set \( \{g_0, \ldots, g_{k-1}\} \) of ring elements are called generators of a ring-based block design if, whenever \( i \neq j \), \( g_i - g_j \) has a multiplicative inverse. 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 order \( v \) and a set of generators \( \{g_0, \ldots, g_{k-1}\} \) 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 is \( \{T_{(y,x)} \mid x \in R, y \in R - \{0\}\} \).

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 \( \text{GF}(p_1^{n_1}) \times \text{GF}(p_2^{n_2}) \times \cdots \times \text{GF}(p_m^{n_m}) \), with operations defined component-wise. The ring will contain \( k \) generators for every \( k \) that satisfies 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 from 0 to \( b - 1 \). To do this, 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 that 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 a pair of integers as a tuple index, we will write the pair as \((j, i)\) rather than \((i, j)\). We then order the tuples by their indices \((1, 0), (1, 1), \ldots, (1, v - 1), (2, 0), (2, 1), \ldots, (2, v - 1), \ldots, (v - 1, 0), (v - 1, 1), \ldots, (v - 1, v - 1)\).

The bijection \( f \) will use the following representation of the ring elements. The elements of the field \( \text{GF}(p^n) \) 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 \( (P_1(x), \ldots, P_m(x)) \), where \( P_i(x) \) has degree at most \( n_i \) and coefficients that are integers mod \( p_i \), as illustrated in Fig. 6.

The bijection \( f \) is defined as follows: Evaluate each polynomial \( P_i \) at \( p_i \), to obtain an \( m \)-tuple \((P_1(p_1), \ldots, P_m(p_m))\) of non-negative integers. The value of \( f(P_1(x), \ldots, P_m(x)) \) will be the rank of \((P_1(p_1), \ldots, P_m(p_m))\) in 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 \( \sum_{i=1}^m (P_i(p_i) \cdot \prod_{j=i+1}^m p_j^{n_j}) \), which can be computed by the
following algorithm \( f \):

\[
\begin{align*}
&\text{\( f(P_1(x), \ldots, P_m(x)) \)} \\
&\quad \text{total} = 0 \\
&\quad \text{for } i = 1 \text{ to } m \\
&\quad \quad \text{total} = \text{total} \times P_i^{p_i} \\
&\quad \quad \text{total} = \text{total} + P_i(p_i) \\
&\quad \text{return total}
\end{align*}
\]

Each iteration of the “for” loop takes \( O(n_i) \) steps (for the polynomial evaluation), so the total time for the \( m \) loop iterations is \( \sum_{i=1}^{m} O(n_i) = O(\log v) \). 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 \( m \leq \sum_{i=1}^{m} n_i \).)

To compute the inverse of \( f \), we must take an integer \( x \) and determine the \( m \)-tuple of polynomials \( (P_1(x), \ldots, P_m(x)) \) 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 \( \text{invf} \) 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 \( P_i(p_i) \):

\[
\begin{align*}
&\text{invf}(x) \\
&\quad \text{for } i = m \text{ to } 1 \\
&\quad \quad // x_i \text{ is } P_i(p_i) \\
&\quad x_i = x \mod p_i^{n_i} \\
&\quad x = x \div p_i^{n_i} \\
&\quad // the coefficients of } P_i \text{ are stored in } a(i,0), \ldots, a(i,n_i-1) \\
&\quad \text{for } j = 0 \text{ to } n_i - 2 \\
&\quad \quad a(i,j) = x_i \mod p_i^{j+1} \\
&\quad x_i = x_i \div p_i^{j+1} \\
&\quad a(i,n_i - 1) = x_i \\
&\quad // the array } a \text{ stores the coefficients} \\
&\quad // of the polynomials } (P_1(x), \ldots, P_m(x)) \\
&\quad \text{return } a
\end{align*}
\]
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 $\sum_{i=1}^{m} \Theta(n_i) = O(\log v)$ 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 $(1,0), (1,1), \ldots, (1,v-1), (2,0), (2,1), \ldots, (2,v-1), \ldots, (v-1,0), (v-1,1), \ldots, (v-1,v-1)$ defines the ring-based data layout. A table of these tuples would consist of $v(v-1)$ rows and $k$ columns. We now describe how to 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:

(1) Convert the address to a rank ($row$) and a position ($col$) in the table;
(2) Compute the numerical values of $f(y)$ and $f(x)$ 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.

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 $f(y) = \lfloor row/v \rfloor$ and $f(x) = (row \mod v)$. 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 $y(g_{col} - g_0) + x$. 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.

Addition in the field is polynomial addition, with coefficients added mod $p_j$. Clearly, adding or subtracting two field elements will take $O(n_i)$ steps. Multiplication in the field is polynomial multiplication, where the product is taken modulo some fixed irreducible polynomial of degree $n_i$ (which must be stored), and all coefficients are computed mod $p_j$. 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 and subtraction of ring elements take $O(\sum_{i=1}^{m} n_i) = O(\log v)$ 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 $O(\log v \log \log v)$ steps.

The offset computed in Step 6 is given by the number of occurrences of the 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 $row$ is $k \cdot \lfloor row/v \rfloor$ (which is $k \cdot f(y)$), plus the number of occurrences of $d$ in tuples of the form $T_{(y,x')}$, where $f(x') < f(x)$. The offset is therefore $O(\log v)$ steps.
To compute this last term, we note that there are at most \( k - 1 \) other positions in which \( d \) can appear: \( i \in \{0, \ldots, \text{col} - 1, \text{col} + 1, \ldots, k - 1\} \). 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 \( T_{y,x'}(y,x) \) contains disk \( d \) and has rank lower than that of \( T_{y,x}(y,x) \). So we compute \( x' \) for each of the \( k - 1 \) positions other than \( \text{col} \), 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 in the word model.

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 is needed 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 particular, we can construct generator \( g_j \) as follows: 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 requirement to \( O(\log v) \) while keeping the same asymptotic running time of \( O(k \log v \log \log v) \).

### 3.3. Reducing the time complexity

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; in particular, each field element is represented as either an integer or \(-\infty\). This representation allows us to multiply two field elements with a single integer addition, add two field elements with two integer additions and a table lookup, and negate a field element with at most one integer addition.

By definition, the non-zero elements of a field form a group under multiplication. In a finite field \( \text{GF}(p^n) \), 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 \( \alpha \). (See Koblitz [5] for further discussion of these concepts.) Given such an \( \alpha \), the **Zech logarithm** of a non-zero element \( x \) is defined to be the unique \( i \in \{0, 1, \ldots, p^n - 2\} \) such that \( x = \alpha^i \). We define the Zech logarithm of 0 to be \(-\infty\), and \( \alpha^{-\infty} \) to be 0. For any non-negative integer, denote \( \{-\infty, 0, \ldots, u - 1\} \) by \( L_u \). The Zech logarithm is a bijection from \( \text{GF}(p^n) \) to \( L_{p^n - 1} \) whose inverse is \( i \mapsto \alpha^i \). This bijection allows us to represent field elements as elements of \( L_{p^n - 1} \). We will call this representation the **Zech logarithm representation** of field elements.

Field multiplication can be performed in constant time in the Zech logarithm representation. Define addition on \( L_u \) to be addition modulo \( u \) except that \(-\infty + i = i + -\infty = -\infty\); then \( \alpha^{i} \cdot \alpha^{j} = \alpha^{i+j} \) for any \( i \) and \( j \) in \( L_{p^n - 1} \). Thus, multiplication of field elements
corresponds to addition of their Zech logarithms, and addition of Zech logarithms can be performed in constant time.

Field addition can also be performed in constant time, but requires some additional space. Since addition of 0 can obviously be done in constant time, we need only show how to add non-zero elements in constant time. Suppose \( i \) and \( j \) are elements of \( L_{p^n-1} - \{-\infty\} \); then \( \alpha^i + \alpha^j = \alpha^j(\alpha^{−i} + 1) \), where \( i - j \) is computed modulo \( p^n - 1 \). Let \( k \) be the Zech logarithm of \( \alpha^{−i} + 1 \); then \( \alpha^i + \alpha^j = \alpha^j \cdot \alpha^k = \alpha^{j+k} \). If we precompute a list \( e_0, \ldots, e_{p^n-2} \) of elements of \( L_{p^n-1} \) such that \( \alpha^{e_i} = \alpha^i + 1 \) for all \( i \in L_{p^n-1} - \{-\infty\} \), then the Zech logarithm representation of \( \alpha^i + \alpha^j \) is just \( j + e_{i-j} \), which can be computed in constant time.

Field negation (and therefore, subtraction) can also be performed in constant time. Negation is simply multiplication by \(-1\), so negation requires only knowing the Zech logarithm of \(-1\). For fields \( GF(2^n) \), \(-1 = \alpha^{p^n-1} \), so the Zech logarithm of \(-1\) is \( p^n-1 \). For fields \( GF(p^n) \) with \( p \neq 2 \), \(-1 = \alpha^{p^n-1}/2 \), so the Zech logarithm of \(-1\) is \( p^n-1/2 \).

Ring elements can be represented by \( m \)-tuples of field elements, where each field element is represented by its Zech logarithm. We will call this the Zech logarithm representation of ring elements. In this representation, we can perform ring addition, subtraction, and multiplication in \( \Theta(m) = O(\log v) \) steps.

To use the Zech logarithm representation to compute disks and offsets for a ring-based data layout, we must describe a bijection \( f' \) from the ring \( R \) to the set \( \{0, 1, \ldots, v-1\} \) corresponding to the bijection \( f \) described in Section 3.1, and analyze the time complexity of computing \( f' \) and its inverse when using the Zech logarithm representation of ring elements.

First, for \( i \in L_u \), define a non-negative integer \( \overline{i} \) as follows:

\[
\overline{i} = \begin{cases} 
0 & \text{if } i = -\infty, \\
i + 1 & \text{otherwise.}
\end{cases}
\]

Thus, \( i \mapsto \overline{i} \) is a bijection from \( L_u \) to \( \{0, \ldots, u\} \).

Given an \( m \)-tuple \( (l_1, \ldots, l_m) \), where \( l_i \in L_{p^n_i-1} \) for each \( i \in \{1, \ldots, m\} \), \( f'(l_1, \ldots, l_m) \) will be the rank of \( (\overline{l_1}, \ldots, \overline{l_m}) \) in the lexicographic ordering. This is clearly a bijection from \( R \) to \( \{0, \ldots, v-1\} \). This rank is given by the expression \( \sum_{i=1}^{m} \left( \overline{l_i} \cdot \prod_{j=i+1}^{m} p^n_j \right) \), which can be computed in \( O(\log v) \) time using an algorithm similar to that for computing \( f \) described in Section 3.1 (where \( P_i(p_i) \) is replaced by \( \overline{l_i} \)). The inverse of \( f' \) can be computed in \( O(\log v) \) time using the following algorithm:

```plaintext
invfprime(x)
for i = m to 1
    a[i] = x mod p^n_i
    // a[i] is now l_i
    if a[i] = 0
        then a[i] = -\infty
        else a[i] = a[i] - 1
    // a[i] is now \overline{l_i}
    x = x \div p^n_i
return a
```

Using the Zech logarithm representation for the ring, the time required for the calculation of disks and offsets is reduced from $O(k \log v \log \log v)$ to $O(k \log v)$ in the word model.

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 $g_j = (j-1, j-1, \ldots, j-1)$, for each $j$ from 1 to $k-1$, and $g_0 = (-\infty, -\infty, \ldots, -\infty)$. The Zech logarithm representation requires $\sum_{i=1}^{m} (p_i^n - 1)$ 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 Zech logarithm 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 Zech logarithm 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.)

4. Computational complexity of data mappings in the bit model

The complexity results given in the previous sections were stated in the word model, where all arithmetic operations have unit cost, regardless of the size of the quantities involved. In the bit model, we consider how many bits are needed to represent each quantity, and compute the cost of each operation as the number of individual bit operations required. (This model is commonly used when considering operations on elements of a finite field.)

Suppose that the linear address space seen by the user contains $A$ addresses. This quantity will generally be larger than the total number of data units $L$ in the layout being used. When we use several copies of the layout to cover the disks in such an array, the first step in computing the disk number and offset for a particular address is to determine which copy of the layout contains the address in question ($\lfloor A/L \rfloor$) and its address within that copy ($A \mod L$). Conversely, once we have computed the disk and offset within that copy of the layout, this value must be modified by adding to it the total number of offsets on the disk that are covered by earlier copies of the layout ($L \cdot \lfloor A/L \rfloor$). In the word model, all of these operations would together take only constant time, so we did not consider them and treated the data layout as though it filled the entire array. However, since these steps work with some of the largest values used in the process of computing the disk and offset, we must take their costs into account when using the bit model.

Throughout this discussion, we will use the facts that we can add or subtract two $n$-bit numbers in $\Theta(n)$ steps, and multiply or divide two $n$-bit numbers in $\Theta(n \log n \log \log n)$ steps (using the Schonhage–Strassen algorithm [8]). Thus for any layout, since $L \leq A$, the initial computation of $\lfloor A/L \rfloor$ and $A \mod L$, and the final addition of $L \cdot \lfloor A/L \rfloor$ to the offset will both take $\Theta(\log A \log A \log \log A)$ steps. This bound will hold for both the DATUM layouts and ring-based layouts under either representation, as will a space requirement of $\Theta(\log A)$.

For the computation of the disk number and offset within a single DATUM layout, the total number of tuples is $\binom{v}{k}$, so the number of bits in $L$ is $\log((k-1)\binom{v}{k}) = \Theta(k \log v - \log k)$. A constant fraction of the $\Theta(kv)$ operations performed by the \texttt{invloc} function
are multiplications of two numbers, one of which is between \((\binom{v/2}{k/2})\) and \((\binom{v}{k})\). Each of these multiplications must take \(\Omega(\log (\binom{v/2}{k/2})) = \Omega(k(\log v - \log k))\) steps, so the complexity of the data mapping will be \(\Omega(k^2v(\log v - \log k))\).

For the computation of the disk number and offset within a single ring-based layout, the total number of tuples is \(v(v-1)\), so the number of bits in \(L\) is \(\log((k-1)v(v-1)) = \Theta(\log v)\). Using the polynomial representation, each of the operations in the computation has at most the complexity of the multiplication of two \(\Theta(\log v)\)-bit numbers. Thus the bit-model complexity of the computation is at most \(\Theta(k \log v \log \log v) \cdot \Theta(\log v \log v \log v \log \log v) = \Theta(k \log^2 v \log^2 \log v \log \log v \log v)\). Using the Zech logarithm representation, the same reasoning would yield a bit-model complexity of \(\Theta(k \log v) \cdot \Theta(\log v \log v \log v \log \log v) = \Theta(k \log^2 v \log v \log \log v \log v)\). However, since all but \(m = O(\log v)\) of the multiplications required are implemented as additions in this representation, the complexity is actually \(\Theta(k \log^2 v + \log^2 v \log v \log \log v \log \log v)\).

The space requirements for the computation of the disk number and offset (within one layout) for the DATUM layout will be \(\Theta(k \log v)\). This is the number of bits that will be required to store the tuple elements \((X_1, X_2, \ldots, X_k)\), which is also sufficient to store any of the binomial coefficients generated by the function \(\text{invloc}\). For the computation of the disk number and offset for ring-based layouts using either representation, the space requirements will exceed those that were calculated for the word model by a factor of \(\Theta(\log v)\), since all of the values stored are at most \(v\). This yields \(O(\log^2 v)\) and \(O(v \log v)\), respectively.

5. Comparisons between DATUM and ring-based layouts

In the word model, 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 Zech logarithms for the 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 in the word model is summarized in Fig. 7.

In the bit model, DATUM layouts require \(\Omega(k^2v(\log v - \log k))\) time to compute disk numbers and offsets with space requirements of \(\Theta(k \log v)\). The implementation of ring-based layouts using the polynomial representation of ring elements requires less time—only \(O(k \log^2 v \log^2 \log v \log \log v)\)—to compute disk numbers and offsets, and the space requirements are \(O(\log^2 v)\). The implementation of ring-based layouts using Zech logarithms for the representation of ring elements requires even less time, \(O(k \log^2 v + \log^2 v \log \log v \log \log v)\), but more space, \(O(v \log v)\). This comparison of the time and space requirements in the bit model is summarized in Fig. 8.

Some of the algorithms used in the data mappings presented here, such as the Discrete Fourier Transform, while asymptotically optimal, may run too slowly in practice for some
arrays. In such cases, simpler algorithms can be used that are asymptotically suboptimal, but whose asymptotic running times do not hide large constant factors. In the word model, if the multiplications of degree-$n$ polynomials are done with the straightforward $\Theta(n^2)$ algorithm, then the running time for the polynomial representation will become $O(k \log^2 v)$, with a small leading constant in the asymptotic notation. The running time for the Zech logarithm representation will not be affected. In the bit model, if both the multiplications of degree-$n$ polynomials and the multiplications of $n$-bit numbers are done with the straightforward $\Theta(n^2)$ algorithms, then the running time for the polynomial representation will become $O(k \log^4 v)$ and for the Zech logarithm representation will become $O(k \log^2 v + \log^3 v)$, both with small leading constants.

Recall that a DATUM layout has size $(v-1)k/(k-1)$ 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, 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 (using the rough definition of usability discussed earlier). For larger $v$, the 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 more than 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.

### References