RJ 10322 (A0408-005) August 9, 2004
Computer Science

IBM Research Report

R5X0: An Efficient High Distance Parity-Based Code with Optimal Update Complexity

Jeff R. Hartline
Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

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

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

R5X0: AN EFFICIENT HIGH DISTANCE PARITY-BASED CODE WITH OPTIMAL UPDATE COMPLEXITY

Jeff R. Hartline

Department of Computer Science
Cornell Univeristy
Ithaca, NY 14853-7501
email: jhartlin@cs.cornell.edu

Tapas Kanungo

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
email: kanungo@us.ibm.com

James Lee Hafner

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
email: hafner@almaden.ibm.com

ABSTRACT: We present a new class of array codes based on a generalization of the RAID5 XOR parity code. The $R5X0(n, r, p)$ code protects an array of n data disks and p parity disks with r symbols per column from as many as p arbitrary column erasures. Decoding and encoding in $R5X0$ is computed using only XOR and cyclic shift operations. $R5X0$ is a simple geometric generalization of RAID5 and has optimal update complexity (the number of parity symbols affected by a single data symbol is exactly p) and asymptotically optimal storage efficiency for its distance. The only geometric constraint on the $R5X0$ layout is that $r \geq (p-1) \cdot n$.

Keywords: Erasure codes, array codes, storage systems, RAID, reliability, performance

1. Introduction

We consider the problem of building reliable codes in the storage system model. The storage model emphasizes three principle metrics: reliability, storage efficiency, and performance. The reliability of an array code is a function of its column distance. A code of column distance d can recover from the erasure of $d-1$ entire disks without data loss. The storage efficiency of a code is the number of independent data symbols divided by the total number of symbols used by the code. We measure a code's performance via its average update complexity (UC): the average number of parity symbols affected by a change in a data symbol. (However, the worst-case UC could also be used instead if that is more appropriate for a paritcular application.) Update complexity affects the number of IOs required to modify a data symbol, which in turn affects the throughput of the storage system.

Reed-Solomon [RS60] codes have been proposed for the storage model [Pla97], but they require finite field arithmetic and are therefore impractical without special purpose hardware. Various new codes like Turbo codes [Mac], Tornado codes [LMSS01], LT codes [Lub02], and Raptor codes [Sho03] have been proposed in the communication literature, but their probabilistic nature does not lend itself well to the storage model. Furthermore, the communication model puts stress on the computational cost of encoding and decoding as opposed to the cost of IO seeks, which dominate in storage systems.

Array codes [BFv98] are perhaps the most applicable codes for the storage model, where large amounts of data are stored across many disks, and the loss of a data disk corresponds to the loss of an entire disk of symbols. Array codes are two-dimensional burst error-correcting codes that use XOR parity along lines at various angles. While Low Density Parity Check (LPDC) codes [Gal62, Mac] were originally invented for communication purposes, the concepts have have been applied to the storage system framework. Convolution array codes [BFv98, FHB89] are a type of array code, but these codes assume semi-infinite length tapes of data and reconstruction progresses sequentially over these tapes, and their parity elements are not independent. These codes are not directly applicable to the storage model where the efficient reconstruction of randomly located data blocks is required.

Conventional RAID algorithms, such as RAID5, RAID51, and RAID6, are another type of array code [HP03, Mas97]. In RAID5, any number of data disks can be protected against a single disk erasure by adding a single disk containing a parity check of the data disks. Previously known high distance extensions to RAID5 are too inefficient to be practical in most storage system environments. For instance, RAID51 achieves column distance 4 by mirroring the RAID5 layout, effectively halving the storage efficiency.

Maximum Distance Separable (MDS) codes, or codes with optimal storage efficiency, have been proposed in the literature [BR99, BBBM95, XB99, ZZS83]. The Blaum-Roth (BR) code [BR99], EvenOdd (EO) code [BBBM95], and Row-Diagonal Parity (RDP) [CEG$^+$04] are all distance three codes and achieve optimal storage efficiency but have non-optimal update complexity. The X-Code (XC) [XB99] and ZZS code [ZZS83] achieve both optimal storage efficiency and optimal update complexity but do not generalize to distances greater than three. A variant of the EvenOdd code [BBV96] achieves column distances greater

than three for certain array dimensions, but still has non-optimal update complexity. The recently patented code from LSI Logic, Inc. [Wil01] provides a non-MDS simple distance 3 code with good update complexity, but only 50% storage efficiency, which is marginally better than RAID51.

The R5X0 code relaxes the MDS constraint by an arbitrarily small amount to achieve optimal update complexity for any distance with very few array constraints. In particular, while EO, BR and RDP codes restrict the number of rows to $p-1$, (where p is a prime and the number of data disks $n \leq p$), R5X0 allows for any n provided the number of rows be greater than $(q-1)n$ where q is the number of parity disks. Furthermore, the redundancy scheme for R5X0 is a simple geometric generalization of RAID5 parity and has an intuitive proof of reconstruction for the general case.

The R5X0 code description, decoding algorithm, and its proof of correctness are described in Section 2. In Section 3 we discuss the properties of R5X0 codes and compare these properties against several other codes. Next, a few efficiency-boosting optimizations for the general R5X0 code are described in Section 4.

2. R5X0 Description

2.1. The R5X0 Encoding

The $R5X0(n, r, p)$ code layout consists of $n+p$ disks and r rows (symbols on a disk). The number of data disks is n the number of parity disks is p, and the constraint on the number of rows r is $r \geq (p-1) \cdot n$. Each symbol on the disk can be of arbitrary size. That is, a symbol can be either a bit, or a byte, or a sector, etc. We denote the jth data disk by D^j, $0 \leq j < n$, and the ith data symbol in the jth disk by D_i^j, $0 \leq i < r$. Similarly, we denote the parity disks by P^k, $0 \leq k < p$. The $R5X0$ code enforces the following constraints:

$$P_i^k = \bigoplus_{j=0}^{n-1} D^j_{\langle i-j\cdot k\rangle_r} \ ; \tag{1}$$

$$D_i^j = 0 \text{ for } 0 \leq j < n \text{ and } r - j\cdot(p-1) \leq i < r \ . \tag{2}$$

where $\langle x\rangle_r$ denotes x modulo r. We denote (1) as our *parity constraint* and (2) as our *preset constraint*. The parity constraint assigns to disk P^k the RAID5 parities of the $r \times n$ data matrix taken along diagonals of slope k with modulo r wrap-around. The preset constraint assigns $(n-1)(p-1)(n)/2$ data elements to zero. We call these assigned elements the $R5X0$ presets. Geometrically, the presets form a triangle of width $n-1$ and height $(n-1)(p-1)$ in the lower right corner of the data matrix. We choose these elements as presets because they allow us to omit the modulus from (1): whenever $i - j\cdot k < 0$, the index wraps around and falls in the preset region where $D^j_{i-j\cdot k} = 0$. The layout for $R5X0(5, 6, 2)$ is illustrated in Figure 1.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Parity 0	Parity 1
$\underline{\mathbf{D}}_0^0$	$\mathbf{D}_0^1$	$\mathbf{D}_0^2$	$\mathbf{D}_0^3$	$\mathbf{D}_0^4$	$\mathbf{P}_0^0$	$\underline{P}_0^1$
D_1^0	D_1^1	D_1^2	D_1^3	D_1^4	P_1^0	P_1^1
D_2^0	D_2^1	D_2^2	D_2^3	$\underline{D}_2^4 \leftarrow 0$	P_2^0	P_2^1
D_3^0	D_3^1	D_3^2	$\underline{D}_3^3 \leftarrow 0$	$D_3^4 \leftarrow 0$	P_3^0	P_3^1
D_4^0	D_4^1	$\underline{D}_4^2 \leftarrow 0$	$D_4^3 \leftarrow 0$	$D_4^4 \leftarrow 0$	P_4^0	P_4^1
D_5^0	$\underline{D}_5^1 \leftarrow 0$	$D_5^2 \leftarrow 0$	$D_5^3 \leftarrow 0$	$D_5^4 \leftarrow 0$	P_5^0	P_5^1

Figure 1: Data and parity layout for $R5X0(n = 5, r = 6, p = 2)$. The inputs to parity element P_0^0 are typeset in bold font. The inputs to parity element P_0^1 are underlined. The presets form a triangle with width $n-1$ and height $(p-1)(n-1)$ and thus there are $(p-1)(n-1)(n)/2$ presets.

2.2. Proof of Distance

Lemma 1. *An $R5X0(n, r, p)$ code can recover from the erasure of any x, $0 \leq x \leq p$, data disks using any x parity disks and the remaining data disks.*

Proof: We prove this Lemma inductively. Consider the topmost unknown elements from each of the x missing data disks. Initially, these elements are simply the topmost row elements of the missing disks. However, in the general case, the topmost unknown elements will form a downward facing convex pattern as shown in Figure 2. We consider the convex hull formed by these unknown elements. Because there are at most x topmost unknown elements, the top surface of this convex hull is defined by at most $x-1$ lines of distinct slope. Because we are given x redundancy (parity) disks, each with a RAID5 XOR parity along a different integer slope, it is clear that at least one parity disk has a slope distinct from the slopes that compose the convex surface. We will use such a parity disk to solve for one of the topmost unknown data elements, thereby reducing the number of unknown elements by one. Repeated application this argument will necessarily solve for all the erased data elements.

To see how we solve for one of the topmost unknown elements, denote $\texttt{Lost}[l]$ as the disk index of the lth lost disk for $0 \leq l < x$ and where $\texttt{Lost}[l] < \texttt{Lost}[l']$ whenever $l < l'$. Likewise, denote $\texttt{Par}[l]$ as the slope of the lth available parity disk for $0 \leq l < x$ and where $\texttt{Par}[l] > \texttt{Par}[l']$ whenever $l < l'$. Furthermore, denote s_l as the slope of the convex hull between disk $\texttt{Lost}[l-1]$ and $\texttt{Lost}[l]$ for $0 < l < x$. We define $s_0 = \infty$ and $s_x = -\infty$. For future reference, observe that $s_0 > \texttt{Par}[0]$ and $s_x < \texttt{Par}[x-1]$.

We claim that if $s_l > \texttt{Par}[l] > s_{l+1}$ we can solve for $\texttt{Lost}[l]$'s topmost unknown element. Define $k = \texttt{Par}[l]$ and $j = \texttt{Lost}[l]$. Denote i as the row of disk j's topmost unknown element. Consider parity element P_{i+jk}^k. The only unknown input to this parity element is D_i^j. The rest of the inputs are either above the convex hull (and therefore not an unknown element) or wrap-around and are in the preset region (and therefore a preset with value 0). A simple parity computation can be used to compute the value of D_i^j.

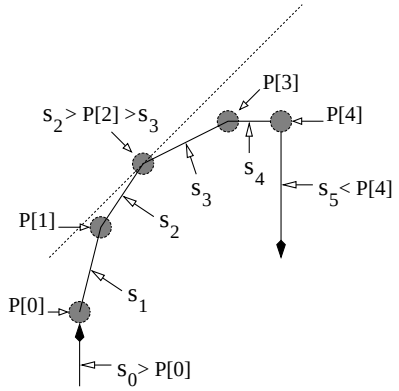

Figure 2: The top surface of the convex hull defined by the topmost unknown data symbols. Slopes $s_1 \ldots s_4$ define the convex surface. Our parities have slopes $\texttt{Par}[0] \ldots \texttt{Par}[4]$. We can solve for the top element of the third erased disk using an element from the third parity disk, shown as a dotted line tangent to the convex hull.

Such an l necessarily exists because $\texttt{Par}[l]$ is strictly increasing, s_i is decreasing, and $s_0 > \texttt{Par}[0]$ but $s_x < \texttt{Par}[x-1]$. At the first l where $\texttt{Par}[l] > s_{l+1}$ it is clear that $s_l > \texttt{Par}[l] > s_{l+1}$. □

Theorem 1. *$R5X0(n, r, p)$ has column distance $p + 1$.*

Proof: Without loss of generality, assume that we lose $0 \leq x \leq p$ data disks and $p-x$ parity disks. In this case, we have x erased data disks and x remaining parity disks. Lemma 1 proves that the $R5X0$ scheme allows us to rebuild x erased data disks from any x parity disks. □

2.3. The Decoding Algorithm

We give a straight-forward algorithm `RECONSTRUCT` to reconstruct x erased data disks $\texttt{Lost}[0..x-1]$ with x available parity disks $\texttt{Par}[0..x-1]$. As in the proof of Lemma 1, we assume that `Lost` is given in increasing disk order and `Par` is given in decreasing slope order. We use a third array $\texttt{Top}[0..x-1]$ where $\texttt{Top}[l]$ contains the row index of the top-most unknown symbol from disk $\texttt{Lost}[l]$. All elements of `Top` are initialized to 0. Our algorithm follows.

```
RECONSTRUCT(Top, Lost, Par, x, r)
  while (⋁_{l=0}^{x-1}(Top[l] ≠ r)
    for l = 0..x − 1
      while (SOLVE(Top[l], Lost[l], Par[l], l))
        Top[l] + +

SOLVE(i, j, k, l)
   i' ← i + j · k
   if (i == r)
     return False // column complete
   if (⋀_{l'=l−1,l+1} Top[l'] > i' − Lost[l'] · k)
     return False // missing data
   if (i ≥ r − j · (p − 1))
     D_i^j ← 0 // preset
   else
     D_i^j ← P_{i'}^k ⊕ ⨁_{j'=0,j'≠j, i'−j'·k≥0}^{n−1} D_{i'−j'·k}^{j'}
   return True
```

The RECONSTRUCT algorithm first checks if we have finished solving for all lost data elements and if not, it solves for individual lost disks. SOLVE first checks if the reonstruction is complete, and then determines whether or not we can solve for data element D_i^j. We are able to solve for D_i^j provided that all other inputs to parity $P_{i+j\cdot k}^k$ are known. Because the topmost unknown elements are on a convex surface we need only check the inputs from the adjacent erased disks.

It is clear from these definitions and the proof of Lemma 1 that RECONSTRUCT makes progress every iteration and therefore successfully solves for all erased data symbols. Each computed symbol requires at most $n-1$ XORs. There are at most $x \cdot r$ erased data symbols. The number of XORs required to complete RECONSTRUCT is thus less than $x \cdot r \cdot (n-1)$. The running time of RECONSTRUCT is $O(x^2 \cdot r \cdot n)$.

3. Comparisons

In Section 2.2 we proved that the $R5X0(n, r, p)$ code has a column distance of $p + 1$. That is, the $R5X0(n, r, p)$ layout can tolerate the erasure of any p disks. In this section we explore two other metrics — storage efficiency and update complexity — and see how they vary as a function of the code parameters n, r, and p. We then compare $R5X0$ to other codes with respect to these metrics.

3.1. Efficiency

The storage efficiency E represents the fraction of the storage space that can be used for independent data. Let D denote the number of independent data symbols and T denote

the total number of symbol blocks used by the layout. The storage efficiency of a layout is defined as:

$$E = \frac{D}{T} \ . \tag{3}$$

The optimal storage efficiency of a distance $p+1$ code with n data disks is given by an MDS code:

$$E_{MDS} = \frac{n}{n+p} \ . \tag{4}$$

The $R5X0$ code is a near-MDS code in the sense that its storage efficiency can be made arbitrarily close to E_{MDS}. The number of independent data symbols in the $R5X0(n,r,p)$ layout is given by the number of data symbols nr in the data matrix minus the number of presets $(p-1)(n-1)(n)/2$. The total number of blocks used by the $R5X0(n,r,p)$ layout is just the size of the whole matrix $(n+p)r$. The storage efficiency of $R5X0(n,r,p)$ is thus:

$$E_{R5X0} = \frac{nr-(p-1)(n-1)(n)/2}{(n+p)r} \ . \tag{5}$$

If we write r as $kn(p-1)$ for rational $k \geq 1$ and assume that n is large, after some minor algebraic manipulation we get:

$$E_{R5X0} \approx \left(\frac{n}{n+p}\right) \cdot \left(1-\frac{1}{2k}\right) \ . \tag{6}$$

As k increases, the storage efficiency of $R5X0$ approaches E_{MDS}. In actuality, it is easy to obtain much higher storage efficiencies for the $R5X0$ code. In Section 4 we discusses how to improve upon the storage efficiency given by (6).

3.2. Update Complexity

The update complexity of a code is the average number of parity symbols affected by a change in a data symbol [XB99]. In the $R5X0(n,r,p)$ code, every data symbol is an input to exactly p parity symbols, one from each parity disk. It is easy to see that the best possible update complexity for a distance $p+1$ code is in fact p. Therefore, for codes with column distance $p+1$,

$$UC_{OPT} = UC_{R5X0} = p \ . \tag{7}$$

Update complexity is particularly important in the storage systems model because symbol reads and symbol writes (IOs) dominate over computation time. For most codes, IOs is directly related to update complexity:

$$IOs = 2(UC+1) \ . \tag{8}$$

This IO cost corresponds to the cost of reading the original data symbol and all its affected parities and then writing the new data symbol and modified parity symbols (i.e., read-modify-write parity increment). As we shall see in the next section, (8) does not hold for some types of inefficient codes.

3.3. R5X0 and Other Codes

In Tables 1-3 we compare a number of codes with the $R5X0$ code of corresponding distances. We do not consider Reed-Solomon codes because they are not XOR-based codes and best suited for special purpose hardware. A number of codes have IO cost better than what is allowed by (8) (as indicated by $*$ in the table). All of these codes achieve better IO costs because of their inefficiency allows them to update parity by recompute rather than read-modify-write as above. The fact that these codes have no more data disks than parity disks allows them to modify a data symbol without reading the symbol's old value and its parities' old values.

In Table 1 we compare various distance 3 codes. The $R51^{-}(a)$ code has a data disks, a mirror disks, and one RAID5 parity disk. The $R6(a \times b)$ code has ab data disks arranged logically in a $a \times b$ matrix and $a + b$ RAID5 parity disks, one for each matrix row and column. $XC(p)$ [XB99] has p total disks and p rows per disk, where the last two symbols in each disk are parity symbols. $ZZS(p)$ [ZZS83] stores a row of parity on the data disks; data chunks corresponding to a parity unit do not form any obvious pattern. $EO(p)$ [BBBM95], $BR(p, n)$ [BR99], and $RDP(p)$ [CEG^{+}04] have at most p data disks ($p - 1$ for RDP) and two parity disks with $p - 1$ symbols per disk.

In Table 2 we compare various distance 4 codes. The $R51(a)$ code has a data disks, a disk with RAID5 parity, and $a + 1$ mirror disks. The $R6^{+}(a \times b)$ code has ab data disks arranged logically in a $a \times b$ matrix and $a + b + 1$ RAID5 parity disks, one for each matrix row and column and one for the entire matrix. $EO^{+}(p, 3)$ [BBV96] has p data disks and three parity disks with $p - 1$ symbols per disk.

In Table 3 we compare various higher distance codes. $EO^{+}(p, d-1)$ [BBV96] has p data disks and $d - 1$ parity disks with $p - 1$ symbols per disk. For $d \geq 5$, some of these codes only work for certain primes, as indicated in the table.

Table 1: Distance three codes.

$d = 3$	Avg. IOs	Efficiency	Array Constraints
$R51^{-}(2)$	4^{*}	40%	$r \times 5$ for any r
$R51^{-}(a)$	5^{*}	$a/(2a + 1)$	$r \times (2a + 1)$ for any r, a
$R6(2 \times 2)$	5^{*}	50%	$r \times 8$ for any r
$R6(a \times b)$	6	$ab/(ab + a + b)$	$r \times (ab + a + b)$ for any r, a, b
$XC(p)$	6	$(p - 2)/p$	$p \times p$ for prime p
$ZZS(p)$	6	$(p - 2)/p$	$(p - 1)/2 \times p$ for prime p
$\mathbf{R5X0(n, r, 2)}$	6	$n/(n + 2) - \epsilon_r$	$r \times (n + 2)$ for any $n, r \geq n$
$EO(p)$	> 6	$p/(p + 2)$	$(p - 1) \times (p + 2)$ for prime p
$BR(p, n)$	> 6	$n/(n + 2)$	see [BR99]
$RDP(p)$	> 6	$(p - 1)/(p + 1)$	$(p - 1) \times (p + 1)$ for prime p

Table 2: Distance four codes.

$d = 4$	Avg. IOs	Efficiency	Array Constraints
$R51(2)$	5^*	$\approx 33\%$	$r \times 6$ for any r
$R51(a)$	6^*	$a/(2a+2)$	$r \times (2a+1)$ for any a, r
$R6^+(2 \times 2)$	7^*	$\approx 44\%$	$r \times 9$ for any r
$R6^+(a \times b)$	8	$ab/(a+1)(b+1)$	$r \times (a+1)(b+1)$ for any a, b, r
$\mathbf{R5X0(n, r, 3)}$	8	$n/(n+3) - \epsilon_r$	$r \times (n+3)$ for any $n, r \geq 2n$
$EO^+(p, 3)$	> 8	$n/(n+3)$	$(p-1) \times (p+3)$ for some primes p

Table 3: Distance $d \geq 5$ codes.

$d \geq 5$	Avg. IOs	Efficiency	Array Constraints
$\mathbf{R5X0(n, r, d-1)}$	$2d$	$n/(n+d-1) - \epsilon_r$	$r \times (n+d-1)$ for any $n, r \geq (d-2)n$
$EO^+(p, d-1)$	$> 2d$	$n/(n+d-1)$	$(p-1) \times (p+d-1)$ for some primes p

Conventional high-distance RAID codes like $R51$ and $R6$ are simple and have very good IO, but are impractical when storage efficiency is important. The Blaum-Roth and Even-Odd codes achieve optimal storage efficiency but do so at the expense of update complexity. In particular, the modification of certain data elements in the EvenOdd code requires an expensive update to almost all of the parity values. ZZS and the X-Code achieve both optimal storage efficiency and update complexity, but they do not generalize to higher distances. $R5X0$ codes achieve near-optimal storage efficiency and optimal update complexity and generalize to arbitrary distances with relatively few array constraints.

4. Optimizations

There are two ways we can improve the storage efficiency of the $R5X0$ layout described in Section 2.1. First, we can reduce the number of presets. Second, we can store non-zero symbols from another $R5X0$ instance in the disk blocks designated for the presets of the first code instance. However, when we try to store data from one code instance in the disk blocks designated for presets, not all preset locations are utilized, and thus results in "wasted" disk blocks (see Figure 5).

Let Z be the number of preset blocks and W be the number of wasted disk blocks in a specific layout. Also, let $N = nr$ be the number of data symbols and let $T = (n+p)r$ be the total number of symbols. The storage efficiency of the $R5X0$ layout is:

$$E_{R5X0} = \frac{N-Z}{T-(Z-W)} \tag{9}$$

$$= \left(\frac{N}{T}\right)\left(\frac{1-\frac{Z}{N}}{1-\frac{Z-W}{T}}\right) \tag{10}$$

$$\approx E_{MDS}\left(1-\frac{Z}{N}\right)\left(1+\frac{Z-W}{T}\right) \quad . \tag{11}$$

Recall from (4) that $N/T = E_{MDS}$. The approximation in (11) relies on the fact that $Z - W$ is much smaller than T.

The storage efficiency given in (6) is for a layout in which $Z = (p-1)(n-1)(n)/2$ presets and $W = Z$. The enhanced $R5X0$ layout shown in Figure 3 requires only $(p-1)\lfloor n/2 \rfloor \lceil n/2 \rceil$ presets. The improved storage efficiency is given by (12) (recall that $r = kn(p-1)$). The proofs of Lemma 1 and Theorem 1 are unaffected by this change because it is still the case that no parity element has independent inputs that wrap around from row 0 to row $r-1$.

$$E_{R5X0} \approx \left(\frac{n}{n+p}\right) \cdot \left(1-\frac{1}{4k}\right) \quad . \tag{12}$$

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Parity 0	Parity 1
$\mathbf{D}_0^0 \leftarrow 0$	$\mathbf{D}_0^1 \leftarrow 0$	$\underline{\mathbf{D}}_0^2$	$\mathbf{D}_0^3$	$\mathbf{D}_0^4$	$\mathbf{P}_0^0$	P_0^1
$D_1^0 \leftarrow 0$	$\underline{D}_1^1$	D_1^2	D_1^3	D_1^4	P_1^0	P_1^1
$\underline{D}_2^0$	D_2^1	D_2^2	D_2^3	D_2^4	P_2^0	$\underline{P}_2^1$
D_3^0	D_3^1	D_3^2	D_3^3	D_3^4	P_3^0	P_3^1
D_4^0	D_4^1	D_4^2	D_4^3	$\underline{D}_4^4 \leftarrow 0$	P_4^0	P_4^1
D_5^0	D_5^1	D_5^2	$\underline{D}_5^3 \leftarrow 0$	$D_5^4 \leftarrow 0$	P_5^0	P_5^1

Figure 3: Data and parity layout for $R5X0(5, 6, 2)$. The presets for this layout are those elements above the inputs to either P_0^0 or $P_{(p-1)\lfloor n/2 \rfloor}^{p-1}$ but not above the inputs to both P_0^0 and $P_{(p-1)\lfloor n/2 \rfloor}^{p-1}$. The presets form two triangles with widths $\lfloor (n-1)/2 \rfloor$ and $\lceil (n-1)/2 \rceil$ and heights $(p-1)\lfloor (n-1)/2 \rfloor$ and $(p-1)\lceil (n-1)/2 \rceil$ and therefore there are $(p-1)\lfloor (n/2) \rfloor \lceil (n/2) \rceil$ presets (proof: we can combine these triangles together into rectangle of dimension $(p-1)n/2 \times n/2$ or $(p-1)(n-1)/2 \times (n+1)/2$).

In both (6) and (12) we have assumed that $W = Z$: all preset symbol blocks are wasted space. In practice, however, there is no reason to waste disk blocks because they are allocated to presets. Instead, we can store unrelated data in these blocks, like data from another $R5X0$ code instance. In the remainder of this section we describe methods to reduce W without the introduction of an unwieldy mapping from algorithmic space to physical space.

For the standard $R5X0$ layout we can achieve $W = (p+1)(p)(p-1)/2$ by using a mirrored layout and nesting it into the original layout as shown in Figure 4. For convenience, we have placed the parity disks to the left of the data disks. We access elements from the mirrored layout by mapping D_i^j to $D_{n+p-i-1}^{r-j-1}$, a very simple transformation. The storage efficiency of the resulting layout can be derived from (11) and is given in (13).

$$E_{R5X0} \approx \left(\frac{n}{n+p}\right) \cdot \left(1 - \frac{p}{2kn} - \frac{1}{4k^2}\right) \quad . \tag{13}$$

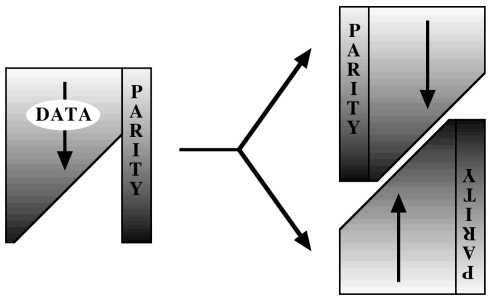

Figure 4: Two copies of $R5X0(5, 6, 2)$ on the same seven disks, arranged so that each copy uses some of the others preset disk blocks to store data and parity symbols. The empty disk blocks form two triangles of width $(p-1)$ and height $(p-1)^2$. $W = (p+1)(p)(p-1)/2$ (proof: combine the two triangles into a $(p-1)p \times (p+1)$ rectangle.)

We can do a similar thing for the enhanced $R5X0$ layout Figure 3 achieving storage efficiency:

$$E_{R5X0} \approx \left(\frac{n}{n+p}\right) \cdot \left(1 - \frac{p}{4kn} - \frac{1}{16k^2}\right) \quad , \tag{14}$$

when $n+p$ is even. The storage efficiency is slightly worse for odd $n+p$ because the layouts do not nest tightly. Our transformation is as follows:

$$D_i^j = \begin{cases} D_i^j & \text{if } j < \lfloor n/2 \rfloor \\ D_{r-i-1}^{j+p} & \text{if } \lfloor n/2 \rfloor \le j < n \\ D_i^{j-\lceil n/2 \rceil} & \text{if } j \ge n \end{cases}$$

In words, we move the p parity disks to the middle of the matrix and vertically invert the data in the right $\lceil n/2 \rceil$ data disks. After performing this transformation, we vertically invert a copy of this layout and shift it by $\lfloor (n+p)/2 \rfloor$. The two copies are then nested on top of each other. The complete transformation is illustrated in Figure 5. If the total number of disks $n+p$ is even, we achieve a nice fit with only $(p-1)\lfloor p/2 \rfloor \lceil p/2 \rceil$ wasted disk blocks per layout. Our fit is off by a disk if $n+p$ is odd, and there are $(p-1)(\lfloor (n+p)/2 \rfloor + \langle p \rangle_2)$ extra wasted symbols split between the two layouts.

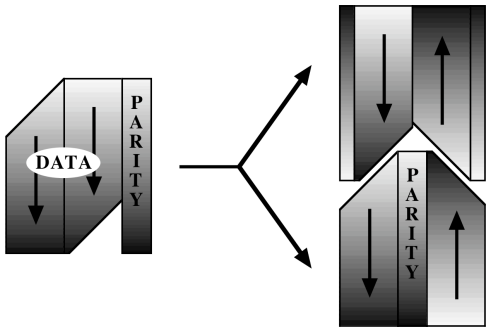

Figure 5: The storage efficiency saving transformation for the enhanced layout.

5. Conclusions

We have presented the $R5X0$ code, a high distance generalization of RAID5 with optimal update complexity and near-optimal storage efficiency. The key insight behind $R5X0$ is the use of *presets*, data cells with known values that initialize the reconstruction process. We have shown how to pack multiple copies of the R5X0 layout onto the same disks to minimize the effect of presets on storage efficiency without destroying the code's clean geometric construction. $R5X0$ has efficient XOR-based encoding, recovery, and updating algorithms for arbitrarily large distances, making it an ideal candidate when storage-efficient reliable codes are required.

Acknowledgements

We would like to thank K.K. Rao, Veera Deenadayalan, Bruce Cassidy, Rich Freitas and Jai Menon for many valuable comments. Tapas Kanungo would like to thank Mario Blaum for providing pointers to array code literature. This work was done while Jeff Hartline was visiting IBM Almaden Research Center. Hartline is supported in part by NSF grant CCR-0105586 and by ONR Grant N00014-01-1-0968.

References

[BBBM95] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: an efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computers*, 44:192–202, 1995.

[BBV96] M. Blaum, J. Bruck, and A. Vardy. MDS array codes with independent parity symbols. *IEEE Transactions on Information Theory*, 42:529–542, 1996.

[BFv98] M. Blaum, P. G. Farrell, and H. C. A. van Tilborg. Array codes. In V. S. Pless and W. C. Huffman, editors, *Handbook of Coding Theory (Vol. 2)*, pages 1855–1909. North Holland, 1998.

[BR99] M. Blaum and R. M. Roth. On lowest density MDS codes. *IEEE Transactions on Information Theory*, 45:46–59, 1999.

[CEG+04] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of USENIX Conference on File and Storage Technology*, pages 1–14, 2004.

[FHB89] T. Fuja, C. Heegard, and M. Blaum. Cross parity check convolution codes. *IEEE Trans. on Information Theory*, 35(6):1264–1276, 1989.

[Gal62] R. G. Gallager. *Low-Density Parity-Check Codes*. MIT Press, Cambridge, MA, 1962.

[HP03] J. H. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, 2003.

[LMSS01] M. G. Luby, M. Mitzenmacher, A. Shokrollahi, and D. A. Spielman. Efficient erasure correcting codes. *IEEE Transactions on Information Theory*, 47:569–584, 2001.

[Lub02] M. Luby. LT codes. In *Proceedings of the 43rd Annual IEEE Symposium on the Foundations of Computer Science*, pages 271–280, 2002.

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

[Mas97] P. Massiglia. *The RAID Book*. The RAID Advisory Board, Inc., St. Peter, MN, 1997.

[Pla97] J. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software: Practice and Experience*, 27:995–1012, 1997.

[RS60] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8:300–304, 1960.

[Sho03] A. Shokrollahi. Raptor codes, 2003.

[Wil01] A. Wilner. Multiple drive failure tolerant raid system, December 2001. U. S. Patent 6,327,672 B1.

[XB99] L. Xu and J. Bruck. X-code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory*, pages 272–276, 1999.

[ZZS83] G. V. Zaitsev, V. A. Zinovev, and N. V. Semakov. Minimum-check-density codes for correcting bytes of errors. *Problems in Information Transmission*, 19:29–37, 1983.