RAID+: Deterministic and Balanced Data Distribution for Large Disk Enclosures

Guangyan Zhang and Zican Huang, *Tsinghua University*; Xiaosong Ma, *Qatar Computing Research Institute, HBKU*; Songlin Yang, Zhufan Wang, and Weimin Zheng, *Tsinghua University*

https://www.usenix.org/conference/fast18/presentation/zhang

This paper is included in the Proceedings of the 16th USENIX Conference on File and Storage Technologies.

February 12–15, 2018 • Oakland, CA, USA

ISBN 978-1-931971-42-3

RAID+: Deterministic and Balanced Data Distribution for Large Disk Enclosures

Guangyan Zhang*, Zican Huang*, Xiaosong Ma†, Songlin Yang*, Zhufan Wang*, Weimin Zheng*
*Tsinghua University, †Qatar Computing Research Institute, HBKU

Abstract

Existing RAID solutions partition large disk enclosures so that each RAID group uses its own disks exclusively. This achieves good performance isolation across underlying disk groups, at the cost of disk under-utilization and slow RAID reconstruction from disk failures.

We propose RAID+, a new RAID construction mechanism that spreads *both normal I/O and reconstruction workloads* to a larger disk pool in a balanced manner. Unlike systems conducting randomized placement, RAID+ employs deterministic addressing enabled by the mathematical properties of mutually orthogonal Latin squares, based on which it constructs 3-D data templates mapping a logical data volume to uniformly distributed disk blocks across all disks. While the total read/write volume remains unchanged, with or without disk failures, many more disk drives participate in data service and disk reconstruction. Our evaluation with a 60-drive disk enclosure using both synthetic and real-world workloads shows that RAID+ significantly speeds up data recovery while delivering better normal I/O performance and higher multi-tenant system throughput.

1 Introduction

For the past 30 years, Redundant Array of Inexpensive Disks (RAID) [40] has been used pervasively in servers and shared computing platforms. With parity-based RAID levels (*e.g.*, RAID-5 and RAID-6), users obtain high performance via parallel accesses and reliability via data redundancy.

With continued advance in disk capacity and slow improvement in speed, however, RAID rebuild time keeps increasing [13, 54]. For example, a recent NetApp document specifies that a 2TB SATA 7200-RPM disk takes 12.8 hours to rebuild on an idle system [12]. When performed online on a heavily loaded system, rebuild can take dramatically longer. Such slow rebuild brings two consequences. First, it raises the risk of a second failure and consequently data loss. Second, prolonged recovery subjects foreground applications to long periods of I/O performance degradation. Note that high-performance solid-state drives (SSDs) actually exacerbate this problem, as their growing deployment promotes storage system construction using more high-density, low-performance hard disks [43].

One inherent reason for such slow recovery is that, with conventional RAID, each disk drive involved in RAID reconstruction is read or written entirely. Despite the growing width of RAID arrays (with each array typically containing several to around a dozen disks), the recovery time is determined by reading/writing an entire disk. No matter how many disk arrays coexist in a shared/virtual storage system, resources are isolated between underlying RAID arrays. Idle or lightly-loaded disks cannot offer help to peers in other RAID arrays, who might be overwhelmed by high access traffic, RAID recovery, or, in the worst case, both.

Many approaches to enhancing the reconstruction performance have been proposed [3, 18, 21, 28, 42, 52, 55, 56], which fall into three categories: 1) *designing better data layout* in a disk group [18, 48, 52, 56], 2) *optimizing the reconstruction workflow* [19, 31, 45, 54], and 3) *improving the rate control of RAID reconstruction* [32, 44, 47]. Most methods focused on a single RAID group, and to our best knowledge, no solution yet has eliminated load imbalance both in normal operations and during RAID reconstruction. While random data placement can utilize larger groups of disks [12, 16, 41, 51], it requires extra book keeping and lookup, and does not deliver load balance within shorter ranges of blocks (crucial for sequential access and RAID rebuild performance, as shown in our experiments).

This paper presents RAID+, a new RAID construction mechanism that spreads *both normal and reconstruction I/O* to effectively utilize emerging commodity enclosures (such as the NetAPP DE6600 and EMC VNX-series) with dozens of or even 100+ disks. Unlike systems conducting random placement, RAID+ employs a deterministic addressing algorithm that leverages the mathematical properties of *mutually orthogonal Latin squares*. Such properties allow RAID+ to construct 3-D data templates, each employing a user-specified RAID level and stripe width, that map logical data extents to uniformly distributed disk blocks within a larger disk pool.

While the total read/write volume remains unchanged, with or without disk failures, RAID+ enlists many more disk drives in data service and disk reconstruction. This allows it to provide more consistent performance, much faster recovery, and better protection from permanent data loss. In addition, in multi-tenant settings it automatically lends elastic resources to individual workloads' varying intensity, via a flexible and scalable integration of multiple disk groups. We find that this often leads to higher overall resource utilization, though like most schemes for workload consolidation, in the worst case it may incur I/O interference. Such elasticity, combined with the capability of constructing multiple logical volumes adopting different RAID levels and stripe widths within the same physical pool, makes RAID+ especially attractive to cloud and shared datacenter environments employing large disk enclosures/trays.

We implemented a RAID+ prototype by modifying the Linux `MD` (Multiple Devices) driver, and evaluated it using a 60-drive disk enclosure. Results show that RAID+ in most cases outperforms both RAID-50 and randomized RAID-5 placement schemes, while offering faster reconstruction (2.1-7.5× over RAID-50, 1.0-2.5× over hash-based random placement). Like randomized placement, it significantly improves overall throughput in multi-tenant environments (average 2.1× over RAID-5). But unlike randomized placement, RAID+'s deterministic addressing allows simple implementation and delivers better sequential performance (for application and rebuild I/O) by guaranteeing uniform data distribution within smaller extents and retaining spatial locality.

2 RAID+ Overview

2.1 Latin Square Based Data Organization

With conventional k-disk RAID arrays, each data stripe is exactly k-block wide (including both data and parity), squarely striking through all disks. The RAID type (level) and stripe width both remain fixed throughout a given disk array. RAID+, instead, uses *Latin-square-based templates* to allocate space from a larger n-disk array. A template constructs $n\times(n-1)$ k-block stripes, each mapped to a k-subset of the n disks. Different k values and RAID types can be adopted by different templates sharing the same n disks. Like conventional RAID, RAID+ arrays can be hardware- or software-based, offered as RAID+ enclosures with special RAID adapters or formed by software on top of connected disks.

Figure 1(a) portrays conventional RAIDs, where disks are physically partitioned into two RAID groups, with potentially different RAID settings. Each disk belongs to one fixed RAID array, except the shared hot spares. One can integrate multiple underlying RAID groups (likely homogeneous in this case) into a logical volume, by *concatenating* them, or *striping* data across them. The widely adopted RAID-50, for example, belongs to the latter case. Alternatively, one can build a logical volume on each underlying RAID group, separately serving different workloads sharing the disk pool. These two options are used in our experiments for single- and multi-workload evaluation, respectively.

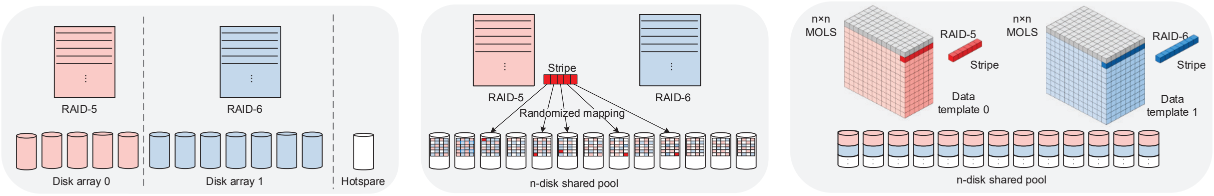
With conventional RAID organization, when a failure happens, the recovery process only involves disks within the same RAID group, reading from the $k-1$ surviving drives and reconstructing lost data on a spare drive in its entirety. As a result, the rebuild speed is capped by the slower between read and write speeds of a single disk.

Figure 1(b) shows an alternative approach, where RAID volumes are built by distributing blocks in each k-block RAID stripe to randomly selected k disks. This retains the fault tolerance of RAID yet spreads each volume to all n disks within the pool.

Figure 1(c) shows our proposed RAID+, also a flat organization of the same n-disk pool, where two data templates are used to carve space uniformly from all disks. Each template is designed by "stacking" a sequence of k $n\times n$ mutually orthogonal Latin squares, whose definition is given in the next section. Each Latin square cell stores a disk ID within $[0,n-1]$. Cells at the same location through the k layers then form a k-width data stripe (highlighted). Given such a set of k Latin-squares, one can easily compute the locations of any stripe' blocks, on a k-subset of the n drives. The mathematical properties of mutually orthogonal Latin squares guarantee the uniform data distribution on all working drives, either for normal or single-failure recovery accesses. Since data distribution by each template is always uniform, users can host different RAID organizations within the same n-drive disk pool, such as RAID-5 using the red and RAID-6 using the blue template.

When a disk failure occurs, both random placement and RAID+ allow all surviving disks to equally participate in reconstruction, cutting theoretical RAID rebuild time to $k/(n-1)$ of that of conventional RAID. Also, hot spares are optional with these organizations. However, as we shall see later in the paper, RAID+' deterministic and uniform data placement enables it to achieve perfect load balance within smaller address extents and retain spatial locality, both significant advantages (crucial to sequential access and RAID rebuild) over random schemes.

In addition, a RAID+ pool can perform in an *interim mode* with multiple disk failures, by continuously main-



(a) Conventional RAID (b) Random data placement (c) RAID+

Figure 1: Different ways of utilizing a disk pool much larger than typical RAID array sizes

	Application I/O		Rebuild I/O		
	Isol.	*Thrp*	$T_{rebuild}$	*Interf.*	*MTTDL*
RAID-5C	High	$T \cdot k$	C/B	Part-High	t
RAID-50	Low	$T \cdot (n-s)$	C/B	Part-High	t
RAID+	Low	$T \cdot n$	$C \cdot k/(B \cdot (n-1))$	Univ-Low	$> t \cdot (k-1)/k$

Table 1: Comparison of RAID-5 organizations

taining its uniform or near-uniform data distribution. In fact, RAID+ reserves space for data recovery and always performs a fast all-to-all reconstruction. When hot spares are available or failed disks are repaired, the recovered data will be replicated to replacement disk(s) in background, hiding the slow single-disk writing latency.

2.2 Comparison of RAID Usage Modes

Table 1 gives several major metrics, comparing RAID+ with common existing solutions utilizing larger disk pools. s denotes the number of hot spare disks, while C and B denote single-disk capacity and bandwidth, respectively. Without loss of generality, we use RAID-5 as the elementary RAID level. Here *RAID-5C* and *RAID-50* refer to the aforementioned "concatenated" and "striped" volume construction modes. We omit random placement schemes as they are similar to RAID+ in these aspects (but suffer from inferior load balance and locality).

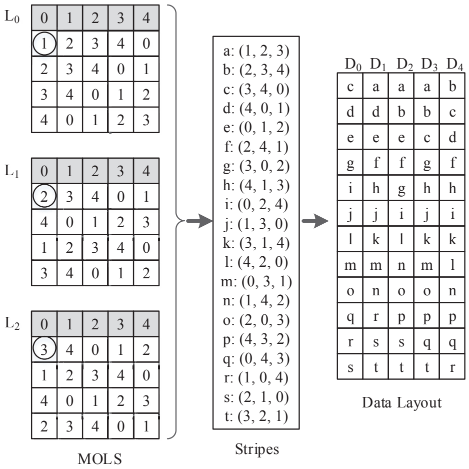
For application I/O, RAID-5C has good inter-application isolation, as different workloads are more likely to involve separate underlying RAID-5 arrays, while both RAID-50 and RAID+ would be subject to performance interference from concurrent workloads. The tradeoff is aggregate performance per volume: files on RAID-5C can only utilize 1-2 physical k-disk RAID array at a time, while RAID-50 and RAID+ could enlist most or all disks. This applies to both sequential accesses (in bandwidth) and random ones (in IOPS).

For RAID rebuild I/O, both RAID-5C and RAID-50 limit reconstruction to the physical array with the disk failure. Their recovery time ($T_{rebuild}$) is equivalent to a single-disk full scan, assuming perfect read-write overlap. In contrast, RAID+ enlists all $n-1$ surviving disks in the read-write of the k-disk capacity ($C \cdot k$), making recovery itself much faster. Regarding application-perceived interference, with RAID-5C the RAID reconstruction process is only visible to accesses to the same physical array. RAID-50 gets similar partial exposure with random accesses, but could be universally affected with larger, sequential reads/writes, as shown in our evaluation (Table 3). With RAID+'s all-to-all data recovery, reconstruction traffic can be perceived by most user requests, but the interference is lighter and lasts shorter.

Finally, the $MTTDL$ column describes *mean time to data loss*, considering the probability of non-recoverable failures (such as second disk failure before reconstruction completes with RAID-5). We find RAID-5C and RAID-50 with the same $MTTDL$ and give a conservative lower bound for RAID+ relative to it. The bound is fairly close to 1 and configurable by k. With RAID-6, however, we found that RAID+ actually enjoys a significant improvement in $MTTDL$ over conventional systems [49], by prioritizing the reconstruction of significantly fewer yet more vulnerable stripes.

3 Latin Squares for Data Distribution

We first introduce basic concepts and theorems of mutually orthogonal Latin squares (MOLS), followed by an example illustrating its use in constructing RAID+.



L_0				
0	1	2	3	4
1	2	3	4	0
2	3	4	0	1
3	4	0	1	2
4	0	1	2	3

L_1				
0	1	2	3	4
2	3	4	0	1
4	0	1	2	3
1	2	3	4	0
3	4	0	1	2

L_2				
0	1	2	3	4
3	4	0	1	2
1	2	3	4	0
4	0	1	2	3
2	3	4	0	1

a: (1, 2, 3)
b: (2, 3, 4)
c: (3, 4, 0)
d: (4, 0, 1)
e: (0, 1, 2)
f: (2, 4, 1)
g: (3, 0, 2)
h: (4, 1, 3)
i: (0, 2, 4)
j: (1, 3, 0)
k: (3, 1, 4)
l: (4, 2, 0)
m: (0, 3, 1)
n: (1, 4, 2)
o: (2, 0, 3)
p: (4, 3, 2)
q: (0, 4, 3)
r: (1, 0, 4)
s: (2, 1, 0)
t: (3, 2, 1)

D_0	D_1	D_2	D_3	D_4
c	a	a	a	b
d	d	b	b	c
e	e	e	c	d
g	f	f	g	f
i	h	g	h	h
j	j	i	j	i
l	k	l	k	k
m	m	n	m	l
o	n	o	o	n
q	r	p	p	p
r	s	s	q	q
s	t	t	t	r

Figure 2: RAID+ layout ($n = 5$, $k = 3$)

Definition 1. *A Latin square of order n is an $n \times n$ array filled with n different items, each occurring exactly once in each row and column.*

Definition 2. *Let L_1 and L_2 be two n-order Latin squares. L_1 and L_2 are mutually orthogonal if, when su-*

perimposed, each of the n^2 ordered pairs occur exactly once across the overlapping cells of the two squares.

Definition 3. *Given a set of Latin squares, if all its member pairs are mutually orthogonal, we call it a set of mutually orthogonal Latin squares (MOLS).*

For example, the left side in Figure 2 gives three sample 5-order MOLS. Here the item set $I = \{0,1,2,3,4\}$ has each member appearing strictly once in any row/column of the three 5×5 squares. When any two of these MOLS are stacked together and one reads through the 25 aligned cell-pairs, each unique pair $\langle i, j\rangle (i \in I, j \in I)$ also appears exactly once.

Theorem 1. *With any given order n, there can be at most $(n-1)$ MOLS, with this upper bound achieved when n is a power of a prime number.*

Theorem 2. *When n is a power of a prime number, a complete set of $(n-1)$ MOLS can be constructed by filling the i^{th} square L_i $(0 < i < n)$ using $L_i[x,y] = i{\cdot}x + y$.*[1]

With such construction, the first row of all $n-1$ MOLS are identical, as shown in Figure 2. However, below the first row, the corresponding values at coordinates $[x,y]$ across all or any subset of the $n-1$ MOLS are *guaranteed* to be distinct.

Next we reuse the Latin squares shown in Figure 2 to illustrate how RAID+ works. With RAID+ templates, the order of Latin squares (n, 5 in this case) corresponds to the disk pool size. The number of Latin squares "stacked together" (k, 3 in this case) corresponds to the RAID stripe width. Suppose we now construct a logical RAID-5 volume, distributing data blocks with a stripe width of 3 across 5 disks.

We ignore the first row of all squares and construct $n(n-1)$ *stripes* by copying contents from the remaining $n(n-1)$ cells across all three squares. The middle column in Figure 2 gives a full list of these 20 stripes. The derived $n(n-1)$ stripe sequence guides block assignment onto the n disks: each item maps to the corresponding disk ID. *E.g.*, the first stripe ("stripe a") is made by looking up the $[1,0]$ cell of L_0, L_1, and L_2, resulting in tuple $\langle 1,2,3\rangle$. Its 3 blocks (2 data and 1 parity) will thus reside on disks (1, 2, 3), respectively, while those from "stripe b" will reside on disks (2, 3, 4), and so on.

The right column in Figure 2 gives the resulted data layout from the disks' point of view. As these 20 3-block stripes guarantee a uniform distribution of disk ID numbers, the 60 blocks form a 5×12 *RAID+ template*, to be repeatedly used in distributing data to the 5 disks.

The intuition is that aside from the first row, k MOLS give us uniform and deterministic data distribution across n disks, with k-block stripes. Unlike traditional RAID systems, where the disk array size equals the stripe width, our MOLS-based design allows k (and the RAID type) to be decoupled from n, enabling the construction of different virtual RAID volumes with small or moderate stripe widths on top of much larger disk pools.

As to be discussed in more details later, another desirable feature of MOLS is that, when one of the disks fails, blocks needed to recover the lost data are also *uniformly distributed among the $n-1$ surviving disks*. This allows for quick read, reproduction, and write of the (temporarily) lost data in parallel by these $n-1$ disks.

4 Normal Data Layout

4.1 Valid Disk Pool Sizes of RAID+

The precondition of building a RAID+ system is that we can construct $(k+m)$ n-order MOLS, k of which used to construct the normal data layout and m reserved as *spare MOLS* for data redistribution in the face of disk failures (details in Section 5). m should be large enough to support the highest fault tolerance level among all RAID volumes within this RAID+ pool. For example, if a volume adopts RAID-6, then we need $m \geq 2$.

Although the number of n-order MOLS for general n remains an open problem, $(n-1)$ is known to exist when n is a power of a prime number [7]. Therefore, as long as n, the total number of disks, is one such *valid pool size*, one can perform the deterministic calculation of $n-1$ MOLS using the algorithm given in Theorem 2. Also, with these valid n values, the corresponding MOLS set possesses several attractive properties for balanced data and recovery load distribution.

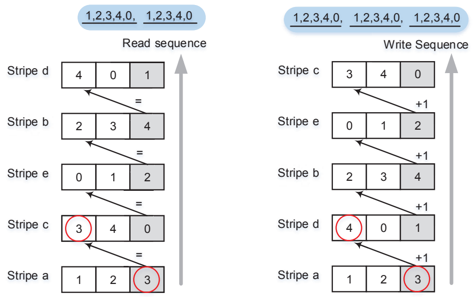
The requirement may sound demanding, but it turns out qualifying numbers are abundant and not far apart. For example, between 4 and 128, we have the following 42 valid n values: 4, 5, 7, 8, 9, 11, 13, 16, 17, 19, 23, 25 27, 29, 31, 32, 37, 41, 43, 47, 49, 53, 59, 61, 64, 67, 71, 73, 79, 81, 83, 89, 97, 101, 103, 107, 109, 113, 121, 125, 127, and 128. Since our envisioned RAID+ disk pools contain dozens of to 100+ disks, there are plenty of valid n values to choose from.

The density of valid n values further allows physical performance isolation should it be desired. Multiple RAID+ logical sub-pools can be constructed within a larger physical pool. *E.g.*, a 60-disk pool can support sub-pools with configuration (11+49), (23+37), (8+11+41), etc., all with valid sub-pool n values.

4.2 Stripes-to-Disks Mapping

RAID+ supports two modes, a *normal layout*, with guaranteed uniform distribution across all disks, and an *interim layout*, with uniform or slightly skewed data distribution among surviving disks, under one or more disk failures. Below we give more formal discussion of data

[1] "$\cdot$" and "$+$" here denote *finite field* multiplication and addition [7].



(a) Read friendly (b) Write friendly

Figure 3: Stripe order for a 5-disk array ($k = 3$). Gray indicates parity blocks. The head of each stripe equals the tail of the previous one added by 0 (read-friendly) or 1 (write-friendly).

organization with RAID+ under normal operation with a valid initial disk pool size n, with failure recovery and interim layout discussed in the next section.

Given k MOLS of order n, $\{L_0, L_1, ..., L_{k-1}\}$, a *RAID+ template* is constructed by traversing these k Latin squares simultaneously in a row-major order from the second row on. For each position $[x,y]$ ($0 < x < n$, $0 \leq y < n$), the k-block stripe $S_{x,y}$ is obtained by listing the corresponding values of L_i at this position: $S_{x,y} = \{L_0[x,y], L_1[x,y], ..., L_{k-1}[x,y]\}$, giving the disk IDs to place the k blocks of $S_{x,y}$. Since $n-1$ rows with n columns in the Latin squares are traversed, there are $n(n-1)$ stripes in a full cycle of this RAID+ template.

Below are the major properties of MOLS-based data layout. The proofs are omitted due to the space limit.

Property 1. *With normal data layout, any two blocks within a data stripe are placed on separate disk drives.*

This property guarantees that (1) the I/O workload in accessing each k-block stripe is uniformly distributed to k disks, and (2) a single disk failure results in the loss of at most one data block within any stripe.

Property 2. *With normal data layout, the n disks are assigned equal shares of both data and parity blocks.*

This property guarantees the same read-write load balancing as with RAID-5, allowing equal distribution of both data and parity blocks. This is particularly important to storage devices with asymmetric read-write performance and/or write leveling requirement, such as NAND flash disks. Unlike RAID-5, though, RAID+ decouples the stripe size k from a potentially much larger pool size n, allowing load balancing to be performed at a much wider scope, without sacrificing the fault tolerance allowed by the adopted RAID level.

4.3 Throughput-Friendly Addressing

So far, the RAID+ template gives a deterministic mapping from data blocks in any k-block stripe to n disks. However, since each stripe will be mapped to a k-subset of n disks, the ordering of the $n \times (n-1)$ stripes within the logical address space has impact on disk contention, I/O parallelism utilization, and spatial locality.

To this end, RAID+ allows stripe ordering (block addressing) to be done in different ways considering workload-specific needs. In particular, different RAID+ volumes sharing the same physical n-disk pool can each adopt its own addressing strategy. Below we describe two sample addressing algorithms targeting large sequential reads and writes, respectively (considering that block addressing matters less with random accesses). For the ease of illustration, we adopt simple RAID-4, where the first two blocks in each stripe are data blocks and the last one parity. The key difference between the two patterns here is that with RAID redundancy, sequential reads will skip parity blocks while sequential writes need to update both data and parity.

RAID+ performs stripe ordering by rows in the MOLS, with the process repeated at each row. To form the n-stripe sequence for the x^{th} row ($S_{x,0}$, $S_{x,1}$, ..., $S_{x,n-1}$), RAID+ starts by setting $S_{x,0}$ as the stripe given at the $[x,0]$ position of the MOLS, walking through the remainder of the row as follows:

- *Sequential-read friendly ordering* The head of each subsequent stripe is the tail of its predecessor (Figure 3(a)). *I.e.*, we choose $S_{x,i}$ such that $S_{x,i}(0) = S_{x,i-1}(k-1)$. The rationale here is that the last block within a RAID-4 stripe is a parity block, which will not be involved in user read operations.
- *Sequential-write friendly ordering* The head of each subsequent stripe is the sum of the tail of the previous one and x (Figure 3(b)). *I.e.*, we choose $S_{x,i}$ such that $S_{x,i}(0) = S_{x,i-1}(k-1) + x$. This is considering that for full-stripe writes resulted from sequential write workloads, all the blocks within a stripe will be updated.

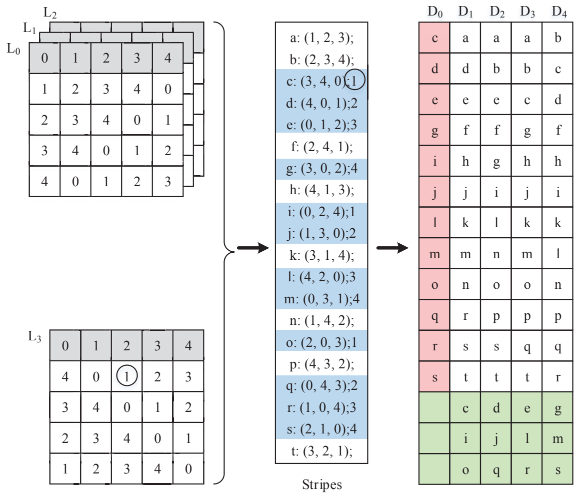
Finally, such logical ordering of stripes within a RAID+ volume also corresponds to the relative ordering of blocks on each disk. *E.g.*, the middle column in Figure 2 gives a "plain" row-major stripe ordering (neither read- nor write-optimized). This ordering uniquely defines the block ordering on each of the 5 disks (the column below each disk ID). In this case, D_0 carries blocks assigned to "0": the 3rd block in stripe c, the 2nd in d, the 1st in e, ..., etc. Given n, k, the block addressing scheme, and the RAID level adopted, the logical to physical block mapping within any RAID+ template can be completed by simple calculation. Our implementation uses a tiny lookup table (sized 93KB for $n = 59$ and $k = 7$) to accelerate such in-template data addressing.

4.4 Multi-Template Storage/Addressing

One major advantage of RAID+ is to accommodate multiple virtual RAID arrays (volumes) within the same shared disk pool, each servicing different users/workloads. Every such virtual array comes with its own MOLS-based template, stripe width k_i and RAID level, block size, as well as block addressing scheme. Within the large n-disk physical address space, capacity is allocated at the granularity of RAID+ templates.

RAID+ uses a per-volume index table to store the physical locations of its template instantiation. Since the templates are rather large, containing $n(n-1) \times k_i$ blocks for the i^{th} volume, maintaining such mapping (one "base address" per template instance) brings little overhead. Logical blocks of a given volume can then be easily mapped to its physical location by coupling the proper template instance offset with in-template addressing discussed earlier. Compared to random data distribution [12, 16, 41, 51], RAID+ offers uniform data distribution and direct addressing while only requiring single-step, template-level offset maintenance and lookup.

5 Data Recovery and Interim Layout



0	1	2	3	4
1	2	3	4	0
2	3	4	0	1
3	4	0	1	2
4	0	1	2	3

0	1	2	3	4
4	0	1	2	3
3	4	0	1	2
2	3	4	0	1
1	2	3	4	0

a: (1, 2, 3);
b: (2, 3, 4);
c: (3, 4, 0);1
d: (4, 0, 1);2
e: (0, 1, 2);3
f: (2, 4, 1);
g: (3, 0, 2);4
h: (4, 1, 3);
i: (0, 2, 4);1
j: (1, 3, 0);2
k: (3, 1, 4);
l: (4, 2, 0);3
m: (0, 3, 1);4
n: (1, 4, 2);
o: (2, 0, 3);1
p: (4, 3, 2);
q: (0, 4, 3);2
r: (1, 0, 4);3
s: (2, 1, 0);4
t: (3, 2, 1);

D_0	D_1	D_2	D_3	D_4
c	a	a	a	b
d	d	b	b	c
e	e	e	c	d
g	f	f	g	f
i	h	g	h	h
j	j	i	j	i
l	k	l	k	k
m	m	n	m	l
o	n	o	o	n
q	r	p	p	p
r	s	s	q	q
s	t	t	t	r
	c	d	e	g
	i	j	l	m
	o	q	r	s

Figure 4: Interim data layout for a 5-disk RAID+ array when one disk fails. RAID+ uses the fourth MOLS, L_3, to generate uniform data distribution.

The MOLS-based RAID+ data distribution offers all-to-all fast data recovery involving all surviving disks in a disk pool, directly into an *interim layout*. When the failed disk gets repaired or replaced, the normal data layout can be restored in background.

5.1 Interim Layout under 1-disk Failure

Upon a disk failure in an n-disk pool, RAID+ performs fast data recovery to recalculate the lost blocks and distribute them to the $(n-1)$ surviving disks. Thanks again to MOLS properties, when n is a valid pool size (power of prime), the resulted *interim data layout* preserves uniform data distribution. Suppose disk D_f $(f \in [0, n-1])$ fails, below we describe the construction of the interim layout with k-block stripes, reusing the former example. Figure 4 illustrates this process, where D_f is D_0.

Let R be the set of stripes affected by D_f's failure, which contain the item f (0 in this example). For any template, there are a total of $n(n-1)k/n = (n-1)k$ blocks on each disk. As each stripe cannot have two blocks assigned to the same disk, these $(n-1)k$ blocks correspond to the same number of stripes that are involved in data recovery, as shown in the middle column of Figure 4.

Recall that a valid pool size n allows for $(n-1)$ n-order MOLS, out of which k are used for the normal layout. Now we select any one from the remaining $(n-k-1)$ MOLS, as L_k (Latin square L_3 in Figure 4). This additional Latin square will be used to guide the placement of blocks assigned to D_f, with the item f in the affected stripes replaced with a new surviving disk ID.

The intuition is that when we append the new Latin square to the back of the existing k MOLS "stack" and read through each position $[x, y]$, we extend the k-block stripes to $(k+1)$-block ones, with again uniformly distributed item-set permutations. Now take each affected stripe, and replace the (now missing) f with the item r at the corresponding position in L_k, we relocate the missing block used to be assigned to D_f to the surviving disk D_r. E.g., in Figure 4, each "0" in these 12 stripes would be replaced with another integer in $\{1, 2, 3, 4\}$, such as stripe c transforming from $(3, 4, 0)$ to $(3, 4, 1)$, as the corresponding position in the additional Latin square ($L_3[1, 2]$) has item "1". The first two blocks in this affected stripe, on D_3 and D_4 respectively, would not need to move.

Below are the major properties associated with MOLS-based data layout concerning data recovery and the interim layout, assuming a valid pool size n.

Property 3. *With the n-disk normal layout, all blocks correlated with those on any given drive (i.e., blocks sharing stripes with blocks on this disk) are distributed evenly among the other disks.*

The implication here is that when the first disk fails, the read workload to recover unavailable blocks is evenly distributed among all surviving disks.

Property 4. *With the $(n-1)$-disk interim layout, any two blocks within a data stripe are still placed on separate disk drives.*

Property 5. *All the $(n-1)k$ missing blocks on any single failed disk can be redistributed to all the surviving $(n-1)$ disks evenly, each receiving k additional blocks.*

These two properties imply that (1) the write workload involved in RAID+ recovery from a single-disk failure is also uniformly distributed among all surviving disks, (2) data stripes in the $(n-1)$-disk interim layout preserves the same RAID fault tolerance as in the normal layout, and (3) the $(n-1)$-disk interim layout also retains the uniform data distribution to allow perfectly balanced I/O servicing even after losing one disk.

The particular significance of $(n-1)$-disk interim layout lies in the fact that the probability of single-disk failure is much higher than that of having two or more failed disks, especially when hot spares are available. Considering this, plus that disk capacity is relatively abundant in typical server environments, RAID+ performs an additional performance optimization by reserving recovery data space with normal data layout. More specifically, RAID+ actually allocates $n \times k$ physical blocks per disk for a data template. $(n-1)k$ of them are used to store data/parity blocks in the normal layout, while the remaining k blocks (the green area in Figure 4) are reserved for storing reconstructed data whenever there is a single-disk failure. This way, under such a failure, the reconstructed data are physically adjacent to the normal layout blocks, preserving spatial locality in data accesses. In our implementation, the content of the aforementioned small lookup table is modified to support fast interim data addressing, without additional space overhead.

5.2 Parallel Data Recovery

Under a single disk failure, the MOLS-based design lets both read and write workloads involved in RAID reconstruction and temporary relocation be uniformly distributed to the entire pool. This breaks the performance limit of conventional RAID systems, where the recovery work is only distributed within the RAID array affected.

However, even with uniform data distribution, the parallel read/write operations in data recovery could still generate resource contention, transient load imbalance, or unnecessary disk seeks, if care is not taken. To this end, RAID+ orchestrates its all-to-all data reconstruction by letting the surviving disks work on a subset of the $(n-1)k$ affected stripes at a time, alternating between reader and writer roles. Barriers are used in between such iterations, creating a natural break point for RAID+ to check upon user I/O requests, potentially slowing down or temporarily suspending the recovery depending on the current application request intensity, QoS specifications, and configurable system policies (such as starvation prevention to ensure the completion of data recovery).

5.3 Multiple Disk Failures

MOLS-based design also handles multiple failures gracefully. If another disk failure occurs after data recovery from a disk failure, we repeat the process described in Section 5.1 with another spare Latin square. When m disks are lost but tolerated by the adopted RAID level, by appending m spare MOLS to the stack of k used in the normal layout, we can calculate the eventual $(n-m)$-disk interim layout. Recognizing that the *affected* data stripes have different degrees of data loss, RAID+ prioritizes the reconstruction of the more vulnerable stripes.

Due to space limit, we give a brief summary of related results: when a RAID+ pool keeps losing disks (without disk replacement), Monte Carlo simulation shows very slight imbalance in data distribution (CoV of up to 0.29%), while system experiments show application performance degradation of up to 6% (except with sequential read, where RAID+ loses the benefit of its unique read-friendly addressing when more disks fail).

6 Evaluation

We implemented RAID+ in the MD (Multiple Devices) driver in Linux Kernel 3.14.35, a software RAID system that forms a common framework for all RAID systems tested in our evaluation. Despite theoretical properties appearing sophisticated, MOLS-based addressing is simple to implement, taking a mere 12 lines of code.

Test platform Our testbed uses a SuperMicro 4U storage server with two 12-core Intel XEON E5-2650 V4 processors and 128GB DDR4 memory, running Ubuntu 14.04 with Linux kernel v3.14.35. Two AOC-S3008L-L8I SAS JBOD adapters, each connected to a 30-bay SAS3 expander backplane via two channels, host 60 Seagate Constellation 7200RPM 2TB HDDs. The I/O channels afford a total I/O bandwidth of 24GB/s (400MB/s for each disk), significantly exceeding the aggregate sequential bandwidth from the disks. In all experiments, 50GB capacity of each disk is used.

RAID configurations Unless otherwise noted, our tests use 59 out of the aforementioned 60-disk pool ($n = 59$). The stripe width k is set at 7 (6+1 RAID).

For comparison, we evaluate two commonly adopted conventional RAID organizations utilizing such large disk pools, both of which build eight 6+1 RAID-5 arrays with 64KB stripe unit size, consuming 56 disks with the last 3 reserved as hot spares. *RAID-5C* divides each array into multiple 1GB extents and concatenates them in a round-robin manner, while *RAID-50* stripes across these 8 arrays at block size of 12MB (2MB×6).

We also evaluate two randomized data placement schemes, RAID_R and RAID_H. Both place blocks within each stripe to different disks while aiming for balanced data distribution to all n disks. To assign a block to a disk, RAID_R utilizes the system random number generator (with system time as seed) and is therefore nondeterministic. RAID_H, instead, uses the Jenkins hash function [26] adopted by systems such as Ceph [9, 50]. If a block is mapped to a disk already used in the current stripe, mapping will be recalculated until collision free (following CRUSH [51] for RAID_H). Our experiments find the two schemes often perform similarly, in which case we show only results of the better one.

# of disks	RAID-50	RAID_R	RAID_H	RAID+
$56+3$	$307s$	$60s$	$102s$	$41s$
$28+3$	$307s$	$99s$	$143s$	$83s$

Table 2: Offline rebuild time comparison

Both RAID+ and the random schemes build (6+1) RAID-4 arrays, with 64KB stripe unit size. Such striping continues on the same set of disks until a 2MB space allocation unit is filled per disk, before starting a new 7-block stripe. We have also implemented RAID-6 and observed similar performance trends, but omit results here due to space limit.

I/O Workloads We use three types of I/O workloads:

- *Synthetic workloads:* we use `fio` [15], a widely used I/O workload generator to produce four representative elementary workloads: sequential read, sequential write, random read, and random write.
- *I/O traces:* We use 8 public block-level I/O traces, namely `src1_1`, `usr_1`, `prn_0`, `prn_1`, `proj_0`, and `prxy_1` from MSR Cambridge [37], plus `Fin2` and `WebS_2` from SPC [1]. Based on load level observed, we followed existing practice in prior research [17, 27, 48] and accelerated the SPC traces (`Fin2` by 5× and `WebS_2` by 3×) while replaying all others a tempo.
- *I/O-intensive applications:* we also use four I/O-heavy real applications: GridGraph [60] (an out-of-core graph engine), TPC-C [46] (in-house implementation of the well-known RDBMS transaction benchmark standard), a Facebook-like photo access workload (synthesized using Facebook's published workload characteristics [4, 24]), and a MongoDB [35] NoSQL workload from the YCSB suite [11].

6.1 Reconstruction Performance

We start by evaluating reconstruction performance, one major advantage of RAID+ over alternative schemes, with disk failures created by unplugging random disk(s).

Offline rebuild Table 2 gives the offline rebuild time with a single-disk failure. RAID+ is tested with two valid n values, 59 and 31 (shared by the random schemes). The same pools would give RAID-50 3 hotspares each, with 8 and 4 RAID-5 arrays respectively. RAID-5C results would be identical to RAID-50 here.

In both cases, RAID+ consistently outperforms RAID-50, delivering a speedup of 7.5× and 3.7×. Such results approach the theoretical speedup of 8.29 and 4.29, respectively, given in Table 1. The gap is mainly due to less sequential reconstruction read/write patterns compared with RAID-50, as RAID+'s recovery load per disk is much smaller yet non-contiguous. Unlike RAID-50, with rebuild time independent of the disk pool size, RAID+ spreads the rebuild workload to larger pools uniformly and lowers the rebuild time proportionally.

The two random schemes outperform RAID-50 here also by having more disks participate in recovery. However, their rebuild takes significantly longer than that of RAID+. Further examination reveals that though they achieve overall balanced data distribution, random schemes suffer much higher skewness within each window of dozens/hundreds of blocks. E.g., within a RAID+ template size, the RAID_H has CoV of rebuild read/write load distribution of 31.9%/38.9%, while RAID_R has 12.72%/33.87%.[2] Such "local load balance" is crucial for RAID rebuild, with sequential and synchronized operations, where overloaded stragglers could easily drag down the entire array's recovery progress. RAID+, in contrast, retains its absolute load balance within such smaller windows and delivers much higher rebuild speed.

Finally, this advantage grows with the disk pool size n, as such perfect local load balance gives RAID+ higher profit margin by evenly utilizing n disks. In this sense, RAID-50 has recovery bandwidth independent of n by utilizing a small fixed-size sub-pool. The random schemes perform between these two extremes, achieving good global yet poor local load balance.

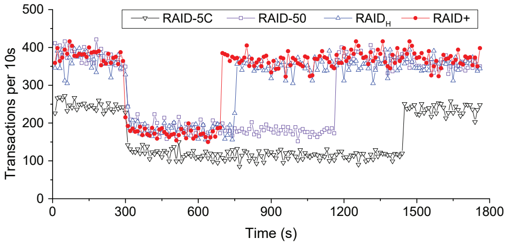


Figure 5: TPC-C online rebuild w. single-disk failure

Single-workload online rebuild Next, we examine online rebuild by creating a single-disk failure and performing reconstruction without stopping the execution of application(s). Figure 5 illustrates one sample test case (TPC-C). It plots the number of transactions committed per 10-second episode along the timeline, with a disk failure incurred at 300 seconds into the execution.

First, results demonstrate that RAID+ matches the `TPC-C` throughput of RAID-50 and RAID_H (all beating RAID-5C, unsurprisingly) in normal operation. Second, RAID+ offers much shorter online rebuild time than conventional RAID (396 seconds vs. RAID-5C's 1137 and RAID-50's 858). RAID_H comes closer, but still takes 11.4% longer than RAID+. Third, RAID+, RAID-50, and RAID_H bring similar degrades to TPC-C performance during rebuild. Although RAID-5C sees smaller relative performance impact, its degraded performance still lags behind due to its lower baseline. Overall,

[2] Here RAID_R has more even read distribution due to larger read volume than write in rebuild. RAID_H however exhibits more skewed distribution of blocks to read involved in recovery, with CoV level appearing to be dependent on the hash function used.

App		RAID-50	RAID-5C	RAID$_H$	RAID+
FaceBook	*app perf*	1	1.02	1.41	1.42
	reb perf	1	1.05	2.29	2.36
TPC-C	*app perf*	1	0.61	1.00	1.03
	reb perf	1	0.75	1.94	2.17
GridGraph	*app perf*	1	0.27	1.22	1.23
	reb perf	1	3.14	2.05	2.06
MongoDB	*app perf*	1	0.89	0.99	1.01
	reb perf	1	1.05	1.60	2.08

Table 3: Online rebuild performance comparison, in terms of speedup against corresponding RAID-50 results

during the 900 seconds following the disk failure's onset, RAID+ manages to complete 44.43%, 139.70%, and 5.11% more transactions than RAID-50, RAID-5C, and RAID$_H$, respectively. TPC-C throughput stays consistent as recovery progresses, as its degradation is dominated by the rebuild I/O activities rather than transactions that happen to hit the failed disk.

Table 3 summarizes online reconstruction performance, giving both the application performance and the rebuild speed, all in the form of speedup with respect to corresponding RAID-50 results (the higher the better). We use the same RAID rebuild rate setting (minimal at 80MB/s and maximum at 200 MB/s) within the MD driver for RAID-50 and RAID-5C, and configure RAID+ and RAID$_H$ to avoid application performance degradation from the RAID-50 baseline during rebuild (with only one exception where RAID$_H$ achieves 99% of the baseline performance). The results reveal that RAID+ and RAID$_H$ simultaneously improve both the application and rebuild performance from RAID-50. Between them, RAID+ is consistently better, with significantly faster rebuild and slightly better application performance for TPC-C and NoSQL. RAID-5C, at least with the default rate control setting, loses on both fronts (except for FaceBook, where it slightly outperforms RAID-50).

Multi-workload online rebuild For multi-workload evaluation, we use a smaller pool size of 29,[3] with stripe width remaining at 7, to construct 4 logical RAID volumes. RAID-5 builds 4 disjoint 6+1 arrays, plus one last disk reserved as hot spare. RAID+ constructs 4 volumes with the same deterministic template ($n = 29, k = 7$) across the entire pool. RAID$_H$ randomly distributes blocks from 4 virtual 6+1 array volumes to all 29 disks.

In each experiment, we sample 4 out of 8 MSR/SPC I/O traces as a *workload mix* to run simultaneously on the RAID volumes, for 28 minutes. The requests are replayed using the original timestamps, therefore identical sets of requests are issued across tests. We create a single-disk failure in the whole pool at time 0 and perform reconstruction without stopping user applications.

Figure 6 illustrates one such test case, showing the average I/O request latency in 60-second episodes along the execution timeline, for each workload. With RAID-5, the failure is contained within one volume (running Fin2 in Figure 6(a)), while with other schemes, it affects all volumes. The vertical lines indicate time points when each scheme finishes online rebuild. Similar to single-workload results, RAID+ has slightly faster rebuild than RAID$_H$, both beating RAID-5 by almost 4 times. Intuitively, RAID+ and RAID$_H$ excel by spreading rebuild work to all 4 volumes rather than only one, which also enables them to eliminate dramatic latency increases brought by RAID-5's online rebuild (Figure 6(a)). Thus compared with RAID-5, during the entire reconstruction, RAID+ and RAID$_H$ reduces the Fin2 workload average latency by over 90%, and the 99% tail latency by 89% (48*ms* vs. 418*ms*).

As expected, involving all disks expose the failure to all 4 volumes, roughly doubling the average latency of RAID+/RAID$_H$ before rebuild completes over RAID-5 for the prxy_1 and WebS_2 workloads. However, the much shorter rebuild time not only reduces system vulnerability, but also prevents any volume to be under dramatic performance degradation for prolonged periods. Note that while inter-volume isolation is broken here, disk failures are not user application artifacts but anomalies from the underlying platform. Therefore, RAID+ allows a larger disk pool to become more resilient, recover faster from failures, and provide more consistent performance during recovery.

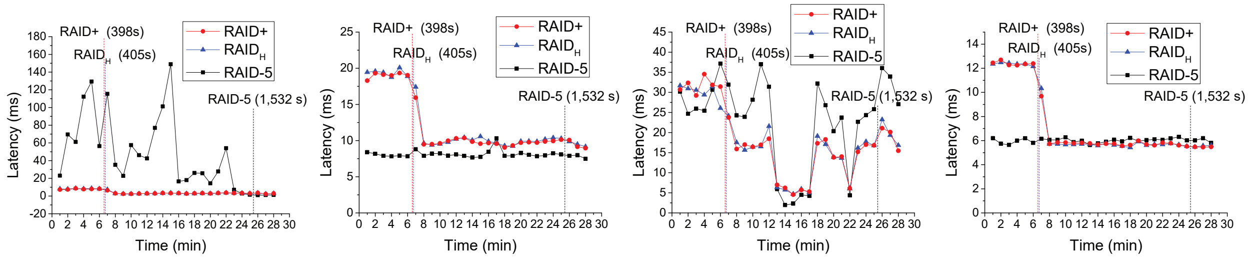
6.2 Normal I/O Performance

Single-workload evaluation Figure 7 gives the normal synthetic workload performance running fio, with varying request sizes. The access footprint is large enough to span all RAID-5 arrays with RAID-5C. All RAID systems, including RAID+, perform very similarly in random read/write tests. Therefore, we only show sequential performance here.

RAID+ slightly outperforms RAID-50 in most cases, by using 59 rather than 56 disks. Compared to them, RAID$_H$ offers moderately lower sequential performance, again due to poor local load balance and inferior spatial locality within each disk. RAID-5C predictably lags behind others with sequential accesses, as in most cases only one RAID-5 array is utilized.

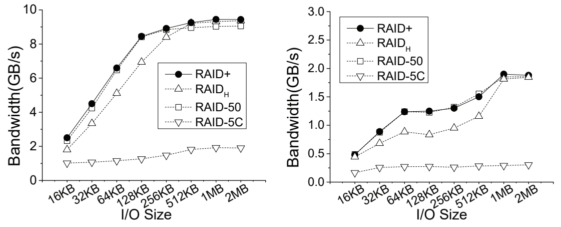
Figure 8 shows results with two sample MSR traces, plotting latency data points (averaged over 60-second episodes) along the execution timeline. Again RAID+ outperforms RAID-50, with an average improvement of 6.23% under prxy_1 and 87.04% under usr_1. This is because RAID+ uses all 59 disks (rather than 56) and usr_1 is read-dominant [37], with RAID+ adopts read-friendly addressing in this set of tests. For similar

[3] Considering the moderate request levels in test programs/traces, the smaller pool size allows us to test smaller (and higher number of) workload mixes, with results easier to plot and analyze.



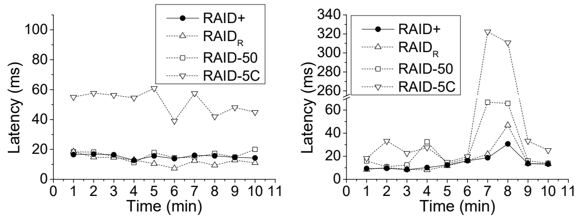
(a) Fin2 (b) prxy_1 (c) src1_1 (d) WebS_2

Figure 6: Sample multi-workload performance w. online rebuild



(a) Sequential read (b) Sequential write

Figure 7: Normal fio sequential performance

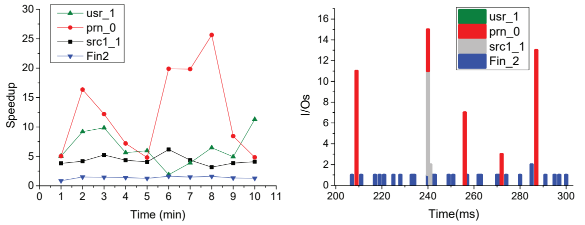


(a) prxy_1 (b) usr_1

Figure 8: Normal trace workload performance

	RAID-5C	RAID-50	RAID$_H$	RAID+
FaceBook (*s*)	168.18	165.85	176.20	168.8
MongoDB (*s*)	160.85	150.76	147.02	147.34
GridGraph (*s*)	1021.86	236.96	236.85	220.98
TPC-C (*TpmC*)	1345.2	2193	2192	2265

Table 4: Normal application performance



(a) RAID+ speedup over RAID-5 (b) Sample load variation

Figure 9: Case study with sample 4-workload mix

reasons, RAID$_R$ loses slightly to RAID+ under usr_1, and wins slightly under prxy_1, as it also uses all disks and the read-friendly addressing may bring minor side-effects for the more write-intensive prxy_1 workload.

Finally, Table 4 compares application performance. Note that unlike other cases, TPC-C uses transactions per minute committed, the higher the better. With Facebook-like photo and MongoDB, both having primarily random accesses, all four RAID organizations have data distributed to all disks and report very similar performance results. Since GridGraph is primarily sequential, RAID-5C can mostly utilize only one or two underlying RAID-5 arrays. Therefore, all three other schemes have a more than 4-fold speedup over RAID-5C, and RAID+ has minor advantage over RAID-50 by using slightly more disks, and over RAID$_H$ by having better spatial locality. With TPC-C, which has both random and sequential accesses, RAID+ slightly outperforms both RAID-50 and RAID$_H$. Again RAID-5C clearly underperforms.

Multi-workload system throughput Now we examine normal performance with multiple workloads sharing the underlying disk pool, using the 29-disk, 4-volume setting similar to that in online reconstruction tests (Figure 6). We evaluated all unique 4-workload combinations from the 8 MSR/SPC traces, executing each of these 70 workload mixes on 4 RAID volumes built with RAID-5, RAID$_H$, and RAID+.

Here we adopt *weighted speedup* [14], a widely-used multi-workload performance metric in computer architecture, to measure the overall system throughput. As each workload is replayed at fixed speed (by timestamps given in traces), we use $1/latency$ to replace the typical IPC (Instructions Per Cycle) measurement in architecture studies, calculating the weighted speedup as $\frac{1}{n}\sum_{i=1}^{n}(L_i^c/L_i)$ for an n-application workload mix. Here L_i^c and L_i denote the average latency of the ith workload using conventional RAID (RAID-5) and the system to be evaluated, respectively. *I.e.*, RAID-5 is used as the baseline for measuring performance speedup.

To summarize the results, both RAID$_H$ and RAID+ deliver considerable weighted speedup in all 70 test cases, demonstrating their capability of consistently improving the overall system throughput by utilizing more disks simultaneously. More specifically, RAID$_H$ obtains an average weighted speedup of 1.83 (over 1.33 in 80% of cases) over the 4-volume RAID-5 baseline, while

RAID+ performs better, with average weighted speedup of 2.05 (over 1.5 in 80% of cases).

Figure 9 showcases one sample test case (running src1_1, usr_1, prn_0, and Fin2), showing the speedup (based on average latency in each one-minute episode) of RAID+ over RAID-5 along the timeline. All but one speedup data points are above 1, with prn_0 reaching over 25 at one point. By zooming into the request patterns, we find such large profit comes from the burstiness in most workloads. Figure 9(b) illustrates this for a 100ms-long window, 200ms into the execution, showing the per-ms request count for each workload. At this granularity, one clearly sees that the workloads have sporadic requests and often form interleaving bursts. The most bursty workloads, prn_0 and src1_1, benefit more from RAID+ and RAID$_{\mathrm{H}}$, which use all disks to serve each volume. In particular, during request peaks these workloads see faster processing and lower I/O queue wait time, as confirmed by our detailed profiling, hence achieving the 25× (transient) speedup.

While the majority of the 70 mixes possess such "complementary" request patterns, there are cases where two or more workloads have sustained simultaneously intensive I/O activities, leading to slowdown of individual workload. However, among the total of 280 executions (70 mixed runs with 4 workloads per mix), there are only 39 cases of slowdown. Again, if a workload has to be guaranteed stronger performance isolation, RAID+ pools can be physically partitioned (such as building 41+19 volumes within a 60-disk enclosure).

6.3 Sensitivity to Internal Parameters

Finally, we study the impact of RAID+'s key parameters. Figure 10(a) shows both the aggregate random read and write throughput (left *y* axis) and the offline rebuild time (right *y* axis) with RAID+ pool size *n*, increased from 41 to 59, while fixing *k* at 7 and block size at 2MB.

These results show that the random read performance increases linearly with *n* (by up to 39%), due to uniform load distribution to all disks in the pool. The write throughput, though also growing steadily (by up to 24%), is much lower, as each of these 64KB write will bring at least four underlying I/O operations, for reading and writing back both the concerned data and parity blocks. In addition, such read-modify-write operations are synchronized, further lowering the aggregate throughput. The offline rebuild time, unsurprisingly, decreases as *n* grows and conforms to the model shown in Table 1.

Figure 10(b) shows similar experiments, with *n* fixed at 59 and varying *k*. With regard to user I/O performance, as modeled in Table 1, the aggregate throughput is mostly independent of *k* and the rebuild time grows linearly with it. One unexpected exception is with $k=3$, where the write bandwidth appears considerably higher than any other *k* values. By using the iostat tool, we find that with $k=3$ there are significantly fewer disk reads for parity calculation. Here with the 2+1 data-parity setup, there are higher chances for parity data to be reconstructed from cached data blocks.

Next, in Figure 10(c) we fix both *n* (59) and *k* (7) and change the block size. As expected, the block size has little impact on the random read/write performance. Meanwhile, the rebuild time decreases significantly, though not linearly. As RAID+'s rebuild access pattern introduces less regular access patterns compared with those of conventional RAID systems, larger block sizes improve performance by promoting sequential accesses.

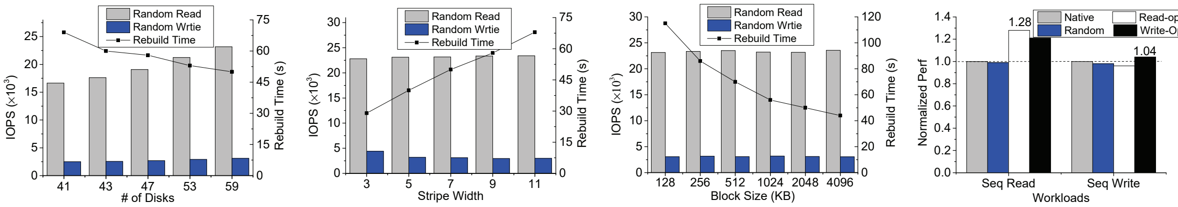
Last, to examine the effect of throughput-friendly block addressing (Section 4.3), we run the fio sequential read and write workloads, with I/O sizes of 2MB. Figure 10(d) shows the results using four stripe ordering strategies: 1) "native", original stripe ordering from a RAID+ template (stripes *a* to *t* in Figure 2), 2) "random", randomized stripe ordering using a pseudo-random function, 3) "read-opt", our proposed read-friendly ordering, and 4) "write-opt", our proposed write-friendly ordering. Bandwidths shown are normalized to "native". While the native and randomized strategies almost perform identically, the read-friendly strategy does generate a 28% improvement with sequential reads. Write-friendly ordering, on the other hand, brings a much smaller profit (4%). Again, unlike the "pure" sequential streams with reads, writes are not exactly sequential due to read-modify-write of both data and parity blocks.

7 Related Work

Data Layout Optimization Existing RAID layout optimizations roughly form two categories: 1) distributing either data blocks or parity blocks evenly across all the disks (*e.g.*, RAID-5 vs. RAID-4), and 2) exploiting spatial data locality (*e.g.*, left-symmetric RAID-5 [30]). Inspired by them, RAID+ spreads data and parity blocks in a much larger shared pool, with its throughput-friendly block addressing promoting access locality.

The parity declustering layout [36] utilizes as few disks as possible in data reconstruction, further analysed and extended/optimized by many [2, 10, 18, 20]. However, unlike with RAID+, rebuild is still capped by the write speed of the replacement disk, though these solutions do spread rebuild reads to remaining disks.

ZFS [6] uses "dynamic striping" to distribute load across "virtual devices", dynamically adjusting striping width and device selection to facilitate fast out-of-place updates and balanced capacity utilization. Systems such as IBM XIV [25] and the Flat Datacenter Storage (FDS) [38] use pseudo-random algorithms to distribute replicated data across all drives. If used for build-



(a) Different disk numbers (b) Different stripe widths (c) Different block sizes (d) Different stripe orders

Figure 10: Impact of several factors on RAID+'s performance

ing logical RAID volumes, in stripe placement ZFS and FDS would follow round-robin order (with optimizations on starting point selection), resulting in limited recovery bandwidth as a fixed "neighborhood" of disks would carry data relevant to recovery. XIV behaves similar to the $\mathrm{RAID_R}$/$\mathrm{RAID_H}$ schemes we evaluated.

Work also exists on designing data organizations sensitive to workload characteristics or application scenarios. *E.g.*, Disk Caching Disk (DCD) [23, 39] uses an additional disk as a cache to convert small random writes into large log appends. HP's AutoRAID [53] partitions RAID storage and differentiates the handling of hot and cold data. ALIS [22] and BORG [5] reorganize frequently accessed blocks (and block sequences) to place them sequentially in a dedicated area. These techniques are orthogonal to ours and can be incorporated by RAID+ to improve application performance.

Optimizations on RAID Reconstruction Prior studies have targeted improving reconstruction performance. Many of them focus on designing better data layout in a disk group [18, 29, 33, 52, 56], to minimize I/O for recovery or distribute rebuild I/O as evenly as possible. Other approaches optimize the RAID reconstruction workflow to make full use of higher sequential bandwidth, such as DOR (Disk-Oriented Reconstruction) [20], PR [31], and others [18, 42, 52, 55, 56]. In addition, PRO [45] rebuilds frequently-accessed areas first and S^2-RAID [48] optimizes reads and writes separately for faster recovery. Finally, task scheduling techniques optimize reconstruction rate control [32, 44, 47].

Except for WorkOut [54], which outsources part of user requests to surrogate disks during reconstruction, existing studies focus on improvement within one RAID group. RAID+ takes a different path from all, with built-in "backup" layouts to utilize all disks in a larger pool in reconstruction, while maintaining the fault tolerance and flexibility of smaller, logical RAID arrays.

RAID Scaling Adding disks to an array requires data movement to regain uniform distribution. Zhang et al. proposed batch movement and lazy metadata update to speed up data redistribution [57, 58]. FastScale [59] uses a deterministic function to minimize data migration while balancing data distribution. CRAID [34] uses a dedicated caching partition to capture and redistribute only hot data to incremental devices.

Another approach is *randomized RAID*, which randomly chooses a fraction of blocks to be moved to newly added disks. Prior work to this end [8, 16, 41] reduces migration, but produces unbalanced distribution after several expansions [34]. Also, existing randomized RAID systems require extra book keeping and look-up.

RAID+, in contrast, allows large disk enclosures to directly host user volumes, each using its own RAID configuration, with templates stamping out allocations in all shapes and sizes. Meanwhile, it is not designed for dynamic, heterogeneous distributed environments targeted by methods like CRUSH [51].

8 Conclusion

This paper proposes RAID+, a new RAID architecture that breaks the resource isolation between multiple co-located RAID volumes and allows the decoupling of stripe width k from disk group size n. It uses a novel Latin-square-based data template to guarantee uniform and deterministic data distribution of k-block stripes to all n disks, where n could be much larger than k. It also delivers near-uniform distribution in both user data and RAID reconstruction content even after one or several disk failures, as well as fast RAID rebuild.

With RAID+, users can deploy large disk pools with virtual RAID volumes constructed and configured dynamically, according to different application demands. By utilizing all disks evenly while maintaining spatial locality, it enhances both multi-tenant system throughput and single-workload application performance.

9 Acknowledgment

We thank all reviewers for their insightful comments, and especially our shepherd Gala Yadgar for her guidance during our camera-ready preparation. We also thank Mu Lin and Xiaokang Sang for helpful discussions. This work was partially supported by the National Natural Science Foundation of China (under Grant 61672315) and the National Grand Fundamental Research 973 Program of China (under Grant 2014CB340402).

References

[1] Umass trace repository. http://traces.cs.umass.edu/index.php/Storage/Storage, 2017.

[2] Guillermo A. Alvarez, Walter A. Burkhard, Larry J. Stockmeyer, and Flaviu Cristian. Declustered disk array architectures with optimal and near-optimal parallelism. In *Proceedings of 25th International Symposium on Computer Architecture (ISCA'98)*, pages 109–120, 1998.

[3] Eitan Bachmat and Jiri Schindler. Analysis of methods for scheduling low priority disk drive tasks. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'02)*, pages 55–65, 2002.

[4] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*, pages 47–60, 2010.

[5] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis. BORG: Block-reORGanization and self-optimization in storage systems. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*, pages 183–196, 2009.

[6] Jeff Bonwick and Bill Moore. ZFS: The last word in file systems. https://wiki.illumos.org/download/attachments/1146951/zfs_last.pdf, 2007.

[7] Raj Chandra Bose and Sharadchandra S Shrikhande. On the construction of sets of mutually orthogonal Latin squares and the falsity of a conjecture of Euler. *Transactions of the American Mathematical Society*, 95(2):191–209, 1960.

[8] André Brinkmann, Kay Salzwedel, and Christian Scheideler. Efficient, distributed data placement strategies for storage area networks. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'00)*, pages 119–128, 2000.

[9] Ceph. libcrush. https://github.com/ceph/libcrush, 2017.

[10] Siu-Cheung Chau and Ada Wai-Chee Fu. A gracefully degradable declustered RAID architecture. *Cluster Computing*, 5(1):97–105, 2002.

[11] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SOCC'10)*, pages 143–154, 2010.

[12] Netapp Corporation. How long does it approximately take for a RAID reconstruction? https://kb.netapp.com/support/s/article/ka21A0000000jOzQAI/how-long-does-it-approximately-take-for-a-raid-reconstruction?language=en_US, 2017.

[13] Oracle Corporation. A better RAID strategy for high capacity drives in mainframe storage. http://www.oracle.com/technetwork/articles/systems-hardware-architecture/raid-strategy-hi-capacity-drives-170907.pdf, 2013.

[14] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE micro*, 28(3), 2008.

[15] fio. https://github.com/axboe/fio, 2017.

[16] Ashish Goel, Cyrus Shahabi, Shu yuen Didi Yao, and Roger Zimmermann. SCADDAR: An efficient randomized technique to reorganize continuous media blocks. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, pages 473–482, 2002.

[17] Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, Pedro García-López, Yosef Moatti, and Eran Rom. Crystal: Software-defined storage for multi-tenant object stores. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, pages 243–256, 2017.

[18] Mark Holland and Garth 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 (ASPLOS'92)*, pages 23–35, 1992.

[19] Mark Holland, Garth A. Gibson, and Daniel P. Siewiorek. Fast, on-line failure recovery in redundant disk arrays. In *Proceedings of The Twenty-Third International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 422–431, 1993.

[20] Mark C. Holland. *On-line data reconstruction in redundant disk arrays*. PhD thesis, Pittsburgh, PA, USA, 2001.

[21] Robert Y. Hou, Jai Menon, and Yale N. Patt. Balancing I/O response time and disk rebuild time in a RAID5 disk array. In *Proceedings of the Twenty-sixth Hawaii International Conference on System Sciences (HICSS-26)*, pages 70–79 vol.1, 1993.

[22] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. The automatic improvement of locality in storage systems. *ACM Transactions on Computer Systems (TOCS)*, 23(4):424–473, November 2005.

[23] Yiming Hu and Qing Yang. DCD - Disk Caching Disk: A new approach for boosting I/O performance. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA'96)*, pages 169–178, 1996.

[24] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of

Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)*, pages 167–181, 2013.

[25] IBM. IBM XIV storage system architecture and implementation. `http://www.redbooks.ibm.com/redbooks/pdfs/sg247659.pdf`, 2017.

[26] Robert J. Jenkins. Hash functions for hash table lookup. `http://burtleburtle.net/bob/hash/evahash.html`, 1997.

[27] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, pages 61–74, 2014.

[28] Hannu H. Kari, Heikki K. Saikkonen, Nohpill Park, and Fabrizio Lombardi. Analysis of repair algorithms for mirrored-disk systems. *IEEE Transactions on Reliability*, 46(2):193–200, 1997.

[29] Osama Khan, Randal Burns, James Plank, William Pierce, and Cheng Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, pages 251–264, 2012.

[30] Edward K. Lee and Randy H. Katz. The performance of parity placements in disk arrays. *IEEE Transactions on Computers (TOC)*, 42(6):651–664, Jun 1993.

[31] Jack Y. B. Lee and John C. S. Lui. Automatic recovery from disk failure in continuous-media servers. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 13(5):499–515, 2002.

[32] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, David Nagle, and Erik Riedel. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation (OSDI'00)*, pages 87–102, 2000.

[33] Jai Menon and Dick Mattson. Distributed sparing in disk arrays. In *Digest of Papers COMPCON Spring 1992*, pages 410–421, 1992.

[34] Alberto Miranda and Toni Cortes. CRAID: online RAID upgrades using dynamic hot data reorganization. In *Proceedings of the 12th USENIX conference on File and Storage Technologies (FAST'14)*, pages 133–146, 2014.

[35] MongoDB. `https://www.mongodb.com/`, 2017.

[36] Richard R. Muntz and John C. S. Lui. Performance analysis of disk arrays under failure. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB'90)*, pages 162–173, 1990.

[37] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10:1–10:23, 2008.

[38] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, pages 1–15, 2012.

[39] Tycho Nightingale, Yiming Hu, Qing Yang, Tycho Nightingale Y, Yiming Hu Z, and Qing Yang Y. The design and implementation of a DCD device driver for Unix. In *Proceedings of the 1999 USENIX Technical Conference (ATC'99)*, pages 295–308, 1999.

[40] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM International Conference on Management of Data (SIGMOD'88)*, pages 109–116, 1988.

[41] Beomjoo Seo and Roger Zimmermann. Efficient disk replacement and data migration algorithms for large disk subsystems. *ACM Transactions on Storage (TOS)*, 1(3):316–345, August 2005.

[42] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'04)*, pages 15–30, 2004.

[43] Marc Staimer and Antony Adshead. Post-RAID alternatives address RAID's shortcomings. `http://www.computerweekly.com/feature/Post-RAID-alternatives-address-RAIDs-shortcomings`, 2010.

[44] Eno Thereska, Jiri Schindler, John S. Bucy, Brandon Salmon, Christopher R. Lumb, and Ganger R. Ganger. A framework for building unobtrusive disk maintenance applications. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)*, pages 213–226, 2004.

[45] Lei Tian, Dan Feng, Hong Jiang, Ke Zhou, Lingfang Zeng, Jianxi Chen, Zzhikun Wang, and Zhenlei Song. PRO: A popularity-based multi-threaded reconstruction optimization for RAID-structured storage systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, pages 277–290, 2007.

[46] tpcc mysql. `https://github.com/Percona-Lab/tpcc-mysql`, 2017.

[47] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Ganger R. Ganger. Argon: Performance insulation for shared storage servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, pages 5–5, 2007.

[48] Jiguang Wan, Jibin Wang, Changsheng Xie, and Qing Yang. S^2-RAID: Parallel RAID architecture for fast data recovery. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 25(6):1638–1647, 2014.

[49] Zhufan Wang. Reliability analysis on RAID+. `https://github.com/RAIDPLUS/Additional-materials/raw/master/reliability.pdf`, 2018.

[50] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 307–320, 2006.

[51] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC'06)*, 2006.

[52] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A. Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the Panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, pages 2:1–2:17, 2008.

[53] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer System (TOCS)*, 14(1):108–136, February 1996.

[54] Suzhen Wu, Hong Jiang, Dan Feng, Lei Tian, and Bo Mao. WorkOut: I/O workload outsourcing for boosting RAID reconstruction performance. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*, pages 239–252, 2009.

[55] Tao Xie and Hui Wang. MICRO: A multilevel caching-based reconstruction optimization for mobile storage systems. *IEEE Transactions on Computers (TOC)*, 57(10):1386–1398, 2008.

[56] Qin Xin, Ethan L. Miller, and Thomas J. E. Schwarz. Evaluation of distributed recovery in large-scale storage systems. In *Proceedings of 13th International Symposium on High-Performance Distributed Computing (HPDC'04)*, pages 172–181, 2004.

[57] Guangyan Zhang, Jiwu Shu, Wei Xue, and Weiming Zheng. SLAS: An efficient approach to scaling round-robin striped volumes. *ACM Transactions on Storage (TOS)*, 3(1):3:1–3:39, 2007.

[58] Guangyan Zhang, Weiming Zheng, and Jiwu Shu. ALV: A new data redistribution approach to RAID-5 scaling. *IEEE Transactions on Computers (TOC)*, 59(3):345–357, March 2010.

[59] Weiming Zheng and Guangyan Zhang. FastScale: Accelerate RAID scaling by minimizing data migration. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, pages 149–161, 2011.

[60] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC'15)*, pages 375–386, 2015.