

Adaptive Data Block Placement Based on Deterministic Zones (AdaptiveZ)

J.L. Gonzalez¹ and Toni Cortes^{1,2}

¹ Department d Arquitectura de Computadors(DAC),
Universitat Politecnica de Catalunya

joseluig@ac.upc.es

² Barcelona Supercomputing Center(BSC)

toni@ac.upc.es

Abstract. The deterministic block distribution method proposed for RAID systems (known as striping) has been a traditional solution for achieving high performance, increased capacity and redundancy all the while allowing the system to be managed as if it were a single device. However, this distribution method requires one to completely change the data layout when adding new storage subsystems, which is a drawback for current applications

This paper presents AdaptiveZ, an adaptive block placement method based on deterministic zones, which grows dynamically zone-by-zone according to capacity demands. When adapting new storage subsystems, it changes only a fraction of the data layout while preserving a simple management of data due to deterministic placement. AdaptiveZ uses both a mechanism focused on reducing the overhead suffered during the upgrade as well as a heterogeneous data layout for taking advantage of disks with higher capabilities. The evaluation reveals that AdaptiveZ only needs to move a fraction of data blocks to adapt new storage subsystems while delivering an improved performance and a balanced load. The migration scheme used by this approach produces a low overhead within an acceptable time. Finally, it keeps the complexity of the data management at an acceptable level.

1 Introduction

The constant growth of new data (at an annual rate of 30% [1] and even 50% for several applications [2]), and the rapid decline in the cost of storage per GByte [3] has led to an increased interest in storage systems able to upgrade their capacity online.

The periodical upgrade of storage systems results in heterogeneous storage environments because a constant improvement in disk capabilities has been observed year after year [3]. Therefore, the addition of new disks requires a compromise between taking advantage of disks with higher capabilities [4] and avoiding wasting disk capacity and/or performance.

On the other hand, scientific and database applications are particularly sensitive to storage performance and expandability issues because of their imposing

I/O requirements, which in some cases can account for between 20 to 40 percent of total execution time [5]. These kinds of applications require storage systems capable of delivering fast service times.

One of the greatest challenges is to design storage systems that can handle capacity demands by adapting new storage subsystems yet achieve high performance, strong data availability, and simple management when faced with huge volumes of information.

Optimizing the data layout on disks is the key to accomplishing such objectives. However, a data layout looking at multi-objective optimization could end up optimizing some objectives more than others.

For example, deterministic data placement, or striping, used in traditional RAID systems [6] delivers high performance, strong data availability[7], and a simple management of information, while it requires a huge effort when adapting to new disks. The reason is, this technique distributes blocks in round robin fashion according to the number of disks in the array (C). A simple mod operation of C is used for locating blocks: $dsk = mod(block;C)$ and $position_within_disk = block/C$. This is quite efficient for reducing location overhead and does not degrade with huge volumes of information. However, if a new storage subsystem is added we must replace C with $C+1$, which involves upgrading all stripes to a new value of C . This technique, called Re-striping, is quite acceptable and maintainable for small environments [8][9].

On the other hand, random block placement [10] moves only a fraction of the data layout when adapting new disks, which results in a reduced time of adaptation. However, huge efforts are required for achieving high-performance I/O and data availability without increasing the management complexity of the data (more details in related work).

As we can see, applications with intensive I/O that require upgrading their storage systems must choose between either the benefits of a deterministic layout, and try to somehow reduce the re-striping time, or a random layout (which guarantees a reduced time of adaptation), and try to somehow deliver performance as close to optimal as possible and not have a complicated data management system.

In order to preserve the benefits of deterministic data placement while reducing the time of adaptation when adding new storage subsystems, we propose to perform re-striping of only a fraction of the data layout.

This paper presents AdaptiveZ, an adaptive data block placement method based on deterministic zones, which spreads blocks on disks and/or mid-ranged RAIDs that are managed as a single device. AdaptiveZ grows dynamically zone-by-zone according to the capacity demands and uses the new storage subsystems added to the system according to their capabilities.

AdaptiveZ uses a migration algorithm for achieving a trade-off between balancing the load of each disk and increasing the overall parallelism of the data layout. The algorithm minimizes I/O operations and uses a mechanism for reducing the overhead by gradually making available the bandwidth of new storage subsystems during data migration.

2 Related Work

Random block placement on disk arrays [10] breaks the functional dependency between allocation and location. The position of a block into the disk array is managed by a hash function, which allows locating blocks by only reading it. An efficient expansion of the storage by moving only a fraction of the data layout was proposed in [11], [12], which works by only registering the changes of the migrated blocks.

SCADDAR [13] avoids the use of hash tables because blocks are placed onto disks in a random, but reproducible, sequence. In this pseudorandom approach each disk carries approximately equal load.

The random and pseudorandom layouts yield a global uniform block distribution on the disks and are able to manage heterogeneous disks. Nevertheless, they require a huge effort for distributing the accesses file by file on all the disks of the array. The reason is that file systems commonly try to place together all the blocks of a file [14], and a random distribution could happen to allocate all data blocks of a single file on only one disk or even in different parts of that disk producing large seek times in the worst case.

This is a drawback for applications with intensive I/O and high concurrency because these applications require spreading data of a file over almost all disks of the array as well as avoiding the overhead of seek times (to improve their I/O throughput). Using random striping can minimize the above effect [13], but not eliminate it.

In addition, for improving their I/O throughput, scientific, general-purpose and database applications commonly use data block sizes from 8 to 256 KB due to both their high concurrency as well as small size of their requests [15].

The management complexity of blocks increases with blocks of small sizes in a random layout because hash tables grow in accordance with the number of data blocks allocated into the storage system. Therefore, a hash table is only maintainable when the number of data blocks is reasonable. For example, a storage system of 15 TB using (large) data blocks of 32 MB requires a hash table to manage approximately half a million data blocks, which is completely reasonable, but if this storage system uses a 128 kB block size, which is a default configuration, then we must manage approximately one hundred million blocks. 192.168.1. In the case of pseudo-random placement the situation is very similar but problems arise when calculating the data block locations.

Logical Volumes Manager (LVM) [16] is a popular technique for using on-fly new storage subsystems, which are configured as virtual volumes/disks by the administrator and only involves new storage subsystems not old ones.

3 AdaptiveZ Overview

In this section we describe the AdaptiveZ approach, its data placement and the algorithm used for adapting new storage subsystems.

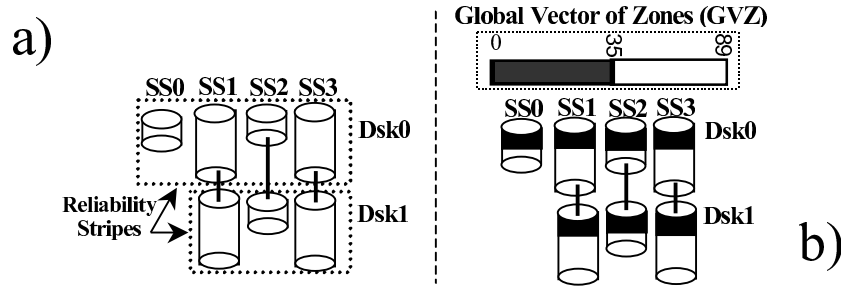


Fig. 1. (a) AdaptiveZ device of 4 SS per 2 horizontal lines with disks and reliability stripes of different sizes. (b) The same AdaptiveZ device using the GVZ.

3.1 AdaptiveZ Approach

AdaptiveZ allows the configuration of a set heterogeneous disks as a single device called an AdaptiveZ device, which can be configured as an orthogonal array[7] of mid-ranged RAIDs or as a simple disk array.

AdaptiveZ can be considered as a heterogeneous matrix of Storage Subsystems per reliability stripes, where a Storage Subsystem (**SS in the rest of the paper**) is a vertical line of n disks/RAID and a reliability stripe is a collection of horizontal disks/RAID (Figure 1 (a) shows an example of this approach).

A *zone* is an abstraction used by AdaptiveZ for allocating/locating blocks within the AdaptiveZ device. In this approach a *zone* is big horizontal part of the AdaptiveZ device on which the blocks are striped as in a traditional RAID.

AdaptiveZ can be configured as a single zone or multi-zone system at the beginning and it can add new zones according to the capacity/performance demands. In order to manage several zones, AdaptiveZ has defined a Global Vector of sequential Zones (GVZ). The file system can access it as a single address space.

In figure 1 (b) we can see how sequential zones make up a GVZ. In this example two zones have been allocated on the AdaptiveZ device: zone 0 (black) allocates 35 blocks in homogeneous manner on all disks, meanwhile zone 1 (white) allocates 54 blocks in a heterogeneous manner on disks at the end of zone 0.

When a *zone* is configured or re-arranged, it uses as many SS as available into the AdaptiveZ device. In this approach, each *zone* works as independent storage and manages its SS according to performance and/or migration needs.

In order to allocate/locate blocks on its SS, each *zone* uses a table with the following fields:

- **The number of SS on the zone (C_{zone}):** used in order to allocate blocks.
- **The redundancy:** used for data availability (mirror, parity, Orthogonal Striping and Mirroring, etc)
- **The last block of a zone:** used to determine the beginning of the next zone.

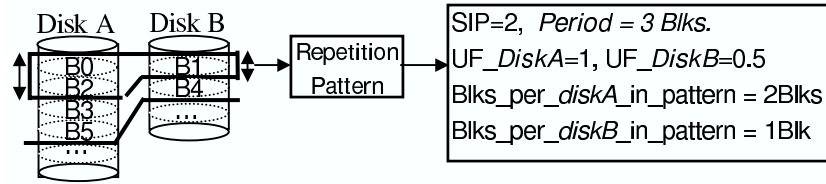


Fig. 2. An example of a repetition pattern

- **A heterogeneous pattern of repetition:** used in order to allocate/locate data blocks within the zone (more details in the data placement section).

3.2 AdaptiveZ Data Placement

In this section we show the method used for both distributing as well as locating data blocks.

Distributing Blocks in AdaptiveZ: AdaptiveZ distributes blocks sequentially on the GVZ (See (b) in figure 1). Afterwards, each zone is striped on disks by using its heterogeneous pattern (included in the table mentioned above).

AdaptiveZ designs the heterogeneous pattern for each zone based on the repetition patterns proposed in AdaptRaid [4].

An Overview of AdaptRaid's Patterns: A repetition pattern is a technique used for distributing blocks on a set of heterogeneous disks according to the utilization factor of each disk (UF). A UF is a number between 0 and 1 that AdaptRaid defines by using the bandwidth and capacity of a disk.

The UF of a disk indicates the % of blocks that a pattern can allocate per disk. For example, if $diskA$ is able to serve twice the number of requests per unit time than $diskB$ then $UF\ diskA = 1$ and $UF\ diskB = .5$.

The parameter $Blks_per_disk_in_pattern$ determines the amount of blocks per disk within the pattern, which is calculated by $Blks_per_disk_in_pattern = UF * SIP$. Where SIP represents the amount of Stripes In Pattern. The size of a pattern or $period$ is the sum of $Blks_per_disk_in_pattern$ value of all disks.

Figure 2 shows a pattern where three blocks ($period=3$) are distributed by two stripes ($SIP=2$) following the $Blks_per_disk_in_pattern$ value. This results in stripes of different sizes, where the first stripe was allocated on $disks\ A\ and\ B$ and the last only on $diskA$ because $Blks_per_disk_B_in_pattern$ only allows 1 block per pattern.

Obviously, the disks have more than only three blocks. Therefore, the pattern is repeated until all disks are full resulting in each disk being filled according to its behavior or UF (see two repetitions of a pattern in figure 2). This requires the pattern features being registered by using both $Blks_per_disk_in_pattern$ table with a size proportional to C disks on the array as well as two tables with a $period$ size for registering disk and RAID(if any) per each block of the pattern. For more details about this technique see [4].

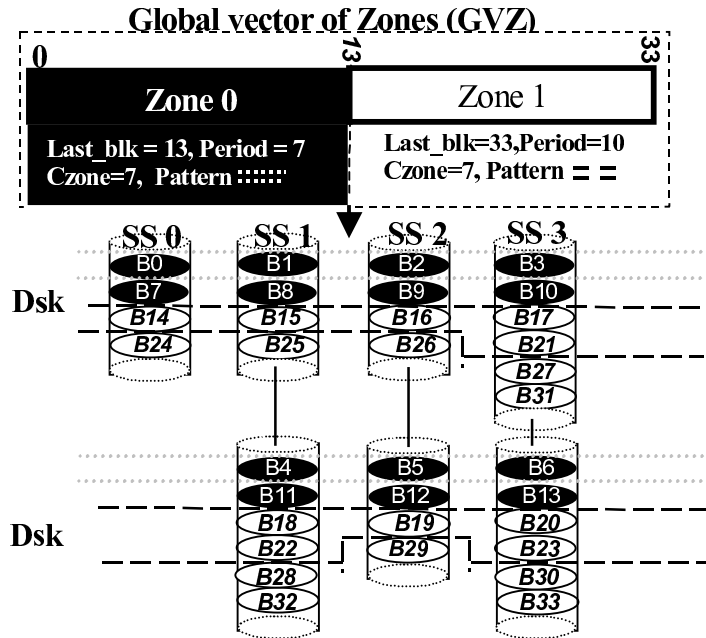


Fig. 3. Distributing blocks within two zones on an AdaptiveZ device

Designing Patterns in AdaptiveZ: AdaptiveZ defines the pattern of a zone depending on performance and/or migration needs by fine-tuning the *UF* of disks. Since a disk can allocate several zones, a disk can be used with different *UF* depending on the zone.

AdaptiveZ reduces the sizes of patterns by using the minimum *SIP* value, which reduces the usage of memory because less blocks are registered in disk and RAID tables. This means that AdaptiveZ only registers in the pattern's tables the stripes required for capturing the behavior of disks.

In figure 3 we can see an example of four heterogeneous SS in an AdaptiveZ device. Two zones have been configured: zone 0 with 14 blocks using a pattern with *period*=7, *SIP*=1 and zone 1 with 20 blocks and a *period*=10, *SIP*=2. In zone 0 disks A and B have a *UF* = .5 meanwhile, in zone 1 disk A has *UF* = 1 and B *UF* = .5. In this figure *UF* = .5 = 1 block and *UF* = 1 = 2 blocks. In both zones the pattern has been repeated two times.

As we can see in this figure, in each zone an amount of blocks equal to its *period* is distributed by *SIP* stripes on disks according to the *UF* of each disk. Afterwards, the pattern is repeated until the number of blocks is equal to the zone's *last block*.

Computing the Location of a Block: The GVZ works as a space address, which allows sequentially accessing the blocks by using the *last block* parameter. The GVZ is handled by a B-Tree delivering the zone where a requested block (*B*) has been allocated.

Once the zone has been chosen, AdaptiveZ then changes the block sequence (BZ) within the *chosen zone*. For example, in Figure 3 block 15 is really block 1 within zone 1, which results in a new sequence. This means if *chosen zone* = 0 then $BZ=B$ otherwise $\mathbf{BZ} = \text{block}(B) - (\text{last_block}(\text{chosen zone}-1)+1)$. Once the BZ has been determined, its position within *chosen zone* is easily calculated by using the following formulas:

$$\begin{aligned} \text{SS}(\mathbf{BZ}) &= \text{location}[BZ\%period].SS \\ \text{pos}(\mathbf{BZ}) &= \text{location}[BZ\%period].pos + (BZ/period) \\ &\quad *Blks_per_SS_in_pattern[BZ] \end{aligned}$$

Where *period* is the sum of all blocks in the pattern.

In the first formula we compute the SS where block BZ is. As we use a repetitive pattern, we first need to find the right position of the block in the pattern. This is easily computed using the modulo function of BZ divided by the *period* of the pattern. Then we can use this value to know the SS where the block is. We have this function computed in advance in SS table.

When the SS is a RAID we also calculate in advance in SS_RAID table thus AdaptiveZ knows the RAID and disk where block BZ is.

Now, we have to find the position of block BZ within the just computed SS. First, in the same way we computed the SS, we can compute the position of this block in the pattern (*pos*). Then, we add the number of blocks in this SS for each repetition pattern. This number of blocks is computed by multiplying the number of times the pattern has been repeated ($BZ/period$) with the number of blocks this SS has in a pattern ($Blks_per_SS_in_pattern[BZ]$).

3.3 Adapting New Storage Subsystems

The AdaptiveZ approach requires only adding a new zone at the end of the GVZ for using the space of a new SS with no data migration. However, problems such as performance and reliability arise when using that space because all data were allocated on old zones (horizontal) meanwhile the new zone can only allocate new data resulting in an unbalanced load. An unbalanced load reduces the parallelism offered by the new zone while increasing the load on old zones, which is not a good situation when trying to achieve reduced service times.

Re-striping is the key for avoiding the above problems. Nevertheless, performing an acceptable re-striping is not trivial because several requirements imposed by current applications must be observed when moving data blocks:

1. **Allocating the migrated blocks on zone(s) with similar or better parallelism than source zone(s).**
2. **Maximizing the parallelism offered by new zone(s).**
3. **Minimizing the time of data migration.**
4. **Balancing the amount of blocks per disk.**
5. **Minimizing the management complexity of the final layout.**

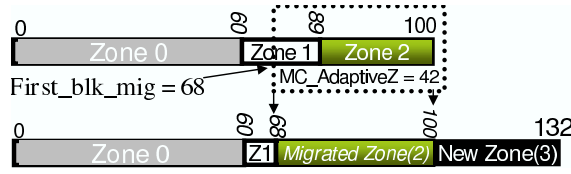


Fig. 4. The GVZ before (top) and after (bottom) of the migration process

AdaptiveZ proposes meeting these requirements in strict priority when adapting new storage subsystems.

Migration Issues: An overall re-striping of the data layout meets almost all of the above requirements. However, it produces a lot of data migration time when handling huge volumes of information, which delays the capacity and bandwidth utilization of new SS. Therefore, the algorithm starts by reducing the time employed when adapting new SS.

The Time of Migration Process: Moving only a part of the data layout is the key for reducing the time of the migration process. Thus, AdaptiveZ starts by calculating $IL_{AdaptiveZ}$, which represents the Ideal Load that each disk should have, new disks included, after the migration process:

$$IL_{AdaptiveZ} = Old_Capacity / (Old_Capacity + Capacity_New_SS)$$

$$MC_AdaptiveZ = (1 - IL_{AdaptiveZ}) * Old_Capacity$$

$IL_{AdaptiveZ}$ delivers a % that is used for determining $MC_AdaptiveZ$, which represents the minimum amount of blocks that the algorithm must migrate.

Migrating data blocks: Once having determined $MC_AdaptiveZ$, AdaptiveZ determines what part of the data layout will be re-striping:

$$first_blk_mig = Old_Capacity - MC_AdaptiveZ$$

Where $first_blk_mig$ is the point in the GVZ indicating where the algorithm must start the block migration. The blocks beyond $first_block_mig$ will be migrated to a zone called *migrated zone*. N number of stripes are read after $first_block_mig$ and then written in the *migrated zone* by performing a re-striping on all SS (new ones included). This extends the zones allocated beyond the $first_block_mig$ increasing its parallelism and solving the first requirement.

In Figure 4 we can see how AdaptiveZ marks block 68 as $first_block_mig$ on the GVZ (top) and afterward, we can also see (bottom) how all data blocks beyond block 68 have been migrated from zone 1 and 2 to the *migrated zone*.

The space obtained by the migration process plus the space of new SS is used for designing a *new zone*, which also uses all available SS of the AdaptiveZ device plus new SS. In the bottom part of Figure 4 we can see how a *new zone* is added to the end of GVZ after migration, which will be used for distributing new data. This operation solves the second requirement of migration.

Table 1. The difference between load per SS when applying re-striping and ideal load

	<i>SS 0</i>	<i>SS 1</i>	<i>SS 2</i>	<i>SS 3</i>	<i>SS 4</i>	<i>New SS</i>
% load re-striping <i>MC_AdaptiveZ</i>	92	92	92	92	92	21
% ideal load (<i>IL_AdaptiveZ</i>)	73	73	73	73	73	73
period	35	35	35	35	35	35
blks_per_SS_in_pattern	5	5	5	5	5	10
Total capacity SS (GB)	93	93	93	93	93	169

Balancing the Load per Disk: Although performing a re-striping of only a fraction of data layout (*MC_AdaptiveZ*) reduces the time employed by the migration process (meeting requirement 3), it does not distribute enough blocks on new SS for achieving a balanced load. The following formulas allow us to determine the *load per SS* produced by *AdaptiveZ* when performing a re-striping:

$$\begin{aligned} \text{Load_New_SS} &= (\text{Blks_per_NEW_SS_in_pattern} \\ &\quad * (\text{MC_AdaptiveZ}/\text{period})) / \text{Capacity_NEW_SS} \\ \text{Load_old_SS} &= 1 - ((\text{Blks_per_OLD_SS_in_pattern} \\ &\quad * (\text{MC_AdaptiveZ}/\text{period})) / \text{Capacity_OLD_SS}) \end{aligned}$$

Where *Blks_per_SS_in_pattern* is the sum of *Blks_per_disk_in_pattern* value of all disks when the SS is a RAID. Table 1 shows the addition of one SS (169GB) to an *AdaptiveZ* device (456GB) of five SS, 93 GB each. This table shows the difference between the *ideal load* and *load per SS* applying the above formulas when re-striping *MC_AdaptiveZ* (123.95GB). We can see in this example that re-striping partially results in more blocks allocated on old SS than new ones, which compromises a balanced load.

The Trade-off an Intuitive Idea: The trade-off consists in achieving an approximation to the ideal load for each SS when re-striping the *migrated zone*. This can be achieved by modifying the pattern that the *migrated zone* will use to allocate blocks.

To design the pattern of migrated zone, the algorithm deals with two factors:

1. Increasing *MC_AdaptiveZ* until 15% to increase the size of the *migrated zone*.
2. Modifying the *Blks_per_SS_in_pattern* parameter of all SS with a load $< / >$ than ideal load (*IL_AdaptiveZ*).

The next iterative algorithm is used for designing the *migrated zone's* pattern:

1. A *range* for the load approximation is defined (*range=5%* as default value, it can be fine-tuned).
2. The algorithm assigns the value of one block to the *Blks_per_SS_in_pattern* of the SS with lowest capabilities and the *Blks_per_SS_in_pattern* of the rest is computed according to it.
3. The *Load_per_SS* is calculated with that pattern and registered in a table.
4. The algorithm verifies whether the *Load_of_SS* yielded by that pattern is $<$ than (*IL_AdaptiveZ+range*) and $>$ than (*IL_AdaptiveZ-range*).
5. If all SS are within range, then we have a good approximation and this pattern will be used for performing the re-striping of the *migrated zone*.

Table 2. Applying trade-off step-by-step. * indicates the load yielded by the chosen pattern.

	<i>SS 0</i>	<i>SS 1</i>	<i>SS 2</i>	<i>SS 3</i>	<i>SS 4</i>	<i>New SS</i>
% Load re-striping MC_AdaptiveZ+15%	88	88	88	88	88	33
% Load trade-off 1rst. iteration	82	82	82	82	82	51
% Load trade-off 2nd. iteration	77	77	77	77	77	63
% * Load trade-off 3rth. iteration	74	74	74	74	74	71
% Ideal load (IL AdaptiveZ)	73	73	73	73	73	73
Total capacity SS (GB)	93	93	93	93	93	169

6. Otherwise, the algorithm increases the *Blks_per_SS_in_pattern* of all SS with a *Load_of_SS* < than (*IL AdaptiveZ-range*) and decreases the *Blks_per_disk_in_pattern* of all SS with a *Load_of_SS* > than (*IL AdaptiveZ+range*) (this is only allowed when the decrement produces *Blks_per_SS_in_pattern* > 0).
7. The algorithm goes to step 3.

In table 2 we can see the same example used in table 1, but now showing the *load per SS* that the iterative algorithm accomplished by tuning the pattern. Note that the algorithm is not migrating blocks yet but fine-tuning the pattern that will be used for performing the re-striping on the migrated zone.

AdaptiveZ Data Layout Migration after Migration: AdaptiveZ gradually distributes less new data blocks on old disks. It reduces bottlenecks in old disks (assuming old data have fewer accesses than new ones) and it produces isolation of old disks allowing to change them with no critical effects on the overall performance.

The Management of Data Blocks: This approach produces two zones per migration in the worst case. Nevertheless, when we increase by 30% [1] or 50%[2] the capacity of an AdaptiveZ device, quite coherent according to capacity demands, it produce one zone per migration and even a reduction of zones can be observed in some cases (More details in results section). Assuming two upgrades of the storage system per year, in a decade 20 zones could result which is not too much.

Reducing the overhead during data migration: Data migration involves overhead on the normal operation of the storage system during the migration process. The file system's requests suffer delays because the migration operations work on different zones of a disk producing seek times even when they are done in the background. Fortunately, mechanisms focusing on using the bandwidth of new disks has shown that the overhead can be gradually reduced until eliminating it even during migration [8],[9] and [17].

In AdaptiveZ, the new disks are gradually used for serving file system's requests during the migration process, which also reduces the accesses on old disks. The idea is based on keeping a mechanism working as a switch, which knows the last block that has been migrated and so when the file system requests blocks beyond that point, the mechanism switches to the area that has not been

Table 3. Specifications of workload used in each migration

	Migration 1	Migration 2	Migration 3
Arrival (ON/OFF model)	9 ms arrival time 500 ms OFF periods	7 ms arrival time 400 ms OFF periods	5 ms arrival time 300 ms OFF periods
% of read requests	70 Uniform distribution	(for all migrations)	
Request size	8Kb Poison distribution	(for all migrations)	
Request location	Uniform distribution 35% sequential	(for all migrations)	

re-striped and uses only old disks, otherwise it switches to the re-striped area, which includes new disks (more details in [8],[9]).

4 Methodology

We have performed a comparison between AdaptiveZ and random data placement because we are going to study the effect of moving only a fraction of the data layout and the movement has been done at random.

We have chosen pseudorandom placement because it can be applied with no metadata/name servers allowing a random distribution of blocks by means of an address space. In addition, it is focused on delivering an approximately equal number of blocks per disk yielding a uniform distribution of data by performing random striping.

4.1 Simulation and Workload Issues

In order to perform this evaluation, we have implemented both AdaptiveZ as well as the pseudorandom placement used in SCADDAR [13] on HRaid [18], which is a storage-system simulator that allows us to simulate storage hierarchy. In the SCADDAR case an offset was added for handling mirrors as proposed in [13] and a weigh was assigned to the disks when managing heterogeneous disks as proposed in [19]. Simulating the next two scenarios preformed the comparisons:

1) *The Storage System Before, During and After Migration(BDA Migration scenario)*: By using synthetic workloads, we can go forwards in time and perform several migrations in order to measure migration-by-migration both the overhead suffered when adding new SS to storage systems as well as the time taken by the migration process for both AdaptiveZ and random data placement.

This experiment makes sense because we want increased bandwidth and/or capacity when the storage system is not able to address an increment on demand. Using synthetic workloads, with increases in both their access and address space, can simulate this.

We have performed one workload per migration in order to simulate a data base system workload. The workload information has been extracted from [20] and their features have been increased according to [21].Table 3 shows the features of the workload used in each migration.

Table 4. Specifications of used disks

	<i>HAWK1</i>	<i>hp 97560</i>	<i>ST15230W</i>	<i>90871u2maxtor</i>	<i>WD204BB</i>	<i>ST136403LC</i>
Formatted capacity (GB)	1	1.28	4	8	18.7	33.87
Block Size (bytes)	1024	1024	512	512	512	512
Sync Spindle Speed(RPM)	4002	5400	5400	7200	7200	10000
Average Latency (ms)	8.2	5.7	5.54	5.54	4.16	2.99
Buffer	64KB	128KB	512Kb	1MB	2MB	4MB
DskID	A	B	C	D	E	F

Table 5. Specifications for BDA Migration Scenario

	<i>Starting Configuration</i>	<i>Migration 1</i>	<i>Migration 2</i>	<i>Migration 3</i>
Starting Capacity(GB)	467.5	654.5	993.2	1331.9
Added Capacity(GB)	0	187	338.7	338.7
Added SS	5	2	2	2
Disks per SS	5	5	5	5
Used disks (dskID from tab4)	E	E	F	F
SS ID	0	1	2	3

2) The Storage System after several Migrations (AS Migrations scenario) : By using real financial traces (OLTP [22]), we can go backwards in time and then start to perform migration after migration until we arrive at the year when the trace was created and until the address space of the real trace is accomplished. Afterwards, we measure the effects of data manageability, balanced load and performance.

This experiment makes sense because a real trace cannot be manipulated in order to predict new accesses, but we can use it on an environment that has been upgraded many times and observe the performance of both AdaptiveZ and pseudorandom placement after several migrations.

4.2 Configurations Studied

In order to evaluate the behavior of our proposals, we perform a set of tests for both scenarios. In both scenarios we always add between 20 % and 50% [1] [2] of the current capacity and choose disks according to technology's trends of hard disks [3] corresponding to each migration. Table 4 shows the features of the disks used by the SS in both scenarios. The DiskID was included to 4 for identifying each kind of disk in the rest of paper.

BDA Migration scenario: We have simulated a storage system of five SS with five disks each. We have performed three migrations to scale the capacity of the storage system from 467GB to 1.34TB, which is enough to analyze the overhead behavior as well as the performance, memory usage and load balancing. Table 5 shows disks and SS used in this scenario as well as details about the capacity added and the overall capacity of the storage system in each migration. The SS_ID means several SS have the same features.

Table 6. Specifications for AS Migrations scenario

	Mig 1	Mig 2	Mig 3	Mig 4	Mig 5	Mig 6	Mig 7	Mig 8	Mig 9	Mig 10	
Starting Cap(GB)	56	68.8	99.51	124.07	148.63	180.77	220.94	277.04	333.14	434.75	536.36
Added Cap(GB)	0	12.8	30.72	24.56	24.56	32.14	40.17	56.1	56.1	101.61	101.61
Added SS	7	1	2	1	1	1	1	1	1	1	1
Disks per SS	8	10	6	6	6	4	5	3	3	3	3
Used disks (diskID)	A	B	B	C	C	D	D	E	E	F	F
SS ID	0	1	2	3	4	5	6	7	8	9	10

We have tested three configurations per migration for both AdaptiveZ and SCADDAR:

1. **Start:** This is the current configuration of the storage system.
2. **Migrate:** This starts using *AdaptiveZ Start* Configuration and is gradually upgraded with the new storage subsystems.
3. **Final:** This is the resultant configuration once storage subsystems have been added.

AS Migrations Scenario: We have performed 10 migrations, resulting in two upgrades of the storage system a year until achieving the address space of the real trace. Table 6 shows the same information shown in table 5 but for 10 migrations. We have tested Start and Final configurations for AdaptiveZ and SCADDAR for this scenario.

5 Experimental Results

5.1 Evaluating BDA Migration scenario

Evaluating the Load per SS: Intuitively a balanced load results in an acceptable performance because it reduces potential bottlenecks.

Table 7 shows the difference between the *load per SS* delivered by AdaptiveZ in each migration and the *Ideal Load*. Table 7 shows that AdaptiveZ achieves a *load per SS* very close to the *Ideal Load*.

Evaluating the migration process: Once we have evaluated the *load per SS* that AdaptiveZ produced for this scenario, we can determine whether our intuition about a balanced load is correct by examining the performance and overhead for this scenario.

For the purposes of our work, we define **the overhead** produced by the migration work as the increment observed in service times of I/O requests.

Table 7. Difference between the load per SS delivered by AdaptiveZ and the Ideal Load

	Migration 1 (%)	Migration 2 (%)	Migration 3 (%)
SS_ID_0	0.2	1.6	0.7
SS_ID_1	-0.4	0.4	1.2
SS_ID_2		-2.5	0.3
SS_ID_3			2.0

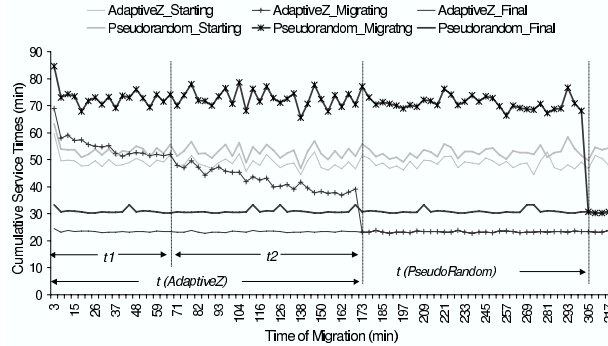


Fig. 5. Comparing service times of *AdaptiveZ Start*, *Migrate* and *Final* with *Pseudorandom Start*, *Migrate* and *Final* configurations during Migration 1

On the other hand, *the migration time* is the time required to complete the migration process. This time is represented by t in the rest of the paper. The time t is divided into two parts: $t1$ represents the overhead and $t2$ the improvement observed in service times during the upgrade process. Therefore $t = t1 + t2$.

In this part we first show the results obtained when performing the **first migration** (see details of the features of this experiment in table 5, column migration 1, and the used workload in table 2, column migration 1).

Figure 5 shows the service times observed for three different configurations with AdaptiveZ layout: *AdaptiveZ Start* (the un-upgraded configuration), *AdaptiveZ Final* (the upgraded configuration), and *AdaptiveZ Migrate* which starts as *AdaptiveZ Start* and is gradually converted to *AdaptiveZ Final*. The same is also shown for SCADDAR with *Pseudorandom Start*, *Pseudorandom Migrate* and *Pseudorandom Final*.

The horizontal axis represents the simulation time, which was measured every 50×10^3 requests. The vertical axis represents the cumulative service times for these requests, to compare easily each point in the three lines.

Comparing Start configurations: In figure 5 we can observe that *AdaptiveZ Start* configuration yields better services times than *Pseudorandom Start*. The reason is that both configurations use a 128 KB block size (a mean for evaluating environments). As we have already said, a random distribution can produce large seek times because some blocks of the same file can be allocated in the same disk and even in different positions within the disk. This effect is increased when the system uses small blocks with a high concurrency.

Comparing Migrate configurations: In figure 5 we can see both *AdaptiveZ Migrate* and *Pseudorandom Migrate* produced service times higher than the respective Start configurations. *AdaptiveZ Migrate* produces smaller overhead than *Pseudorandom Migrate*. The reason is *Pseudorandom Migrate* starts using the *Pseudorandom Start* configuration, which delivers high services times. In this figure we can see how *AdaptiveZ Migrate* has been gradually reducing the

Table 8. Times t_1, t_2 and t of migration process for BDA Migration scenario

	t_1 Overhead (hrs)	t_2 Improvement (hrs)	t migration time (hrs)
Migration 1	1.14	1.39	2.56
Migration 2	1.33	1.53	3.26
Migration 3	1.11	2.38	3.49

overhead (t_1) until it is eliminated, then we can see how the algorithm improves the storage performance during the greatest part of the migration process (t_2).

AdaptiveZ Migrate configuration yields this behavior because it is able to gradually use the new disks to serve file system requests. This behavior was observed when migrating blocks in previous studies for disk arrays [8], [9], for virtualized environments in [23]. The re-striping process reduces the amount of blocks per disk reducing disks seeks (of course this benefit will disappear when the array is fully upgraded and the file system is able to use the new blocks).

In the case of *Pseudorandom Migrate* configuration we can see a linear behavior because in this case SCADDAR are not using the rearranged fraction. Registering the blocks that have been reorganized and then making an indirection to the new location could solve this. However, this was not included in the original proposal [13] from where we have performed the implementation. We believe this is not a drawback of the SCADDAR proposal since the overhead can be handled in a similar way for both proposals.

There is a sudden drop in Figure 5 at the end for both *AdaptiveZ Migrate* and *Pseudorandom Migrate* configurations because at this point the migration process has been done and there are no delays introduced to incoming I/O requests.

At this point we can compare the migration time taken, t , for both configurations. The time t for *AdaptiveZ Migrate* is quite acceptable (See table 8). This is possible because in *AdaptiveZ* the blocks have been sequentially allocated and several blocks can be migrated with a single I/O Request. Meanwhile, the time t for *Pseudorandom Migrate* configuration is high because it has to migrate block-by-block, which produces one I/O Request per block due to the random allocation.

Comparing Final configurations: In figure 5 we can also observe that *AdaptiveZ Final* yields better service times than the *Pseudorandom Final* configuration. The reason is the same as when comparing the *Start* configurations for both proposals.

Note that SCADDAR was not designed for distributing blocks in these kinds of environments while *AdaptiveZ* was specially designed for this.

The results for migration 2 and 3: Figure 6 (left) shows above-mentioned comparisons but performed in migration 2 and 3 (right). As we can see, the behavior is quite similar. The difference with the first migration is that *AdaptiveZ Start* has three zones in the second migration and four in the third one. In addition, both in migration 2 and 3 *AdaptiveZ Migrate* yields even lower service times than *Pseudorandom Start*, which indicates that when increasing

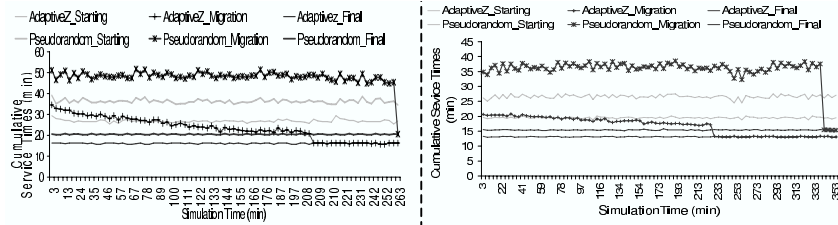


Fig. 6. The migration 2 (left) and 3 (right)

accesses, concurrency and space address produce an impact on the storage system using *Pseudorandom Start*. The time t for AdaptiveZ is quite acceptable (See table 8).

Evaluating the manageability of data AdaptiveZ: The data manageability of a storage system can be measured according to the time used for determining locations of data blocks as well as the memory used for that purpose. AdaptiveZ only uses Mod operations for locating blocks, which is the simplest method; so we have to measure the memory used for managing the GVZ AdaptiveZ. AdaptiveZ has only needed less than 1 Kb for managing the zones produced after 3 migrations. (More details about this issue are treated in the *AS Migrations Scenario*, where 10 migrations were performed).

5.2 Evaluating AS Migration Scenarios

Evaluating the Load per SS. This scenario allows us to evaluate the *load per SS* after several migrations.

In figure 7 we can see the *load per SS* produced by AdaptiveZ after ten migrations (See table 6 for details of this experiment). In this figure the horizontal axis shows ten migrations. The vertical axis represents the difference between the *load per SS* delivered by AdaptiveZ and the ideal load for each migration.

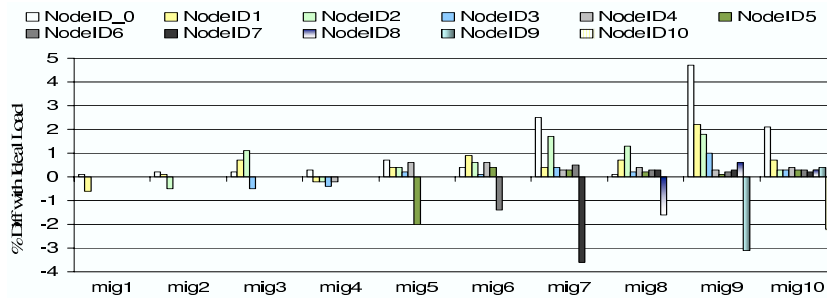


Fig. 7. Difference between the *load per SS* and the Ideal Load (*IL AdaptiveZ*)

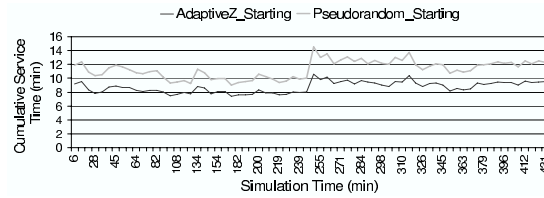


Fig. 8. Comparing service times of *AdaptiveZ Start* with *Pseudorandom Start* after ten migrations

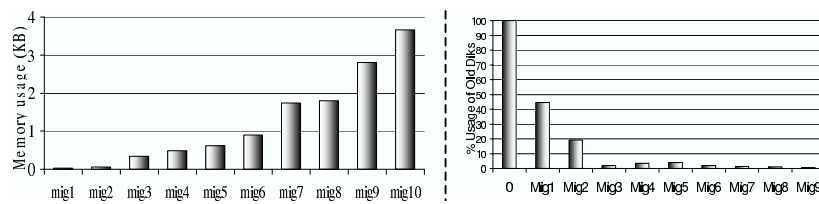


Fig. 9. The usage of memory (left). The usage of oldest disk (right) per Migration.

As we can see, AdaptiveZ achieves a *load per SS* very close to the Ideal Load, the difference is always less than 5% in the worst case.

Evaluating service times: In figure 8 we can observe that *AdaptiveZ Start* configuration yields better service times than *Pseudorandom Start* for this scenario. AdaptiveZ distributes more new data on the new SS, which reduces the load of the old disks reducing also the bottlenecks in the old disks. This can be kept even after several migrations.

Evaluating the manageability of data AdaptiveZ: This scenario allows us to evaluate whether, after 5 years and 10 migrations, AdaptiveZ is able to keep the memory usage for managing its global vector at an acceptable level.

In figure 9 (left) we can see the memory used by AdaptiveZ in each migration. As we can see, memory usage is not a problem for AdaptiveZ because after 10 migrations it requires less than 4 KB for management of its GVZ. Nevertheless, in this figure we can also see how the linear growth increases and some peaks can be observed. In each peak (for example in migration 3) the system has decided to not use the old disks thereby increasing the usage of the new disks in order to deliver a balanced load. Figure 9 (right) shows the % of usage of oldest disks within the *migrated zone* (vertical axis) in ten migrations (horizontal axis). As we can see, when a peak in figure 9 is observed, the usage of oldest disks is reduced for the *migrated zone*.

It could be a good warning sign for the storage administrator, indicating that old disks should be replaced with new ones when several memory peaks have been observed. When this happens, AdaptiveZ is effectively not using old disks for distributing new data. The benefits of a replacement at this time are more

than the costs because the capacity of these disks represents only a few % of the overall capacity of the AdaptiveZ device.

Besides the above issues, AdaptiveZ still uses all disks in each migration process while requiring only a few KB for managing its GVZ.

6 Conclusions

AdaptiveZ only rearranges a fraction of its data layout when adapting to new storage subsystems, which produces a reduced time of data migration with an acceptable overhead. The data layout designed by AdaptiveZ after the migration process yields both improved performance as well as a balanced load even after several migrations. The data management using AdaptiveZ preserves the benefits of deterministic data placement and does not degrade over time.

References

1. Charles, P., Good, N., Jordan, L.L., Lyman, P., Varian, H.R., Pal, J.: How much information? 2003? (2004), http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/printable_report.pdf
2. Coffman, K.G., Odlyzko, A.M.: Internet growth: is there a moore's law for data traffic?, 47–93 (2002)
3. Grochowski, E., Halem, R.D.: Technological impact of magnetic hard disk drives on storage systems. *IBM Syst. J. IBM Corp.*, 338–346 (2003)
4. Cortes, T., Labarta, J.: Taking advantage of heterogeneity in disks arrays. *Journal of Parallel and Distributed Computing* 63, 448–464 (2003)
5. Ailamaki, A., DeWitt, D.J., Hill, M.D., Wood, D.A.: Dbmss on a modern processor: Where does time go?, pp. 266–277 (1999)
6. Gibson, G., Patterson, D.A., Katz, R.H.: A case for redundant arrays of inexpensive disks (raid). *SIGMOD*, 109–116 (1988)
7. Gibson, G.A., Patterson, D.A.: Designing disk arrays for high data reliability. *Journal of Parallel and Distributed Computing*, 4–27 (1993)
8. Gonzalez, J.L., Cortes, T.: Increasing the capacity of raid5 by online gradual assimilation. In: *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/O*, p. 17 (2004)
9. Gonzalez, J.L., Cortes, T.: Evaluating the effects of upgrading heterogeneous disk arrays. In: *SPECTS 2006* (2006)
10. Santos, J.R., Muntz, R.: Performance analysis of the rio multimedia storage system with heterogeneous disk configurations. In: *ACM 1998*, pp. 303–308 (1998)
11. Brinkmann, A., Heidebuer, M., Meyer auf der Heide, F., Rückert, U., Salzwedel, K., Vodisek, M.: V:drive - costs and benefits of an out-of-band storage virtualization system. In: *21st MSST*, pp. 153–157 (April 2004)
12. Miller, E.L., Honicky, R.J.: A fast algorithm for online placement and reorganization of replicated data. In: *IPDPS 2003. 17th. International Parallel and Distributed Symposium*, pp. 267–268 (April 2003)
13. Yao, S.D., Zimmermann, R., Goel, A., Shahabi, C.: Scaddar: An efficient randomized technique to reorganize continuous media blocks. In: *ICDE 2002. IEEE 18th. International Conference on Data Engineering*, p. 473 (2002)

14. Leffle, S.J., McKusick, M.K., Joy, W.N., Fabry, R.S.: A fast file system for unix. *ACM Trans. Comput. Syst.* , 181–197 (1984)
15. Chen, P.M., Patterson, D.A.: Maximizing performance in a striped disk array. In: *ISCA 1990*, pp. 322–331 (1990)
16. Malschagen, H.: Logical volume management for linux
17. Zhang, G., Shu, J., Xue, W., Zheng, W.: Slas: An efficient approach to scaling round-robin striped volumes. *Trans. Storage* 3(1), 3 (2007)
18. Labarta, J., Cortes, T.: Hraid: A flexible storage-system simulator. In: *Proceedings of the International Conference on parallel and Distributed Processing Techniques and Applications*, vol. 163, p. 772. CSREA Press (1999)
19. Yao, S.D., Shahabi, C., Zimmermann, R.: Broadscale: Efficient scaling of heterogeneous storage systems. *Int. J. on Digital Libraries* , 98–111 (2006)
20. Franke, H., Gautam, N., Zhang, Y., Zhang, J., Sivasubramaniam, A., Nagar, S.: Synthesizing representative i/o workloads for tpc-h. In: *HPCA*, pp. 142–151 (2004)
21. Madhyastha, T.M., Hong, B., Zhang, B.: Cluster based input/output trace synthesis. In: *ipccc 2005* (2005)
22. OLTP Application I/O.
<http://traces.cs.umass.edu/index.php/storage/storage>
23. Brinkmann, A., Effert, S., Heidebuer, M., Vodisek, M.: Influence of adaptive data layouts on performance in dynamically changing storage environments. In: *PDP 2006*, pp. 155–162 (2006)