Extending Heterogeneity to RAID level 5*

T. Cortes and J. Labarta
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
{toni, jesus}@ac.upc.es, http:/www.ac.upc.es/hpc

Abstract

RAIDs level 5 are one of the most widely used kind of disk array, but their usage has some limitations because all the disks in the array have to be equal. Nowadays, assuming a homogeneous set of disks to build an array is becoming a not very realistic assumption in many environments, especially in low-cost clusters of workstations. It is difficult to find a disk with the same characteristics as the ones in the array and replacing or adding new disks breaks the homogeneity. In this paper, we propose a block-distribution algorithm that can be used to build disk arrays from a heterogeneous set of disks. We also show that arrays using this algorithm are able to serve many more disk requests per second than when blocks are distributed assuming that all disks have the lowest common speed, which is the solution currently being used.

1 Introduction

Heterogeneous disk arrays are becoming (or will be in a near future) a common configuration in many sites. Let us describe two scenarios that end up in a heterogeneous disk array. The first one appears whenever a component of a traditional array fails and it has to be replaced by a new one. As disk technology improves quite rapidly, it is quite probable for the new disk to be faster and larger than the ones already in the array [7]. A similar scenario appears when the capacity needs of a site grow and new disks have to be acquired to grow the size of the array (by increasing the number of disks in the array). In this case, it will also be difficult to buy the same disks as the ones in the original configuration, and thus newer disks will be added. In both cases, we will make the array a heterogeneous one because it will be made of disks with different characteristics. This kind of situation is especially common in low-cost clusters of workstations, where cost is an important issue and old components have to be used as well as possible. According to the study performed by Dr. Grochowski at IBM [7], disk capacity nearly doubles every year while the price per Mbyte is decreasing about 40% per year. This means that the price of arrays will remain about the same throughout the years, although the capacity will be increased a lot, of course. If a given site wants to buy a 32 disk array (assuming for example 18GB Seagate disks at today prices), it costs between \$17000 and \$26400 (depending on the interface, RPM, and seek time) [18]. At this price, changing all these disks at a time because one of them breaks is too expensive for many institutions and/or companies, especially if the problem can be solved by just buying a single disk. The only exception appears when the site does not need to grow its capacity and thus replacing the 32-disk array by a few new ones (reducing the size of the array) is enough. Nevertheless, this does not seem to be the trend as disk usage grows constantly.

To handle this kind of disk array, current systems do not take into account the differences between the disks. All disks are treated as if they had same capacity (the smallest one) and performance (the slowest one). This is not the best approach because improvements in both capacity and response time of the heterogeneous array could be achieved if each disk were used accordingly to its characteristics.

In this work, we present a simple solution to this problem by proposing AdaptRaid5, a block-distribution algorithm that improves the performance and effective capacity of heterogeneous disk arrays compared to current solutions. We should note that this proposal has been especially evaluated for scientific and general purpose workloads (under-

*This work has been supported by the Spanish Ministry of Education (CICYT) under the TIC-95-0429 contract.

standing as workload the requests that reach the disk controller, after being filtered by the-file system cache) because the multimedia case has already been addressed quite successfully by other research groups [6, 17, 24]. Nevertheless, the proposed algorithm also works well in a multimedia environment.

This paper is divided into 8 Sections. Section 2 presents the most relevant work in the area of heterogeneous disk arrays. Section 3 introduces the reader to some important concepts that need to be clarified before describing the algorithm, which is explained in full detail in Section 4. Section 5 presents the methodology used to obtain the results presented in Section 6. Section 7 presents the future work we plan to do in this field. Finally, Sections 8 and 9 present the conclusions that can be extracted from this work and how to get more information about this work.

2 Related Work

Some projects have already addressed the same problem, but they have been focused on multimedia systems (and especially video and audio servers). The work done by Santos and Muntz [17] proposed a random distribution with replications to improve the short and long-term load balance. In a similar line, Zimmermann proposed a data placement policy based on the creation of logical disks composed of fractions or combinations of several physical disks [24]. Finally, Dan and Sitaran proposed the usage of fast disks to place "hot" data while the less important data would be located in the slow disks [6]. The main difference from our approach is that all these projects were targeted to multimedia systems while we want a solution for general purpose and scientific environments. Due to their focus on multimedia, they could make some assumptions such as that very large disk blocks (1Mbyte) are used, that reads are much more important than writes and that the main objective is to obtain a sustained bandwidth as opposed to achieving the best possible response time. These assumptions are not valid in our environment where blocks are only a few Kbytes in size, writes are as important as reads, and sustained bandwidth is not as important as the fastest response time. We have to keep in mind that we evaluate the accesses that reaches the disk controller after being filtered by the file-system cache.

The only two works, as far as we know, that deal with this problem in a non-multimedia environment are the HP-AutoRaid [23] and a software RAID that has been implemented in Linux [21]. In the case of the AutoRaid, heterogeneity in the devices is not the objective, but its architecture supports different kind of disks. Nevertheless, in that work only size has been taken into account and no studies to improve performance by using the disks according to their characteristics have been presented. In the software RAID in Linux, any of the disks in the array can be built by putting several disks together. Each disk will store part of the blocks assigned to this *virtual* disk. The problem with this approach is that it is too simple because it only works if you can find a set of disks that match the size of the others in the array (unless you want to waste disk space). Furthermore, it only works for RAIDs level 0, and not for level 5.

Other projects have also dealt with a heterogeneous set of disks, but their objective was to propose new architectures using different disks for different tasks. Along this line we could mention the DCD architecture [10]. In our work, we do not try to decide which is the best hardware and then buy it, we want to deal with already existing devices whichever they are.

The work done by Holland and Gibson in 1992 [9] and by Lee and Katz in 1993 [12] is also related to this project, although not from the heterogeneity point of view. In both studies, ways to handle stripes with smaller striping units than disks in the array are presented. This idea is also used in our work, as will be seen throughout the paper.

Finally, our research group has also proposed a solution to the same problem for disk arrays level 0 [5]. Although it is a much simpler algorithm, because there are no parity problems, many of the ideas presented here have evolved from that first proposal.

3 Preliminary Issues

3.1 Disk Arrays and Parallelism

Disk arrays were especially designed to group several disks into a single address space and to offer high bandwidth by exploiting data access parallelism. Understanding how this parallelism improves the performance of an array is very important to understanding the design and results presented in this paper.

A first kind of parallelism is achieved within a single request. In this case, all disks work together to fulfill a single request and thus the time spent transferring data from the magnetic surface is divided by the number of disks.

A second kind of parallelism occurs when several requests do not use all disks in the array and can be served in parallel. This kind of parallelism makes sense when requests are small compared to the size of the stripe. If requests are large, they will use all disks and the parallelism between requests will decrease significantly.

3.2 Small-Write Problem

One of the most important performance problems in a RAID5 is the small-write problem. In this kind of array, writing data implies that the parity information has to be updated. For this reason, it is recommended to write full stripes as the parity can be computed only using the blocks to be written. If a write operation does not write all the blocks in a stripe, some blocks have to be read from the array to recompute the new parity. This means that a write also implies a read, which penalizes the performance of the operation.

In this work, we consider the read-write-modify approach as opposed to the regenerate-write [3] because it offers more parallelism between requests. The first one (read-write-modify) consists of reading the same blocks that are being written and the parity block. Then, the parity block is XORed with the old blocks (just read) and with the new blocks (just to be written) obtaining the new parity block. The other possibility (regenerate-write) is to read the blocks that are not being modified and thus the new parity blocks can be computed because we have all the blocks in the stripe.

4 AdaptRaid5

4.1 Block-Distribution Algorithm

The best way to understand this algorithm is to describe its evolution starting from the most intuitive, but problematic, version. Then, we discuss the problems we have detected and the solutions we have proposed. To conclude, we present the final version, which should produce a high-performance and high-capacity heterogeneous RAID5.

Intuitive idea

As we have already mentioned, replacing an old disk by a new one or adding new disks to an old array are two common scenarios. In both cases, new disks are usually larger and faster than the old ones [7]. For this reason, we start by assuming that faster disks are also larger, although we will drop this assumption at the end of this section.

The intuitive idea is to place more data blocks in the larger disks than in smaller ones. This makes sense when larger disks are also faster, and thus they can serve more blocks per unit of time. Following this idea, we propose to use all D disks (as in a regular RAID5) for as many stripes as blocks can fit in the smallest disk. Once the smallest disk is full, we use the rest of the disks as if we had a disk array with D-1 disks. This distribution continues until all disks are full with data.

A side effect of this distribution is that the system may have stripes with different lengths. For instance, if the array has D disks where F of them are fast, the array will have stripes with D-1 blocks (plus the parity block), but it will also have stripes with F-1 blocks (plus the parity block). This was not a problem in RAIDs level 0 [5], nevertheless, the effect it may have on a RAID 5 will be discussed later in this paper.

Finally, the parity block for each stripe is placed in the same position it would have been in a regular array with as many disks as blocks in the stripe.

In Figure 1, we present the distribution of blocks in a five-disk array where disks have different capacities. Each block has been labeled with the block number in the array followed by the stripe in which it is

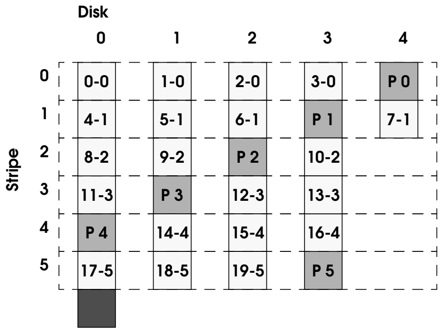

Figure 1: Distribution of data and parity blocks according to the intuitive version.

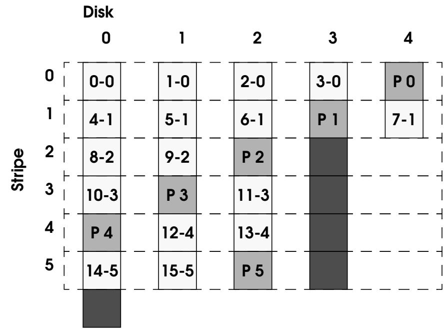

Figure 2: Distribution of data and parity blocks when the stripe size is taken into account.

located (i.e. 8-2 represents data block 8, which is in the strip number 2). Parity blocks are just labeled with a P and the stripe to which they belong. We have to notice that the last block of the largest disk is not used. This happens because stripes must be at least two blocks long, otherwise there is no room to store the parity block for the stripe.

Reducing the small-write problem

As we mentioned in Section 3.2, the file system or controller should organize writes in order to avoid small writes as much as possible [1, 8, 20]. On the other hand, our array has stripes with different sizes and thus if the file system or controller optimizes writes for a given stripe size, it will not be appropriate for stripes with a different size. For instance, if the file system tries to write chunks of 3 blocks (plus the parity one) in a 4-disk stripe, a full stripe will be written. However, if the same chunk is written into a 3-disk stripe, it will perform one full write for two of the data blocks and a small write for the other data block. This means that the performance of a write operation will greatly depend on the stripe it is written to.

The solution to this problem can be approached from two different levels: file system or device. In the first case, the file system has to know that there are different stripe sizes in order to optimize writes accordingly. In the second case, which is the one we propose, the array hides the problem from the file system that assumes a fixed stripe size.

The array can hide the problem of having different stripe sizes by making sure that the number of data blocks in each stripe is a divisor of the number of data blocks in the largest stripe, which we assume is what is being used by the file system. This condition guarantees that full stripes, from the file system point of view, are divided into a set of full stripes, and thus the number of small writes is not increased.

In Figure 2, we present the new distribution for the example in Figure 1. We should notice that the last four blocks in disk 3 become unused. As we have mentioned, the number of data blocks in a stripe has to be a divisor of the data blocks in the largest one. In this example, the largest stripe has 4 data blocks, and thus a stripe with three data blocks is not a valid one. For this reason, stripe number 2 becomes a three-block stripe and all the space in disk 3 that comes after P1 remains unused.

Increasing the effective capacity

The distribution for solving the small write problem above has created a capacity problem in that some blocks must go unused to keep the smaller stripe sizes divisible into the maximum stripe size. For example, the dark blocks in Figure 2 cause the utilization of disk 3 to be only 33%. Thus, the next step is to reclaim our ability to utilize those extra blocks.

We will describe this optimization in two steps. First, we will find a way to use all the available disks without worrying about the capacity. And second, we will use this distribution to increase the effective capacity.

The first problem, then, is how to map stripes that are N-blocks long in a set of D disks ($D > N$) using all the disks. One way to do this mapping is to

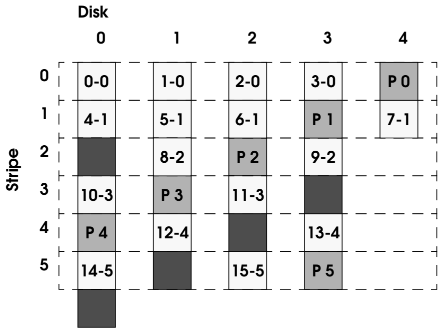

Figure 3: Distribution of stripes, which are 3-blocks long, among four disks.

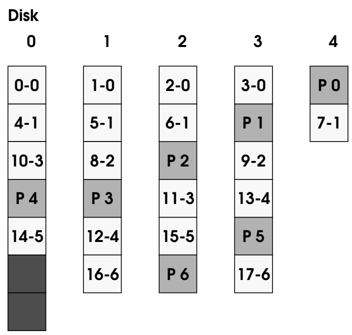

Figure 4: Distribution of stripes, which are 3-blocks long, among four disks filling all empty blocks.

start each stripe in a different disk. For instance, if stripe *i* starts in disk *d*, then stripe *i+1* should start on disk *d-1*. Figure 3 shows an example where stripes that are 3-blocks long (2 data plus 1 parity) are distributed among four disks. Please notice that this only happens for stripes 2 to 5.

The previous step uses all disks, but the number of unused blocks is not reduced at all. To fill these unused blocks we can use a *Tetris*-like algorithm. We can push all blocks so that all empty spaces are filled. Figure 4 presents the previous example once the *pushing* has been done. We can observe now, that all the blocks in the disk are used regardless of the size of the stripe (2 additional data blocks plus one parity block can be accommodated). With this algorithm, we can have stripes with different sizes while all the blocks in all disks are used to store either data or parity information.

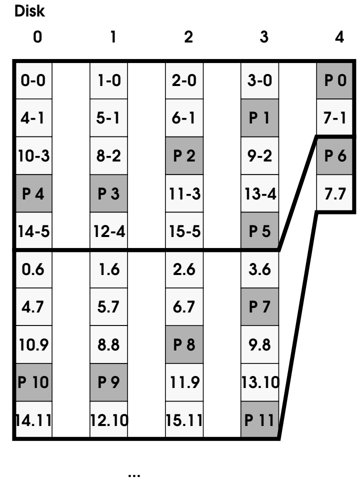

Figure 5: Example of pattern repetition.

Reducing the variance in parallelism

If we apply the algorithm as we have presented it so far, we observe that longer stripes are placed in the lower portion of the address space of the array while the shorter ones appear in the higher portion of the address space. This means that requests that fall in the lower part of the address space can use more disks (longer stripes) while the requests that fall in the higher part of the address space only use a small subset of the disks (shorter stripes).

This can be a problem if our file system tries to place all the blocks of a file together, which is a common practice [13, 14, 19]. This means that a given file may have most of its blocks in the lower part of the address space (long stripes) while another file may have all its blocks in the higher part of the address space (short stripes). Although the global access in the system will be an average, the first file will have a faster access time (more parallelism) while the second one will have a slower access time (less parallelism). For this reason, evenly distributing the location of long and short stripes all over the array will reduce the variance between the accesses in the different portions of the disk array, which we believe is how the storage system should behave.

To make this distribution, we introduce the concept of a pattern of stripes. The algorithm assumes, for a moment, that disks are smaller than they actually are (but with the same proportions in size) and distributes the blocks in this *smaller* array. This distribution becomes the pattern that is repeated until all disks are full. The resulting distribution has the same number of stripes as the previous version of the algorithm. Furthermore, each disk also has the same number of blocks as in the previous version. The only difference is that short and long stripes are distributed all over the array, which was our objective. An example of this pattern repetition can be seen in Figure 5.

With this solution, we can see the pictures presented so far (Figures 1, 2, or 4) as patterns that can be repeated in disks thousands of times larger than the ones presented.

It is also important to notice that the concept of patterns will simplify the algorithm to find a block as will be described later (Section 4.2).

Limiting the size of the pattern

Finally, we want to solve a very focused problem that will only appear in special cases, but that may be important in some cases. Nevertheless, the solution is very simple and has no negative side effects in the rest of cases, making it appropriate to be implemented.

In a regular disk array, all stripes are aligned to a multiple of the number of data blocks in the stripe. We may have systems, or applications, that try to align their full-stripe requests to the beginning of a stripe to avoid making extra read operations. For example, if we have a distribution where the pattern is the one in Figure 4, accessing 4 blocks starting from block 16 should be a full stripe. However, it is not with our new block-distribution algorithm.

Before presenting the solution, we need to define the concept of *reference stripe*. A reference stripe is the stripe that the system or application assumes to be a full stripe. For instance, in the previous example the reference stripe has 4 data blocks and 1 parity block.

The solution to this problem is quite simple. The algorithm only has to make sure that the number of data blocks in a pattern is a multiple of the number of blocks in the reference stripe. This condition guarantees that all repetitions of the pattern start at the beginning of a file system *full stripe*. The result of applying this last step in the example can be seen in Figure 5.

Generalizing the solution

So far, our algorithm has been based on an assumption that the size of disks and their performance grow at the same pace, but this is not usually the case [7]. For this reason, we want to generalize the algorithm in order to make it usable in any environment.

If we examine the algorithm we can see that there are two main ideas that can be parameterized. The first one is the number of blocks we place in each disk. So far, we assumed that all blocks in a disk are to be used. Now, we want to add a parameter to the algorithm that defines the proportion of blocks that are placed in each disk. The **utilization factor (UF)**, which is defined on a per-disk basis, is a number between 0 and 1 that defines the relationship between the number of blocks placed in each disk. The disk with the most blocks always has a UF of 1 and the rest of disks have a UF related to the number of blocks they use compared to the most loaded one. For instance, if a disk has a UF of 0.5, it means that it stores half the number of blocks as compared to the most loaded one. This parameter allows the system administrator to decide the load of the disks. We can set values that reflect the size of the disks, or we can find values that reflect the performance of the disk instead of the capacity.

The second parameter is the number of **stripes in the pattern (SIP)**. The number of stripes in the pattern indicates how well distributed are the different kinds of stripes along the array. Nevertheless, we should keep in mind that smaller disks will participate in less than SIP stripes.

Figure 5 presents a graphic example of how blocks are distributed in the first two repetitions of the pattern if we use the following parameters: $UF_0 = UF_1 = UF_2 = UF_3 = 1$, $UF_4 = 0.4$ and $SIP = 6$. Please note that there are no empty blocks in the picture because we assume much larger disks and the empty blocks would be placed at the end. Remember that the picture only shows the first two repetitions of the pattern.

Fast but small disks: a special case

The current algorithm can be used with any kind of disks. Nevertheless, it does not make much sense if the fastest disk is also significantly smaller. In this case, a better use for these disks would be to keep "hot data" as proposed by Dan and Sitaran [6].

4.2 Computing the Location of a Block

Besides all the aspects already mentioned about performance of disk accesses, we also need to make sure that finding the physical location of a given block can be done efficiently.

This is done in a very simple way. When the system boots, the distribution of blocks in a pattern is computed and kept in three tables. The first one (`location`) contains the disk and position within that disk of any block in the pattern. The second one (`parity`) keeps the location of the parity block for each stripe. Finally, the third table (`Blks_per_disk_in_pattern`) stores the number of blocks each disk has in a pattern. These tables should not be too large. In our experiments the `position` table has 152 entries, the `parity` one only has 19 entries, and the `Blks_per_disk_in_pattern` has 9 entries. These sizes can be assumed by any RAID controller or file system. The formulas to compute the location of a given block (B) follow:

$$\mathbf{disk}(\mathbf{B}) = location[B\%Blks\_in\_a\_pattern].disk$$
$$\mathbf{pos}(\mathbf{B}) = location[B\%Blks\_in\_a\_pattern].pos + (B/Blks\_in\_a\_pattern) * Blks\_per\_disk\_in\_pattern[disk(B)]$$

As these operations are very simple, the algorithm to locate blocks is very fast. To check this time, we profiled the simulator and we found that the time spent in deciding the location of blocks was less than 81μs in average per request[1], which is insignificant compared to the time of a disk access.

5 Methodology

5.1 Simulation and Environment Issues

In order to perform this work, we have used HRaid [4], which is a storage-system simulator[2]. The simulator has been validated using the HP-92 suit of traces [15, 16] and also comparing the results of many tests to the ones obtained by D. Kotz's simulator [11], which is also a validated simulator.

All tests presented in this paper were performed simulating an array with a combination of slow and fast disks. The model used for these disks is the one proposed by Ruemmler and Wilkes [16]. The parameters used for the slow disks were taken from the Seagate Barracuda 4LP [18] and to emulate the fast disk we used the parameters of a Cheetah 4LP [18], which is also a Seagate disk. A list with some important characteristics for each disk (controller and drive) are presented in Table 1. Finally, the size used for the striping unit is 128Kbytes. This size has been computed using the ideas presented by Chen et al. [2]. Although the formulas presented in that paper were for homogeneous disk arrays, we have assumed they would be adequate for heterogeneous ones.

Table 1: Disk characteristics.

	Fast Disk	**Slow Disk**
Size		
Disk size	4.339 Gb	2.061 Gb
Cache size	512Kbytes	128Kbytes
Sector size	512Bytes	512Bytes
Cache model		
Read/Write fence	64Kbytes	64Kbytes
Prefetching	YES	YES
Immediate report	YES	YES
Overheads		
New-command	1100μs	1100μs
Track switch	800μs	800μs
Bandwidth		
RPM	10033	7200
Seek model		
Limit (in cylinders)	600	600
Sort: a+b*sqrt(d) μs	a = 1.55	a = 3.0
	b = 0.155134	b = 0.232702
Long: a+b*(d) μs	a = 4.2458	a = 7.2814
	b = 0.001740	b = 0.002364

These disks and the hosts were connected through a Gigabit network (10μs latency and 1Gbits/s bandwidth). We simulated the contention of the network, but no protocol overhead was simulated.

We also have to keep in mind that in the simulations we only took the network and disks (controller and drive) into account. The possible overhead of the requesting hosts was not simulated because it greatly depends on the implementation of the file system. The only issue we simulated from the file system was that it can only handle 10 requests at a time. The rest of requests wait in a queue until one of the previous requests has been served.

[1]Times taken in a SGI2000

[2]http://www.ac.upc.es/homes/toni/software.html

Finally, we have to mention that when using the synthetic traces presented in the next section, we made 10 runs for each one of them (all with different seed to generate the access pattern) and report the average value. In these runs we always obtained very similar results and the difference was never larger than 2%.

5.2 Workload issues

In order to get the first results, we have studied the behavior of the system on a set of synthetic workloads based on the following parameters:

- **Kind of request**: whether requests were reads or writes.
- **Request size**: the size of all the requests in the load.
- **Request alignment**: the position of the requests is always chosen randomly, but the start of the request can be either aligned to a block in the first disk (to avoid small writes) or not.

Table 2 presents the characteristics of the synthetic workloads used.

Table 2: Synthetic-workload characteristics.

	Request Size	**Aligned**	**Operation Type**
W8	8Kbytes	No	Writes
W256	245Kbytes	No	Writes
W1024	1024Kbytes	Yes	Writes
W2048	2048Kbytes	Yes	Writes
R8	8Kbytes	No	Reads
R2048	2048Kbytes	Yes	Reads

On the other hand, we also wanted to obtain results for a real system, and thus we used a portion of the traces gathered by the Storage System Group at the HP Laboratories (Palo Alto) in 1999 [22]. These traces represent a detailed characterization of every low-level disk access generated in the system over a 6 month period. This system contained a file server and some workstations used by the people in the Storage System Group to perform their work (compilations, edition, databases, simulations, etc.). As the size of the traces was too large (6 months) we will only present the results obtained during the busiest hour of February 14th. The tested portion has 159208 reads and 115044 writes and the average request size is around 12 Kbytes. With these traces, as with most traces, dependencies such as that a given operation has to follow another one are not recorded. However, this does not invalidate the results presented because the general load they represent continues to be real.

5.3 Configurations Studied

All the experiments presented in this paper have been done using disk arrays with 9 disks. This number of disks is large enough to see the possible advantage and limitations of the proposal. Furthermore, it is small enough to make things easy to understand.

Another important issue is the way small writes are handled. All the arrays we have evaluated used the read-write-modify algorithm, which means that the blocks read in a small write are the same ones as the blocks written [3]. This option has been used because it increases the parallelism between requests.

For simplicity, the configurations used always have all fast disks in the first positions and the slow ones in the last position of the array.

Finally, we have chosen a single SIP of 19 for all experiments, also for simplicity reasons. Regarding the utilization factors we have used a UF of 1 for the fast disks and .46 for the slow disks. These values have been decided experimentally and a sensitivity analysis for this parameter is presented in Section 6.6. We know that better values could be used for some of the experiments, but this is not the important issue as we want to prove the goodness of the idea and not to propose the best possible parameters.

5.4 Reference Systems

We have compared AdaptRaid5 with the following two base configurations:

- **RAID5:** This is the traditional RAID5 algorithm and it uses all the disks (fast and slow). It is important to notice that this leads to fast disks being treated as if they were slow ones

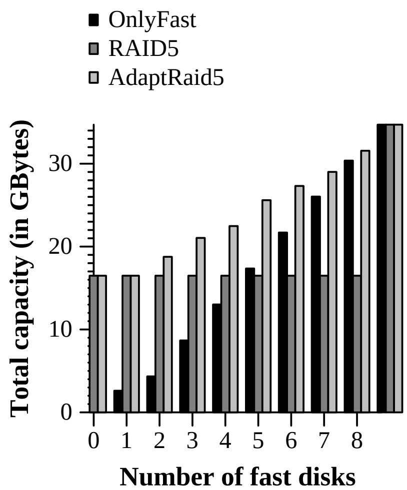

Figure 6: Effective capacity for the studied configurations.

and that only a portion of their capacity is effectively used.

- **OnlyFast:** This is also a traditional RAID5, but only using fast disks. The number of fast disks will be the same as the number of fast disks in the heterogeneous configuration. This comparison will give us the idea of whether it is better to throw the old disks away instead of using them.

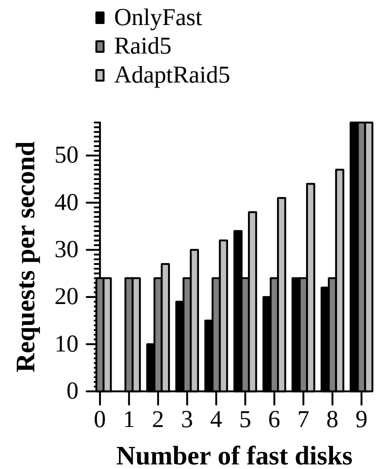

Figure 7: Writing 1024Kbytes blocks (W1024).

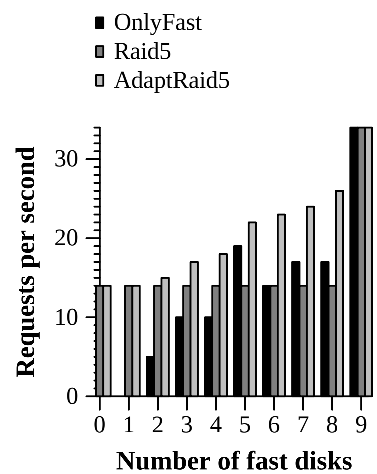

Figure 8: Writing 2048Kbytes blocks (W2048).

6 Experimental Results

6.1 Capacity Evaluation

We present a graph (Figure 6) of the effective capacity based only on data blocks as we vary the number of fast disks out of the total of 9 disks for each distribution algorithm used. We can see that AdaptRaid5 is the one that obtains the largest capacity. This happens because it knows how to take advantage of the capacity of all disks in the array. Furthermore, we can see that the extra number of parity blocks used by our proposal does not affect the effective capacity significantly.

6.2 Full-Write Performance

The performance obtained by a RAID5 when a full stripe is written is one of the important results for this kind of array. For this reason, we start evaluating the case where a write operation does not imply a previous read. To study this performance, we have measured the number of requests per second each of the evaluated systems can handle when requests are 1024Kbytes and 2048Kbytes long (workloads W1024 and W2048 described in Section 5.2). Although these may seem to be very large requests for the target environment, it is the only way to test full writes. Controllers or file systems may use logging and achieve such request sizes in non multimedia environments. Figures 7 and 8 present these results.

If we concentrate our attention on each of the systems individually, we can see that RAID5 does not change its performance when more of the disks are fast. This happens because this algorithm does not know how to use the better performance of newer disks.

The second system, OnlyFast, has a very inconsistent behavior. It can achieve high performance under some configurations and a very bad one under others. The reason behind this behavior is the increase in the number of small writes. As we have mentioned in Section 4.1, if the number of data disks used is not a divisor of number data blocks in a stripe, a full-stripe write operation ends up performing a small write. This scenario occurs when the system has 4, 6, 7 and 8 disks. In the rest of the configurations, the performance obtained by OnlyFast is quite good and proportional to the number of fast disks. We should notice that this system has not been evaluated for 0 or 1 fast disks because we need at least 2 disks to build a RAID5.

The last evaluated system is our proposal (AdaptRaid5). We can observe that the performance of this system increases at a similar pace as the number of fast disks used, which was our objective.

If we compare the behavior of traditional RAID5 with our proposal, we can see that AdaptRaid5 always achieves a much better performance. This happens because AdaptRaid5 knows how to take advantage of fast disks while RAID5 does not. The only exception to this rule appears when only 0 or 1 fast disks are used. In this case, AdaptRaid5 cannot use the fast disks in any special way.

The comparison between AdaptRaid5 and OnlyFast also shows that our proposal is a better one. On the one hand, AdaptRaid5 is much more consistent than OnlyFast and it does not present a bad performance in any of the configurations. On the other hand, our system always obtains a better performance than OnlyFast. AdaptRaid5 is faster because it takes advantage of the parallelism within a request (it has more disks), which is very important when only a few fast disks are available or when requests are large. Furthermore, when OnlyFast starts to take advantage of the parallelism (when more fast disks are used), AdaptRaid5 starts to use the slow disks less frequently, which out-weighs the improvements of OnlyFast.

6.3 Small-Write Performance

The other possibility for a write operation is to perform a small write. In this case, some blocks have to be read in order to compute the parity of the stripe. This situation is different from the previous one, besides introducing the issue of the extra reads, because requests do not use all disks and this increases the parallelism between requests. This extra parallelism can be important in configurations with few fast disks because this parallelism will not be exploited by AdaptRaid5 and OnlyFast when only fasts disks are used, while it will be exploited by RAID5 that always uses all disks.

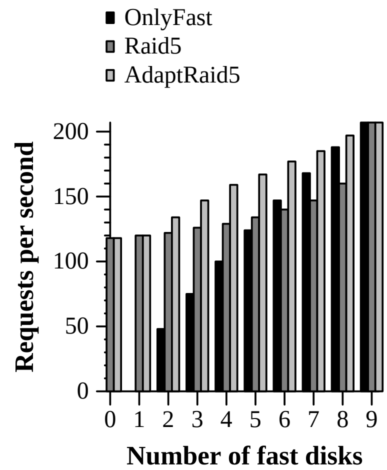

Figure 9: Writing 8Kbytes blocks (W8).

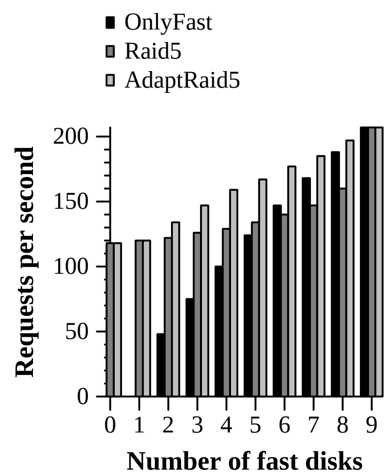

Figure 10: Writing 256Kbytes blocks (W256).

To do this evaluation we have measured the number of requests per second achieved by each evaluated system when 8Kbytes and 256Kbytes requests are done (workloads W8 and W256 described in Section 5.2) (Figures 9 and 10).

In this case, AdaptRaid5 is also better than RAID5, for the same reason as before. It knows how to use

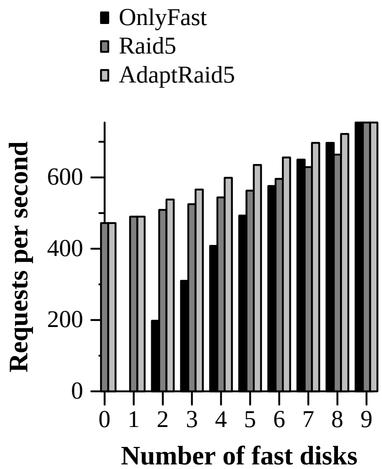

Figure 11: Reading 8Kbytes blocks (R8).

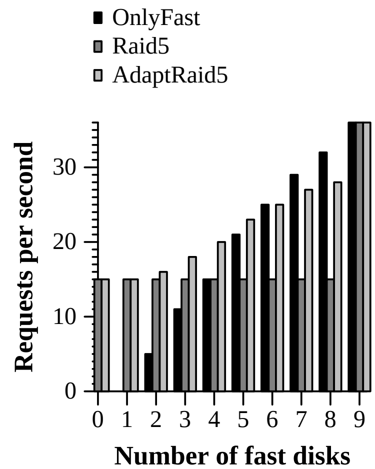

Figure 12: Reading 2048Kbytes blocks (R2048).

the fast disks. Furthermore, we can also see that the extra parallelism RAID5 can exploit is not enough compared to the benefit of only using fast disks for many of the requests.

When we compare AdaptRaid5 with OnlyFast, we observe that our proposal has a better performance than OnlyFast. This happens because AdaptRaid5 can use more disks and it can take advantage of the parallelism between requests.

6.4 Read Performance

Once the write performance has been evaluated, we need to measure the performance obtained by read operations. This evaluation has been done measuring the number of requests per second obtained by the system when requesting read operations 8Kbytes, and 2048Kbytes long (workloads R8 and R2048 described in Section 5.2). These results are presented in Figures 11 and 12.

In the first case (Figure 11), where requests are 8Kbytes, we observe a very similar behavior as in the previous cases. The only difference is that the performance of RAID5 and OnlyFast gets closer to AdaptRaid5 than in previous experiments. This happens because on these read operations, only one disk is used per request and more parallelism between requests can be achieved by OnlyFast and the probability of using a slow disk decreases in RAID5.

In the second case (Figure 12), the requests are much larger and this has two effects. If we observe RAID5 performance, it remains unmodified when more fast disks are added. This is because all disks are used in the request and thus, slow disks are always included. If we focus on OnlyFast, we can see that it outperforms AdaptRaid5 when more than 6 fast disks are used. This happens because when these many fast disks are used, OnlyFast has enough parallelism within a request to obtain a good performance. On the other hand, AdaptRaid5 has to handle slow disks in many of the requests slowing down its performance. This means that if enough fast disks are used and only large reads are to be done, AdaptRaid5 is not the best solution.

6.5 Real-Workload Performance

The last experiment consists of running the trace file from HP described in Section 5.2. These results are presented in Figure 13. In this graph, we present the performance gains (in %) obtained by our distribution algorithm when compared to RAID5 and OnlyFast. The graph is divided in two parts. The left part shows the gain for read operations and the right part presents the results for write operations.

As expected, our algorithm is significantly faster than the other ones tested. The reasons are the same ones we have been discussing so far. The only exception is when 8 fast disks are used. In this case, OnlyFast is faster as it can achieve enough parallelism between requests and no slow disks are ever used. Nevertheless, maintaining only one slow disk does not seem to be very reasonable, and in this case we would recommend to discard the old disk

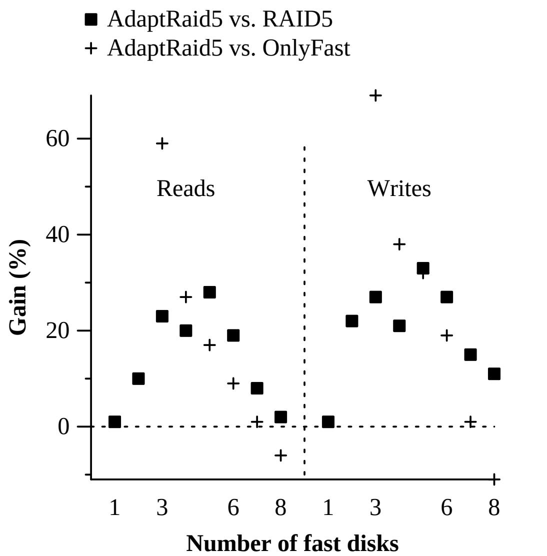

Figure 13: Performance gain of AdaptRaid5 over the rest of configurations in a real workload.

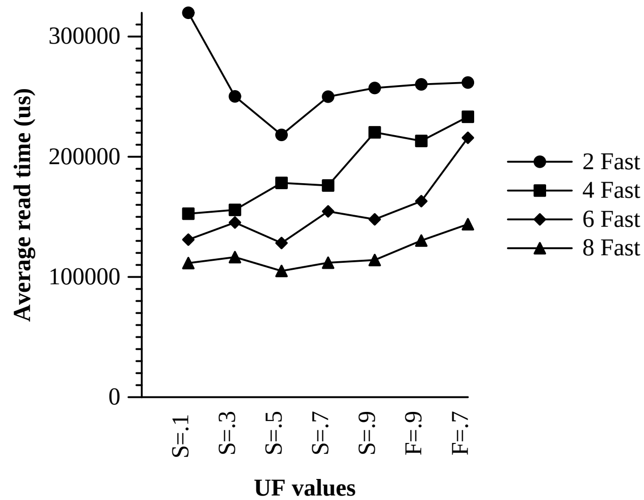

Figure 14: Variation in the average READ time when changing the UF factors for different disk configurations.

(unless the capacity is needed.)

6.6 Sensitivity Analysis of the UF Parameter

In all the experiments run so far, we have used UF values that maximize the utilization of the disks as far as capacity is concerned. Now, we want to see how sensitive is the performance of the array to the different values of UF. For this reason, we have tested the HP99 workload varying the UF factors on different array configurations. The different array configurations have 9 disks, but the number of fast disks used varies. These different configurations are represented by the different curves presented in Figures 14 and 15. The combinations of UF values used range, on the one hand, from $UF_{fast} = 1$ and $UF_{slow} = .1$ to both $UF_{fast} = UF_{slow} = 1$. These tests have been marked in Figures 14 and 15 using the name $S = .X$, which represents the value of UF_{slow} because UF_{fast} remains 1 all the time. On the other hand, we have also tried a couple of configurations where the slow disks have higher UF values. In these two tests, $UF_{slow} = 1$ and UF_{fast} takes .9 and .8 as values. These experiments are marked using the name $F = .X$, which represents the value of UF_{fast} because UF_{slow} remains 1 all the time

Figure 14 presents the average read times obtained in these experiments. The first thing we can observe is that, in general, the more the slow disks are used, the longer it takes to perform read operations. The exception to this rule appears when only a few fast disks are used. In this case, the higher speed of fast disks cannot outweigh the parallelism obtained by the larger number of slow ones and the best read access time is achieved when slow disks are used half the time the fast ones.

It is also important to notice, that the curves are not perfect because there are other parameters that also have their effect in the performance. Changing the UF values also changes the placement of data and parity blocks, which also has an effect in performance.

Figure 15 presents the results of the same experiments, but for the average write time. In this figure we can observe the same behavior as with read operations.

Summarizing, the election of UF values is especially important if the number of fast disks is small and the higher performance of the fast disks cannot outweigh the parallelism of the large number of slow disks. Otherwise, using the fast disks as much as possible seems to be the way to go. Nevertheless, this election should also take capacity into account because different UF values achieve arrays with different capacities.

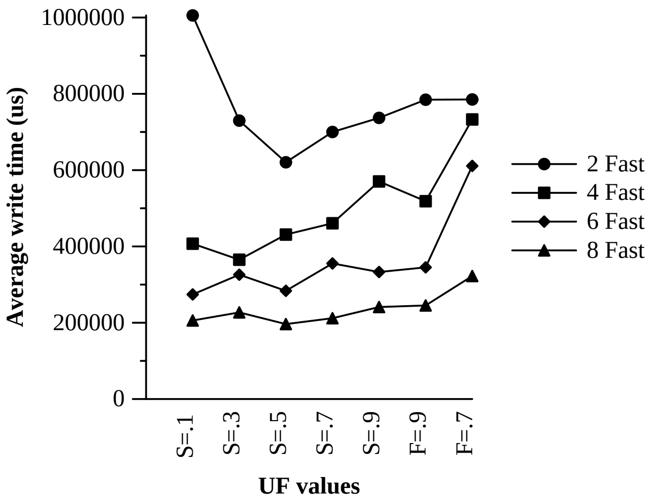

Figure 15: Variation in the average WRITE time when changing the UF factors for different disk configurations.

7 Future Work

In the future, we plan to concentrate our work on these issues:

- Find the best block size for this kind of algorithm as was done for regular disk arrays by Chen [2].
- Implement the algorithm in the Linux kernel.
- Study mechanisms to allow adding/replacing disks while the array is on-line.

8 Conclusions

In this paper, we have presented AdaptRaid5, a block-distribution policy that takes full advantage of heterogeneous disk arrays.

This algorithm achieves a significant performance compared to the policies currently being used. We have proven this by using both synthetic and real loads.

Furthermore, we have also shown that arrays using AdaptRaid5 are able to serve many more disk requests per second than when blocks are distributed assuming that all disks have the lowest common speed, which is the solution currently being used.

Finally, we have to keep in mind that this algorithm has been evaluated for an array built from disks attached to a SAN, but it would also work in other array configurations.

9 Availability

Other papers and reports about heterogeneous disk arrays

- `people.ac.upc.es/toni/papers.html`

Simulator information and downloading

- `people.ac.upc.es/toni/software.html`

A simplified version of the algorithm in pseudo code

- `people.ac.upc.es/toni/AdaptRaid/pcAR5.html`

Acknowledgments

We thank the Storage System Group at HP Laboratories (Palo Alto), and especially to John Wilkes, for letting us use their 1999 disk traces and for their interesting comments. We are also grateful to all the anonymous referees whose comments helped us to improve the quality of this paper. Finally, we thank Carla Ellis, who has been our shepherd and has shown us many ways to improve the quality of this work.

References

[1] Anderson, T. E., Dahlin, M. D., Neefe, J. M., Patterson, D. A., Roselli, D. S., and Wang, R. Y. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles* (December 1995), pp. 109–126.

[2] Chen, P., and Lee, E. K. Striping in a RAID level 5 disk array. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May 1995), pp. 136–145.

[3] Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A. RAID: High-performance and reliable secondary storage. *ACM Computing Surveys 26*, 2 (1994), 145–185.

[4] Cortes, T., and Labarta, J. HRaid: A flexible storage-system simulator. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications* (June 1999), CSREA Press, pp. 772–778.

[5] Cortes, T., and labarta, J. A case for heterogenenous disk arrays. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'2000)* (November 2000), pp. 319–325.

[6] Dan, A., and Sitaram, D. An online video placement policy based on bandwidth to space ratio (bsr). In *Proceedings of the SIGMOD* (1995), pp. 376–385.

[7] Grochowski, E., and Hoyt, R. F. Future trends in hard disk drives. *IEEE Transactions on Magnetics 32*, 3 (May 1996).

[8] Hartman, J., and Ousterhout, J. K. The zebra striped network file system. *Transactions on Computer System 13*, 3 (1995), 274–310.

[9] Holland, M., and Gibson, G. A. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of the 5th Conference on Architectural Support for Programming Languages and Operating Systems* (Octouber 1992), pp. 23–35.

[10] Hu, Y., and Yang, Q. A new hierarchical disk architecture. *IEEE Micro* (November/December 1998), 64–75.

[11] Kotz, D., Toh, S. B., and Radhakrishnan, S. A detailed simulation model of the HP-97560 disk drive. Tech. Rep. PCS-TR94-220, Department of Computer Science, Dartmouth College, July 1994.

[12] Lee, E. K., and Katz, R. H. The performance of parity placements in disk arrays. *IEEE Transactions on Computers 42*, 6 (June 1993), 651–664.

[13] McKusick, M., Joy, W., Leffler, S., and Fabry, R. A fast file system for unix. *ACM Transactions on Computer Systems 2*, 3 (August 1984), 181–197.

[14] McVoy, L., and Kleiman, S. Extent-like performance from a unix file system. In *Proceedings of the Summer Technical Conference* (June 1990), USENIX Association, pp. 137–144.

[15] Ruemmler, C., and Wilkes, J. Unix disk access patterns. In *Proceedings of the Winter USENIX Conference* (January 1993), pp. 405–420.

[16] Ruemmler, C., and Wilkes, J. An introducction to disk drive modeling. *IEEE COMPUTER* (March 1994), 17–28.

[17] Santos, J. R., and Muntz, R. Performance analysis of the rio multimedia storage system with heterogenenous disk configurations. *ACM Multimedia* (1998), 303–308.

[18] Seagate. Segate web page. http://www.seagete.com, January 2000.

[19] Smith, K. A., and Seltzer, M. A comparison of ffs disk allocation policies. In *Proceedings of the Annual Technical Conference* (January 1996), USENIX Association.

[20] Stodolsky, D., Gibson, G., and Holland, M. Parity logging overcoming the small write problem in redundant disk arrays. In *Proceedings of the 21th Annual International Symposium on Computer Architecture* (1993), pp. 64–75.

[21] Vepstas, L. Software-raid howto. http://www.linux.org/help/ldp/howto/Software-RAID-HOWTO.html, 1998.

[22] Wilkes, J. Personal communication, Septemeber 1999.

[23] Wilkes, J., Golding, R., Staelin, C., and Sullivan, T. The HP AutoRAID hierarchical storage system. In *Proceedings of the 15th Operating System Review* (December 1995), ACM Press, pp. 96–108.

[24] Zimmermann, R. *Continuous media placement and scheduling in heterogeneous disk storage systems.* PhD thesis, University of Southern California, December 1998.