Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE @ DIRECT•

J. Parallel Distrib. Comput. 63 (2003) 448–464

Journal of Parallel and Distributed Computing

http://www.elsevier.com/locate/jpdc

Taking advantage of heterogeneity in disk arrays[☆]

T. Cortes[*] and J. Labarta

Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Campus Nord, C6-E202, C/Jordi Girona, 1-3, ES-08034 Barcelona, Spain

Received 22 January 2002; revised 27 May 2002; accepted 3 March 2003

Abstract

Disk arrays, or RAIDs, have become the solution to increase the capacity and bandwidth of most storage system, 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 two block-distribution algorithms (one for RAID0 and an extension for RAID5) 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.

Keywords: RAID; Heterogeneity; AdaptRaid; Block distribution

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 18 GB Seagate disks at today prices), it costs between \$17,000 and \$26,400 (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

[☆]This work has been supported by the Spanish Ministry of Education (CICYT) under the TIC98-0511 contract.

[*]Corresponding author.

E-mail addresses: toni@ac.upc.es (T. Cortes), jesus@ac.upc.es (J. Labarta).

URL: http://people.ac.upc.es/toni/AdaptRaid.html.

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 AdaptRaid0 and AdaptRaid5, two block-distribution algorithms that improve the performance and effective capacity of heterogeneous disk arrays compared to current solutions. We should note that these proposals have been especially evaluated for scientific and general purpose workloads (understanding 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, there is no reason to believe that the proposed algorithms will have any problems in a multimedia environment.

This paper is divided into 7 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 algorithms, which are explained in full detail in Section 4. Section 5 presents the methodology used to obtain the results presented in Section 6. Finally, Sections 7 and 8 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 (1 Mbyte) 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 reach 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.

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 [3]. 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.

3.3. Target environment

The methods presented in this paper have been designed to work on any kind of disk arrays (hardware or software, tightly or loosely coupled, etc.), although we only evaluate the behavior in an array made by network-attached disks in a storage-area network (SAN), which is a very promising state-of-the-art approach. Furthermore, low-cost clusters of workstations seem to be the most adequate target for the proposed algorithms. Nevertheless, we have to keep in mind that it is not restricted to this architecture.

The mechanisms proposed in this paper can be applied to arrays of any size. Even in the case of building large RAIDs level 0 where each component is a RAID level 5, our proposals can be used. This approach could also be used for each of the two levels in the hierarchy of the HP AutoRaid.

Finally, and as we have already mentioned, we focus our attention on scientific and general purpose applications because multimedia environments, and their special assumptions, have already been addressed [6,17,24].

4. Block-distribution algorithms

The best way to understand these algorithms is to describe their 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 disk array.

4.1. AdaptRaid0

4.1.1. 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 assuming that faster disks are also larger, although we will drop this assumption at the end of this section. Until we drop this assumption, whenever we refer to a fast disk we will also refer to large disks.

The intuitive idea is to place more data blocks in the larger disks than in smaller ones. This makes sense because, as we have assumed, 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 RAID0) 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 blocks, but it will also have stripes with F blocks. Nevertheless, we will show that this effect is not a problem.

In Fig. 1, we present the distribution of blocks in a five-disk array where disks have different capacities (disk 0 has 7 blocks, disks 1–3 have 5 blocks, and disk 4 has only one block). Each block has been labeled with the block number in the array followed by the stripe in which it is located (i.e. 10–2 represents data block 10, which is in the strip number 2).

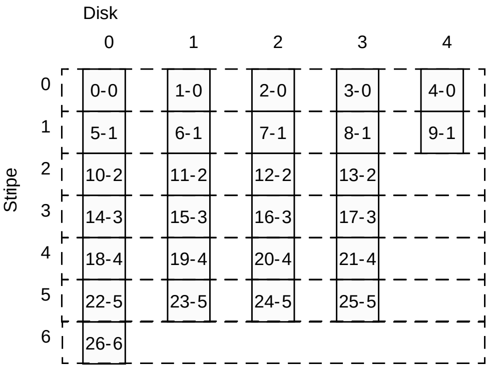

Fig. 1. Distribution of data blocks according to the intuitive version for AdaptRaid0.

4.1.2. *Reducing the variance in the 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 Fig. 2.

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 4.4).

4.2. *AdaptRaid5*

4.2.1. *Intuitive idea*

Like in the case for RAID0, we start by assuming that faster disks are also larger, although we will drop this assumption later on in this paper. And, like for RAID0 our algorithm will place more data blocks in the larger disks than in smaller ones. The only difference is that stripes will have one less data block because we need one block to keep the parity for each stripe. For instance, in a stripe with D disks, D-1 will be data blocks and 1 will contain the parity information.

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.

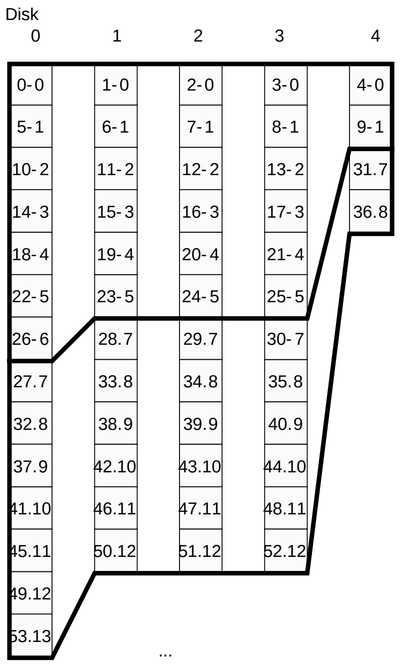

Fig. 2. Example of pattern repetition for AdaptRaid0.

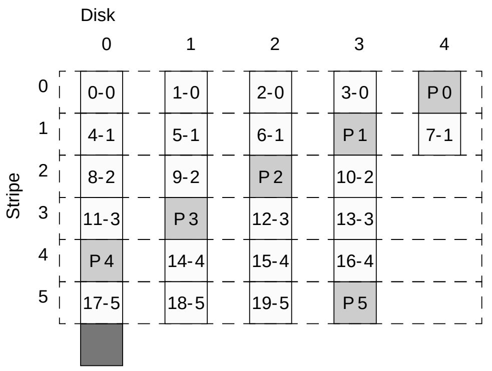

Fig. 3. Distribution of data and parity blocks according to the intuitive version for AdaptRaid5.

In Fig. 3, 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 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.

4.2.2. *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.

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 3 data blocks and 1 parity block.

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 *reference stripe*, which we assume for practical reasons to have one block in all disks. This condition guarantees that full stripes, from the file system point of view, are divided into a set of full stripes from the array point of view, and thus the number of small writes is not increased.

In Fig. 4, we present the new distribution for the example in Fig. 3. 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 *reference stripe*. In this example, the *reference 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.

4.2.3. *Increasing the effective capacity*

The above distribution, proposed to solve the small-write problem, has created a capacity problem in that some blocks must go unused to keep the smaller stripe sizes divisible into the size for the *reference stripe*. For example, the dark blocks in Fig. 4 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 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. Fig. 5 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 2–5.

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. Fig. 6 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.

4.2.4. *Reducing the variance in parallelism*

This is not a new problem as we ran into it in the AdaptRaid0 algorithm. The solution then, will be the same one used in the previous case. We will define a

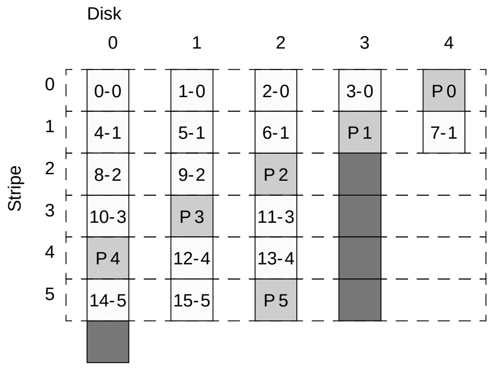

Fig. 4. Distribution of data and parity blocks when the stripe size is taken into account for AdaptRaid5.

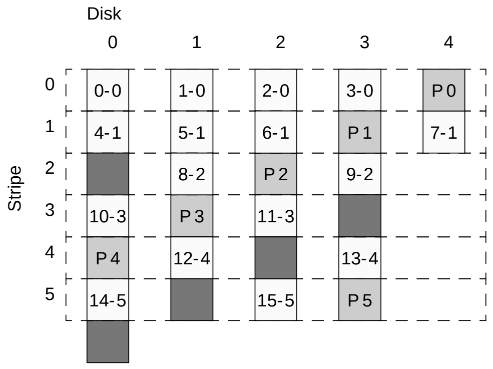

Fig. 5. Distribution of stripes, which are 3-blocks long, among four disks for AdaptRaid5.

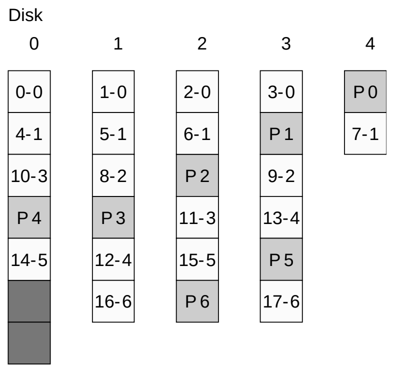

Fig. 6. Distribution of stripes, which are 3-blocks long, among four disks filling all empty blocks for AdaptRaid5.

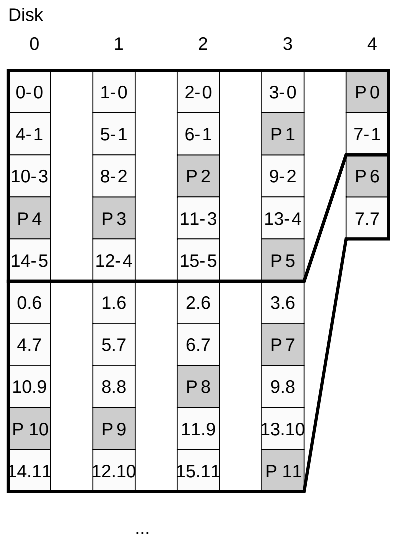

Fig. 7. Example of pattern repetition for AdaptRaid5.

pattern with both, long and short, stripes and this pattern is repeated as many times as needed to fill the disks. An example of this pattern repetition can be seen in Fig. 7.

With this solution, we can see the pictures presented so far (Figs. 3, 4, or 6) as patterns that can be repeated in disks thousands of times larger than the ones presented.

4.2.5. 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 situations. 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 Fig. 6, accessing 4 blocks starting from block 16 should be a full stripe. However, it is not with our new block-distribution algorithm.

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* (already presented when reducing the small-write problem). 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 Fig. 7.

4.3. Generalizing the solutions

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

If we examine the algorithms, we can see that there are two main ideas that can be parameterized. The first one is the number of blocks we use from 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 used 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 used in each disk. The disk that uses the higher number of 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. It is important to notice that a UF of 1 means that this disk will be the one where more blocks are used, but it will not be necessarily the large one. We may have a very large disk and only use a few of its blocks (for whatever reason), and thus this disk will have a low value for its UF.

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.

Figs. 2 and 7 present a graphic example of how blocks are distributed in the first two repetitions of the pattern for the two algorithms presented in this paper. In Fig. 2 (AdaptRaid0) we have used the following parameters: $UF_0 = 1$, $UF_1 = UF_2 = UF_3 = 0.86$, $UF_4 = 0.29$ and $SIP = 7$. In Fig. 7 (AdaptRaid5) we have used the following parameters: $UF_0 = 1$, $UF_1 = UF_2 = UF_3 = 0.86$, $UF_4 = 0.29$ and $SIP = 6$. Please note that there are no empty blocks in the pictures because we assume much larger disks and the empty blocks would be placed at the end, if any. Remember that the picture only shows the first two repetitions of the pattern.

It is also important to notice that the placement of the disks is not really important performance wise. We have always used the largest disks first and the smaller ones last (both in the examples and the experiments), but any other mixture could be used. The only important difference appears in AdaptRaid5. In this case, some configurations may load some disks with more parity blocks than others. If we want to avoid this unbalance, our ordering guarantees that these parity blocks are quite evenly distributed according to the expected load of each disk.

4.3.1. 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].

4.4. 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.

4.4.1. AdaptRaid0

This is done in a very simple way. When the system boots, the distribution of blocks in a pattern is computed and kept in two tables. The first one (`location`) contains the disk and position within that disk of any block in the pattern. The second 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 between 83 to 161 entries (depending on the number of fast disks), 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:

$$\mathbf{disk}(\mathbf{B}) = location[B\%Blks\_in\_a\_pattern].disk,$$

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

In the first formula we compute the disk where block B 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 B divided by the blocks in the pattern (`Blks_in_a_pattern`). Then we can use this value to know the disk where the block is. We have this function computed in advance in table `location`.

Now, we have to find the position of block B within the just computed disk. First, in the same way we computed the disk, we can compute the position of this block in the pattern (first part of the formula). Then, we add the number of blocks in this disk for each pattern repetition. This number of blocks is computed by multiplying the number of times the pattern has been repeated $((B/Blks\_in\_a\_pattern))$ times the number of blocks this disk has in a pattern $(Blks\_per\_disk\_in\_pattern[disk(B)])$.

4.4.2. AdaptRaid5

In the case of AdaptRaid5, two new tables are needed: `stripe` and `parity`. The first table keeps the stripe for each block in the pattern, and has as many entries as blocks in the pattern. The second table keeps the location of the parity block for each stripe the pattern, and in our case it only has 19 entries. The formulas to compute the location of a given block (B) are the same like in AdaptRaid0 and the formula to compute the location of the parity block in stripe ($\mathbf{S}$) follows:

$$\mathbf{S} = stripe[B\%Blks\_in\_a\_pattern] + (B/Blks\_in\_a\_pattern) * SIP,$$

$$\mathbf{disk}(\mathbf{parity\ of\ S}) = parity[S\%SIP].disk,$$

$$\mathbf{pos}(\mathbf{parity\ of\ S}) = parity[S\%SIP].pos + (S/SIP) * Blks\_per\_disk\_in\_pattern[disk(parity\ of\ S)].$$

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 81 μs in average per request (times taken in a SGI2000), which is insignificant compared to the time of a disk access.

4.5. Adding and replacing disks

In any kind of disk array we have to worry about how a disk is added or replaced. Adding new disks always implies a costly operation of remapping data. This will also happen in our case. The only exception is AutoRaid, and as we said we can build an AutoRaid using our mechanisms.

In the case of AdaptRaid0, the behavior is like in a regular RAID0. In this case, if a disk fails the information in the disk is lost because no redundancy is kept. When a new disk has to be added, the only possibility is to backup the information, add the new disks, reconfigure the array, and finally, restore the backup. This is exactly the same steps we have to do with AdaptRaid0.

In the case of AdaptRaid5, replacing a disk can usually be done while the array is being used. In our case, if the systems needs to run 7 days a week 24 h a day, we can place in the new disk the same blocks as in the older one. This will allow the system to function without the benefits of the new disk. Then, we can do the remapping little by little during the normal operation of the array. The best way to do this operation is still being under study.

5. Methodology

5.1. Simulation and environment issues

In order to perform this work, we have used HRaid [5], which is a storage-system simulator. 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 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 128 kbytes. This size has been computed using the ideas presented by Chen et al. [2,4]. Although the formulas presented in that paper were for homogeneous disk arrays, we have assumed they would be adequate for heterogeneous ones.

These disks and the hosts were connected through a Gigabit network (10 μs latency and 1 Gbits/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 (that can be of any size requested by the application) at a time. The rest of requests wait in a queue until one of the previous requests has been served.

Finally, we have to mention that when using the synthetic traces presented in the next section. In order to be able to run several simulations for each experiment, we randomly generated 10 different synthetic traces for a given set of parameters, each using a different random seed. The we have simulated all of them, and we have reported the average value. In these runs we always obtained very similar results and the difference was never larger than 2%.

Table 1
Disk characteristics

	Fast disk (Cheetah 4LP)	Slow disk (Barracuda 4LP)
Size		
Disk size	4.339 GB	2.061 GB
Cache size	512 kbytes	128 kbytes
Sector size	512 bytes	512 bytes
Cache model		
Read/write fence	64 kbytes	64 kbytes
Prefetching	Yes	Yes
Immediate report	Yes	Yes
Overheads		
New command	1100 μs	1100 μs
Track switch	800 μs	800 μs
Bandwidth		
RPM	10,033	7200
Mbytes/s	13.628 MB/s	9.120 MB/s
Seek model		
Limit (in cylinders)	600	600
Short: $a + b \times \text{sqrt}(d)$ ms	$a = 1.55$	$a = 3.0$
	$b = 0.155134$	$b = 0.232702$
Long: $a + b \times (d)$ ms	$a = 4.2458$	$a = 7.2814$
	$b = 0.001740$	$b = 0.002364$

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.

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

Table 2
Synthetic-workload characteristics

	Request size (kbytes)	Aligned	Operation type
W8	8	No	Writes
W256	256	No	Writes
W1024	1024	Yes	Writes
W2048	2048	Yes	Writes
R8	8	No	Reads
R2048	2048	Yes	Reads

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 159,208 reads and 115,044 writes and the average request size is around 12 kbytes.

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.

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 0.47 for the slow disks. These values have been decided experimentally and a sensitivity analysis for these parameters is presented in Sections 6.6 and 6.7. 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.

Regarding AdaptRaid5, an 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.

5.4. *Reference systems*

We have compared our policies with the following two base configurations:

- *RAID[0,5]*: This is the traditional RAID[0,5] 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 and that only a portion of their capacity is effectively used.
- *OnlyFast[0,5]*: This is also a traditional RAID[0,5], 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.

6. Experimental results

The evaluation of a disk array is twofold. On the one hand, we have to evaluate the effective capacity a given distribution algorithm offers. On the other hand, we have to test the performance the array is able to a achieve. Both evaluations are necessary because the importance of them depends on the site in which the array is installed. In some cases capacity is the important issue, while in other sites performance is the goal.

Finally, we will also present a detailed study on the sensitivity of the two parameters that appear on an AdaptRaid system (*SIP* and *UFs*). This analysis will give us an idea on how to set them and how difficult is to find the right values.

6.1. *Capacity evaluation*

Fig. 8 presents the effective capacity of all studied configurations as we vary the number of fast disks out of the total of 9 disks. We have to clarify that the effective capacity only takes into account data blocks usable by the “user”, parity information is not counted.

We can see that in both cases, level 0 (left) and level 5 (right), AdaptRaid 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.

We can also see that the extra number of parity blocks used by AdaptRaid5 (right) does not affect the effective capacity significantly.

6.2. *Read performance*

In order to study the performance of AdaptRaid, we will start presenting the results obtained with read operations and then a discussion about write operations will follow.

The evaluation of read performance has been done measuring the number of requests per second obtained by the system when requesting read operations 8, and 2048 kbytes long (workloads R8 and R2048 described in Section 5.2). These results are presented in Figs. 9 and 10. For each figure, the graph in the left presents the results for configurations level 0 while the right graph presents the results for configurations level 5. We should

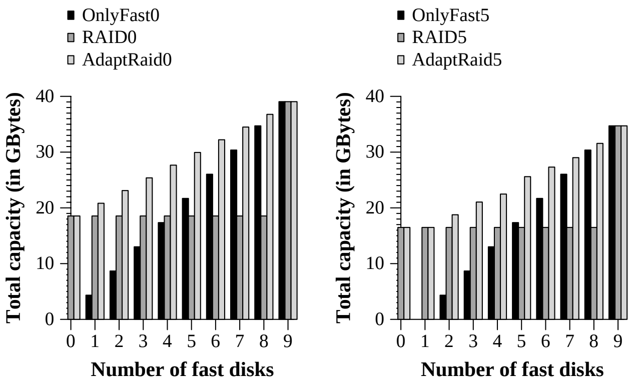

Fig. 8. Effective capacity for the studied configurations.

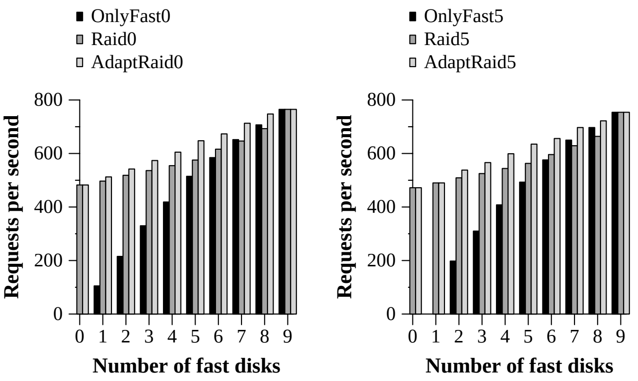

Fig. 9. Reading 8 kbytes blocks (R8).

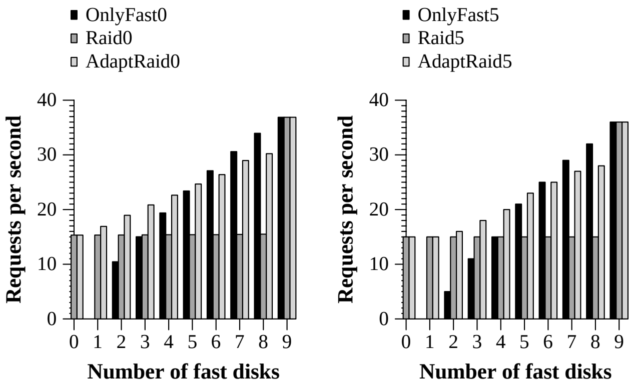

Fig. 10. Reading 2048 kbytes blocks (R2048).

notice that AdaptRaid5 configurations have not been evaluated for 1 fast disks because we need at least 2 disks to build a RAID5.

In the first case (Fig. 9), where requests are 8 kbytes, we observe a very similar behavior for both cases, levels 0 and 5. In both cases, AdaptRaid always obtains the better performance because it is able to use the faster disks more that the slower ones without overloading them. We can also observe that using a traditional RAID with fast and slow disks is better than only using a RAID with only fast disks. This happens because the parallelism obtained by using all disks is more important than the performance obtained by the fast disks. This holds until the system has 7 fast disks. From this point on, using the slow disks in a traditional RAID slows the performance and it is better only to use the fast disks, unless AdaptRaid policies are used. This change in the behavior appears because with 7 disks, the parallelism obtained by OnlyFast is enough to outweigh the parallelism achieved by RAID, which has to use slow disks to achieve it.

In the second case (Fig. 10), the requests are much larger and this has two effects. If we observe the performance of RAID0 and RAID5, it remains unmodified when more fast disks are added. This happens because all disks are used in the request and thus, slow disks are always included becoming the bottleneck of the operation. If we focus on OnlyFast, in both levels 0 and 5, we can see that it outperforms AdaptRaid 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, AdaptRaid 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, AdaptRaid is not the best solution.

6.3. *Full-write performance*

Now it is time to study the performance of write operations. This study will also be done testing several request sizes to guarantee that all cases have been studied (especially small writes and full-stripe writes).

The performance obtained by a RAID when full stripes are written is one of the important results for this kind of array because, in this case, write operations achieve their best possible performance (a write does not imply a previous read). To study this case, we have measured the number of requests per second each of the evaluated systems can achieve when requests are 1024 and 2048 kbytes 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. On the other hand, controllers or file systems may use logging to achieve such request sizes in non multimedia environments. Figs. 11 and 12 present these results.

In the case of level 0 (left graphs), the behavior is quite similar to the one we presented for long reads. This similarity appears because, in a RAID level 0, there is no difference between read and write operations.

In the case of level 5, the results are not similar to the ones obtained for read operations. This happens because in write operations, a parity block has to be computed and written, and this makes a difference in the behavior.

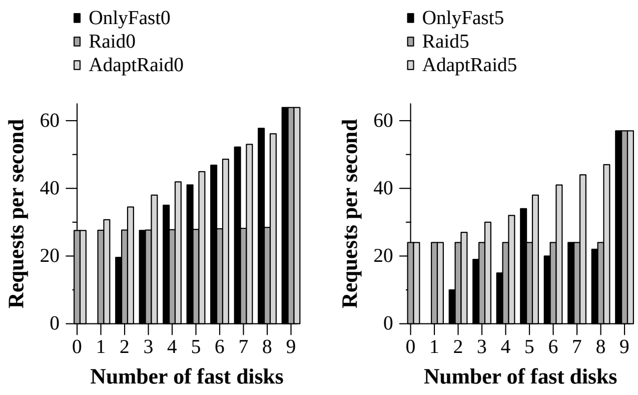

Fig. 11. Writing 1024 kbytes blocks (W1024).

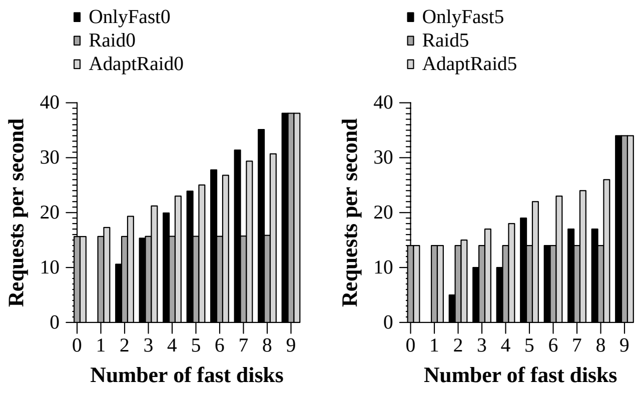

Fig. 12. Writing 2048 kbytes blocks (W2048).

Especially in the case of OnlyFast5 as we will seen in the following paragraphs.

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 all requests use all disks equally (slow ones included), as we saw while studying read operations.

The second system, OnlyFast5, 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, 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 OnlyFast5 is quite good and proportional to the number of fast disks.

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 OnlyFast5 also shows that our proposal is a better one. On the one hand, AdaptRaid5 is much more consistent than OnlyFast5 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 OnlyFast5. 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 for the large requests. Furthermore, when OnlyFast5 starts to take advantage of the parallelism (when more fast disks are used), AdaptRaid5 starts to use the slow disks less frequently, which outweighs the improvements of OnlyFast5.

6.4. *Small-write performance*

The other possibility for a write operation is to perform a small write. In this case, some blocks need 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 AdaptRaid and OnlyFast when only fasts disks are used, while it will be exploited by RAID that always uses all disks.

To do this evaluation we have measured the number of requests per second achieved by each evaluated system when 8 and 256 kbytes requests are done (workloads W8 and W256 described in Section 5.2) (Figs. 13 and 14).

Like in the case of long writes, disk arrays level 0 do not change their behavior form the one observed in the read operations. For this reason, we only present a detailed description of short writes for arrays level 5.

In this case, AdaptRaid5 is also better than RAID5, for the same reason as before. It knows how to use 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 OnlyFast5, we observe that our proposal has a better performance than OnlyFast5. This happens because AdaptRaid5 can use more disks and it can take advantage of the parallelism between requests.

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 Fig. 15. In these graphs, we present the performance gains (in %) obtained by our distribution algorithm when compared to RAID[0,5] and OnlyFast. The graphs are 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 many fast disks are used. In this case, OnlyFast can be 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 (unless the capacity is needed.)

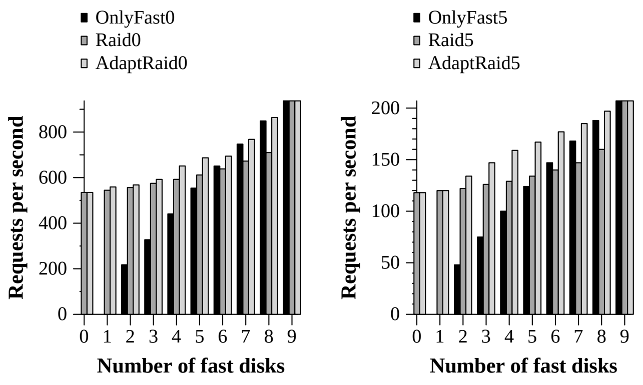

Fig. 13. Writing 8 kbytes blocks (W8).

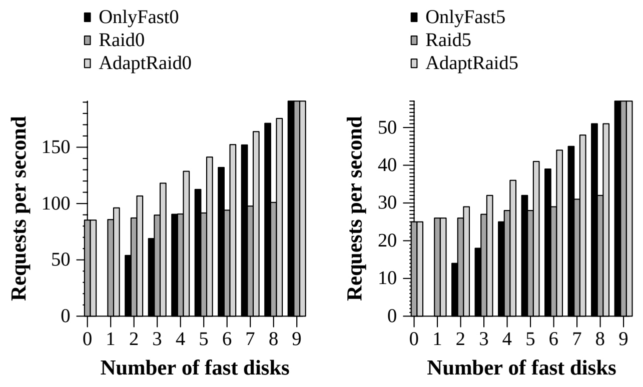

Fig. 14. Writing 256 kbytes blocks (W256).

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 run some of the experiments 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 Figs. 16–19. The combinations of UF values used range, on the one hand, from $UF_{\text{fast}} = 1$ and $UF_{\text{slow}} = 0.1$ to both $UF_{\text{fast}} = UF_{\text{slow}} = 1$. These tests have been marked in Figs. 16–19 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, $UF_{\text{slow}} = 1$ and UF_{fast} takes 0.9 and 0.7 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.

The synthetic workloads chosen are R8, W8, R2048 and W2048. This subset is representative enough

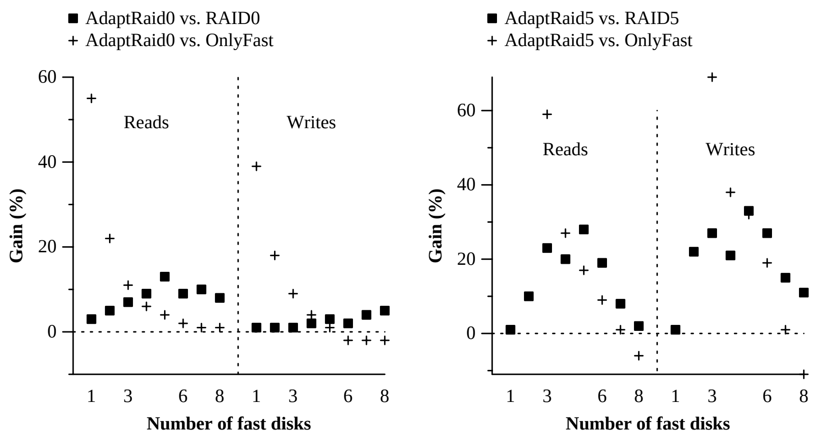

Fig. 15. Performance gain of AdaptRaid over the rest of configurations in a real workload.

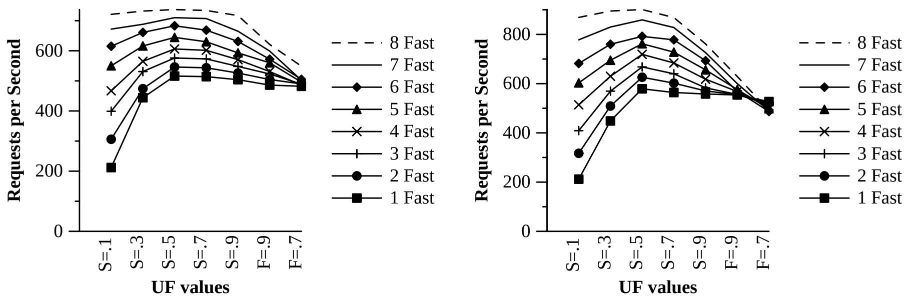

Fig. 16. Requests per second obtained while varying the value of the *UF* parameter in a AdaptRaid0 for 8 kbytes read (left) write (right) requests.

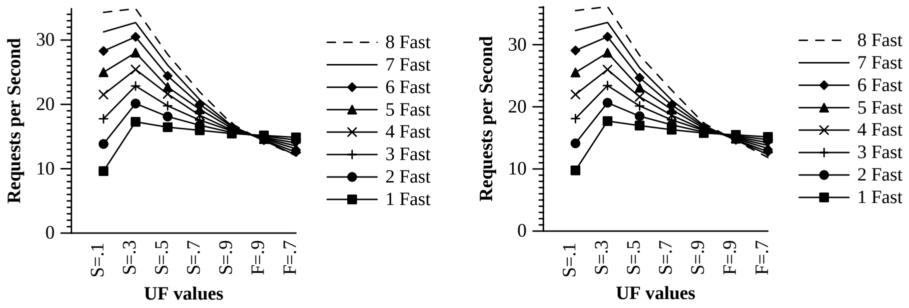

Fig. 17. Requests per second obtained while varying the value of the *UF* parameter in a AdaptRaid0 for 2048 kbytes read (left) and write (right) requests.

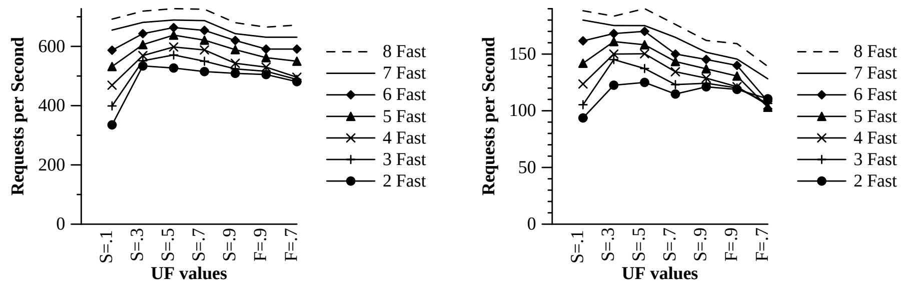

Fig. 18. Requests per second obtained while varying the value of the *UF* parameter in a AdaptRaid5 for 8 kbytes read (left) and write (right) requests.

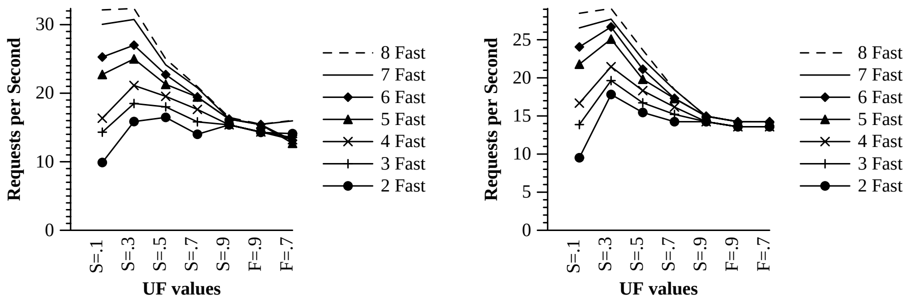

Fig. 19. Requests per second obtained while varying the value of the *UF* parameter in a AdaptRaid5 for 2048 kbytes read (left) and write (right) requests.

because it contains both reads and writes for both short and long requests.

Finally, we have run this analysis for both AdaptRaid0 and AdaptRaid5.

6.6.1. *AdaptRaid0*

Figs. 16 and 17 present the number of requests per second that each AdaptRaid0 configuration achieves depending on the *UFs* used.

The first thing we notice is that using either slow disks or fast disks too much ends up in a performance penalty. This means that finding a good balance between their usage is very important.

We can also observe that the effect of the *UF* factors depends on whether the configuration has few or many fast disks. On the first case, in arrays with few fast disks, using them too much reduces the potential parallelism and incurs in a performance penalty. On the other hand, using fast disks too little does not have much impact in the performance because parallelism is the key issue to performance in these configurations. On the second case, in arrays with many fast disks, using them too little decreases the performance as many requests use the few slow disks. On the other hand, using them too much is not a big problem because they achieve enough parallelism and at the same time avoids using the slow disks as much as possible.

If we focus on the small requests (Fig. 16), we can see that the optimal value is around $S = 0.5$ ($UF_{\text{slow}} = 0.5$ and $UF_{\text{fast}} = 1$) but if we focus on large requests the optimal value is around $S = 0.3$ ($UF_{\text{slow}} = 0.3$ and $UF_{\text{fast}} = 1$). These optimal values are different because in the first case it reflects the difference between the latency of both disks while the second case it reflects the difference in the bandwidth.

6.6.2. AdaptRaid5

In the case of AdaptRaid5 (Figs. 18 and 19), the results are quite similar to the ones presented for AdaptRaid0. The main difference is that the curves are not as smooth as they were in the previous case. This happens because a change in the *UF* parameters can lead to completely different mapping of parity blocks, which may change the behavior of the array. Nevertheless, we can still detect that a value between $S = 0.3$ and 0.5 is the one that achieves the better performance for the disks we have used.

6.7. Sensitivity analysis of the SIP parameter

Once we have seen that using the capacity of the disk to compute the UFs is a good compromise (in the studied cases), we can fix them and study the behavior of the array when the *SIP* parameter is modified.

In this section, we present a set of graphs (Figs. 20–23) that show the effect that using different values of *SIP* (from 2 to 128) has in the number of requests per second for each configuration.

6.7.1. AdaptRaid0

From Fig. 20, we can see that the value of *SIP* is not important when requests are small. Especially in our case where 2 stripes can represent quite well the *UFs* used (1 and 0.47). Actually, any number of stripes that can be divided into values similar to what the *UF* factors represent, will achieve a good performance results for small requests. Remember that the distribution is a uniform one, and thus requests are spread evenly throughout all the array address space.

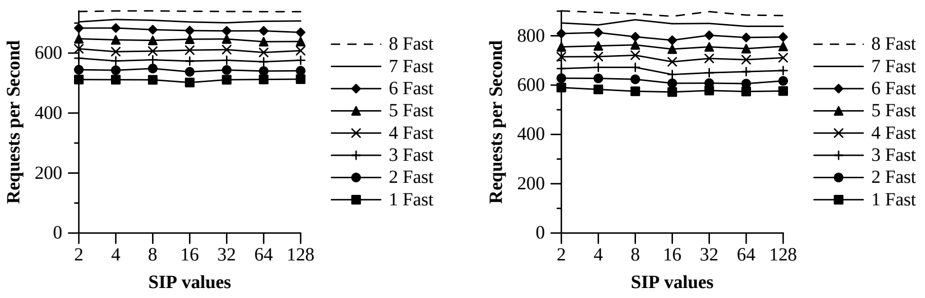

Fig. 20. Requests per second obtained while varying the value of the *SIP* parameter in a AdaptRaid0 for 8 kbytes read (left) and write (right) requests.

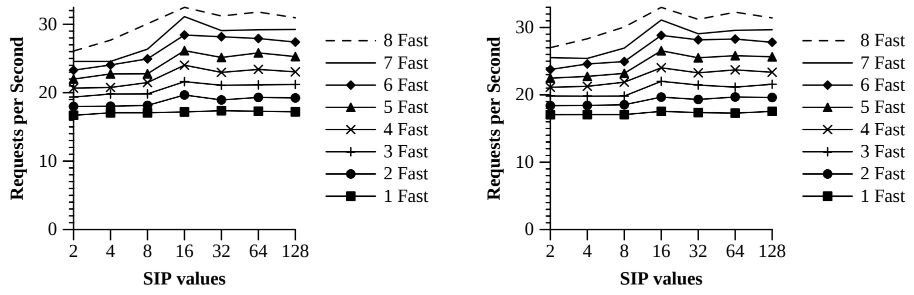

Fig. 21. Requests per second obtained while varying the value of the *SIP* parameter in a AdaptRaid0 for 2048 kbytes read (left) and write (right) requests.

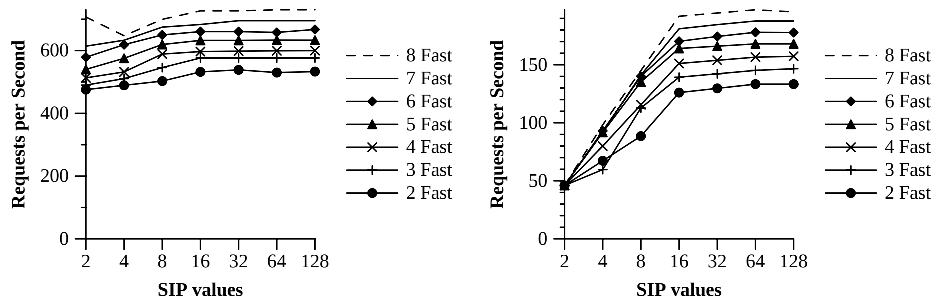

Fig. 22. Requests per second obtained while varying the value of the *SIP* parameter in a AdaptRaid5 for 8 kbytes read (left) and write (right) requests.

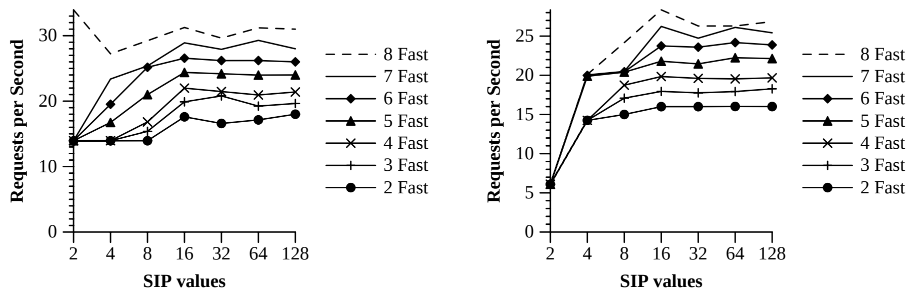

Fig. 23. Requests per second obtained while varying the value of the *SIP* parameter in a AdaptRaid5 for 2048 kbytes read (left) and write (right) requests.

In the case of large requests (Fig. 21), small values of *SIP* are not very good as slow disks are used in all requests. After a given point, between 16 and 32, the performance gets stable and increasing the number of stripes in the pattern does not affect the performance. Remember that, in this case, we also have a uniform distribution of requests and thus the average of request time will not be affected by a large value for *SIP*.

6.7.2. AdaptRaid5

In the case of small reads (Fig. 22—left), the behavior of AdaptRaid5 is quite similar to the one observed for small requests in AdaptRaid0 and the same reasoning can be used.

Small writes (Fig. 22—right)have a quite different behavior. In this case the location of the parity blocks is quite vital. Remember that we have a pattern, with SIP stripes, that is repeated continuously, and the same thing happens for the parity location. This means that if $SIP = 2$ there are two stripes and only two disks hold parity blocks. This overloads them in the case of writes, and thus large enough values for *SIP* are needed to reach a balance in the distribution of parity blocks. Nevertheless, in this case, the optimal behavior starts somewhere from 16 to 32. From that point on, the behavior remains quite stable, as expected.

Finally, in the case of long requests, we can see a stable behavior after $SIP = 16$. The experiments with smaller values of *SIP* are affected by the fact that *SIP* has to be large enough not to use slow disks in all requests, plus the fact that parity information is not evenly distributed among all disks overloading some of the disks.

7. Conclusions

In this paper, we have presented AdaptRaid0 and AdaptRaid5, two block-distribution policies that take full advantage of heterogeneous disk arrays.

These algorithms achieve a significant performance compared to the policies currently being used.

Furthermore, we have also shown that arrays using AdaptRaid0 or 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.

We have also presented a detailed sensitivity analysis of the parameters that take part in the configuration of an AdaptRaid system. We have seen that defining *UF* factors greatly depends on whether performance or capacity is the important issue. Nevertheless, the possible values are in a range easy to detect. We have also seen that setting a value for the *SIP* parameter is very easy. It just needs to be large enough to represent the possible *UF* factors and, in the case of AdaptRaid5, large enough to distribute parity blocks among all disks.

It is also important to notice, that a system administrator could run the same experiments we have used in the sensitivity analysis to find the adequate values of *UF* and *SIP* for a given configuration. With the results of these experiments, and knowing the balance between performance and capacity needed for the given configuration, setting the parameters is a simple task.

Finally, we have to keep in mind that these algorithms have been evaluated for an array built from disks attached to a SAN, but there is no reason to believe that they would not work in other array configurations.

8. Availability

Home page for this project

- `http://people.ac.upc.es/toni/AdaptRaid.html`

Other papers and reports about heterogeneous disk arrays

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

Simulator information and downloading

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

References

[1] T.E. Anderson, M.D. Dahlin, J.M. Neefe, D.A. Patterson, D.S. Roselli, R.Y. Wang, Serverless network file systems, in: Proceedings of the 15th Symposium on Operating Systems Principles, Copper Mountain Resort, CO, 1995, pp. 109–126.

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

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

[4] P.M. Chen, D.A. Patterson, Maximizing performance in striped disk arrays, in: Proceedings of the 1990 International Symposium on Computer Architecture, Seattle, WA, 1990, pp. 322–331.

[5] T. Cortes, J. Labarta, HRaid: A flexible storage-system simulator, in: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, CSREA Press, Las Vegas, NV, 1999, pp. 772–778.

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

[7] E. Grochowski, R.F. Hoyt, Future trends in hard disk drives, IEEE Trans. Magn. 32 (3) (1996) 1850–1854.

[8] J. Hartman, J.K. Ousterhout, The zebra striped network file system, Trans. Comput. System 13 (3) (1995) 274–310.

[9] M. Holland, G.A. Gibson, Parity declustering for continuous operation in redundant disk arrays, in: Proceedings of the Fifth Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA, 1992, pp. 23–35.

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

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

[12] E.K. Lee, R.H. Katz, The performance of parity placements in disk arrays, IEEE Trans. Comput. 42 (6) (1993) 651–664.

[13] M. McKusick, W. Joy, S. Leffler, R. Fabry, A fast file system for unix, ACM Trans. Comput. Systems 2 (3) (1984) 181–197.

[14] L. McVoy, S. Kleiman, Extent-like performance from a unix file system, in: Proceedings of the Summer Technical Conference, USENIX Association, Dallas, TX, 1990, pp. 137–144.

[15] C. Ruemmler, J. Wilkes, Unix disk access patterns, in: Proceedings of the Winter USENIX Conference, San Diego, CA, 1993, pp. 405–420.

[16] C. Ruemmler, J. Wilkes, An introduction to disk drive modeling, IEEE Comput. (1994) 17–28.

[17] J.R. Santos, R. Muntz, Performance analysis of the rio multimedia storage system with heterogenenous disk configurations, ACM Multimedia September 12–16, 1998, Bristol, UK, (1998) 303–308.

[18] Seagate, Seagate web page, http://www.seagate.com (January 2000).

[19] K.A. Smith, M. Seltzer, A comparison of FFS disk allocation policies, in: Proceedings of the Annual Technical Conference, USENIX Association, San Diego, CA, 1996.

[20] D. Stodolsky, G. Gibson, M. Holland, Parity logging overcoming the small write problem in redundant disk arrays, in: Proceedings of the 21st Annual International Symposium on Computer Architecture, San Diego, CA, 1993, pp. 64–75.

[21] J. Wilkes, personal communication, Septemeber 1999.

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

[23] J. Wilkes, R. Golding, C. Staelin, T. Sullivan, The HP AutoRAID hierarchical storage system, in: Proceedings of the 15th Operating System Review, ACM Press, New York, 1995, pp. 96–108.

[24] R. Zimmermann, Continuous media placement and scheduling in heterogeneous disk storage systems, Ph.D. Thesis, University of Southern California, December 1998.

Toni Cortes is an associate professor at the Universitat Politecnica de Catalunya (Barcelona–Spain) since 1999. He obtained his M.S. and Ph.D. in computer science at the same university, and is currently the coordinator of the single-system image technical area in the IEEE Task Force on Cluster Computing (TFCC). His main interests are cluster computing and parallel I/O. Dr. Cortes has also been working on several European industrial projects such as Paros, Nanos, and POP. As a result of his research, both in basic research and technology transfer, Dr. Cortes has published more than 25 papers in international journals and conferences, has published 2 book chapters and has edited a book on parallel I/O (“High Performance Mass Storage and Parallel I/O: Technologies and Applications”, IEEE press). He also acts a program committee for many international conferences and journals.

Jesus Labarta is full professor at the Computer Architecture department of UPC (technical University of Catalonia) since 1990. Since 1981 he has been lecturing on computer architecture, operating systems, computer networks and performance evaluation. His research interest has been centered on parallel computing, covering areas from multiprocessor architecture, memory hierarchy, parallelizing compilers, operating systems, parallelization of numerical kernels, metacomputing tools and performance analysis and prediction tools. He has leaded the technical work of UPC in 15 industrial

R + D projects. Since 1995 he is director of CEPBA where he has been highly motivated by the promotion of parallel computing into industrial practice, and especially within SMEs. In this line he has acted as responsible of three technology transfer cluster projects where his team managed 28 subprojects.

His major directions of current work relate to performance analysis tools and OpenMP. The most representative result of his work on tools are Paraver and Dimemas. The work on OpenMP and OS scheduling is reflected in the NANOS platform. He actively participates in the OpenMP ARB Futures Committee proposing and evaluating the potential of future extensions to the standard.

Since 2000 he is strongly committed in carrying out and promoting productive research co-operation with IBM as part of the CEPBA-IBM Research Institute.