37

Redundant Arrays of Inexpensive Disks (RAIDs)

When we use a disk, we sometimes wish it to be faster; I/O operations are slow and thus can be the bottleneck for the entire system. When we use a disk, we sometimes wish it to be larger; more and more data is being put online and thus our disks are getting fuller and fuller. When we use a disk, we sometimes wish for it to be more reliable; when a disk fails, if our data isn't backed up, all that valuable data is gone.

In this note, we introduce the **Redundant Array of Inexpensive Disks** better known as **RAID** [P+88], a technique to use multiple disks in concert to build a faster, bigger, and more reliable disk system. The term was introduced in the late 1980s by a group of researchers at U.C. Berkeley (led by Professors David Patterson and Randy Katz and then student Garth Gibson); it was around this time that many different researchers simultaneously arrived upon the basic idea of using multiple disks to build a better storage system [BG88, K86,K88,PB86,SG86].

From the outside, a RAID looks like a disk: a group of blocks each of which one can read or write. Internally, however, the RAID is a complex beast, consisting of multiple disks, memory (both volatile and non-volatile), and one or more processors to manage the system. Thus, a hardware RAID box is very much like a computer system, but just specialized for the task of managing a group of disks.

RAIDs offer a number of advantages over a single disk. One advantage is *performance*. Using multiple disks in parallel can greatly speed up I/O times. Another benefit is *capacity*. Large data sets

demand large disks. Finally, RAIDs can improve *reliability*; spreading data across multiple disks (without RAID techniques) makes the data vulnerable to the loss of a single disk; with some form of **redundancy**, RAIDs can tolerate the loss of a disk and keep operating as if nothing were wrong.

Amazingly, RAIDs provide these advantages **transparently** to systems that use them, i.e., a RAID just looks like a big disk to the host system. The beauty of transparency, of course, is that it enables one to simply replace a disk with a RAID and not change a single line of software; the operating system and client applications continue to operate without modification. In this manner, transparency greatly improves the **deployability** of RAID, enabling users and administrators to put a RAID to use without worries of software compatibility.

DESIGN TIP: TRANSPARENCY

When considering how to add new functionality to a system, one should always consider whether such functionality can be added **transparently**, in a way that demands no changes to the rest of the system. Requiring a complete rewrite of the existing software (or radical hardware changes) lessens the chance of impact of an idea.

We now discuss some of the important aspects of RAIDs. We begin with the interface, fault model, and then discuss how one can evaluate a RAID design along three important axes: capacity, reliability, and performance. We then discuss a number of other issues that are important to RAID design and implementation.

37.1 Interface and RAID Internals

To a file system above, a RAID looks like a big, (hopefully) fast, and (hopefully) reliable disk. Just as with a single disk, it presents itself as a linear array of blocks, each of which can be read or written by the file system (or other client).

When a file system issues a *logical I/O* request to the RAID, the RAID internally must calculate which disk (or disks) to access in order to complete the request, and then issue one or more *physical I/Os* to do so. The exact nature of these physical I/Os depends on the RAID level, as we will discuss in detail below. However, as a simple

example, consider a RAID that keeps two copies of each block (each one on a separate disk); when writing to such a **mirrored** RAID system, the RAID will have to perform two physical I/Os for every one logical I/O it is issued.

A RAID system is often built as a separate hardware box, with a standard connection (e.g., SCSI, or SATA) to a host. Internally, however, RAIDs are fairly complex, consisting of a microcontroller that runs firmware to direct the operation of the RAID, volatile memory such as DRAM to buffer data blocks as they are read and written, and in some cases, non-volatile memory to buffer writes safely and perhaps even specialized logic to perform parity calculations (useful in some RAID levels, as we will also see below). At a high level, a RAID is very much a specialized computer system: it has a processor, memory, and disks; however, instead of running applications, it runs specialized software designed to operate the RAID.

37.2 Fault Model

To understand RAID and compare different approaches, we must have a fault model in mind. RAIDs are designed to detect and recover from certain kinds of disk faults; thus, knowing exactly which faults to expect is critical in arriving upon a working design.

The first fault model we will assume is quite simple, and has been called the **fail-stop** fault model [S84]. In this model, a disk can be in exactly one of two states: working or failed. With a working disk, all blocks can be read or written. In contrast, when a disk has failed, we assume it is permanently lost.

One critical aspect of the fail-stop model is what it assumes about fault detection. Specifically, when a disk has failed, we assume that this is easily detected. For example, in a RAID array, we would assume that the RAID controller hardware (or software) can immediately observe when a disk has failed.

Thus, for now, we do not have to worry about more complex "silent" failures such as disk corruption. We also do not have to worry about a single block becoming inaccessible upon an otherwise working disk (sometimes called a latent sector error). We will consider these more complex (and unfortunately, more realistic) disk faults later.

37.3 How to Evaluate a RAID

As we will soon see, there are a number of different approaches to building a RAID. Each of these approaches has different characteristics which are worth evaluating, in order to understand their strengths and weaknesses.

Specifically, we will evaluate each RAID design along three axes. The first axis is **capacity**; given a set of N disks, how much useful capacity is available to systems that use the RAID? Without redundancy, the answer is obviously N; however, if we have a system that keeps a two copies of each block, we will obtain a useful capacity of N/2.

The second axis of evaluation is **reliability**. How many disk faults can the given design tolerate? In alignment with our fault model, we assume only that an entire disk can fail.

Finally, the third axis is **performance**. Performance is somewhat challenging to evaluate, because it depends heavily on the workload presented to the disk array. Thus, before evaluating performance, we will first present a set of typical workloads that one should consider.

We now consider three important RAID designs: RAID Level 0 (striping), RAID Level 1 (mirroring), and RAID Levels 4/5 (parity-based redundancy). The naming of each of these designs as a "level" stems from the pioneering work of Patterson, Gibson, and Katz at Berkeley [P+88].

37.4 RAID Level 0: Striping

The first RAID level is actually not a RAID level at all, in that there is no redundancy. However, RAID level 0, or **striping** as it is better known, serves as an excellent upper-bound on performance and capacity and thus is worth understanding.

The simplest form of striping will **stripe** blocks across the disks of the system as follows (assume here a 4-disk array):

From Table 37.1, you get the basic idea: spread the blocks of the array across the disks in a round-robin fashion. This approach is designed to extract the most parallelism from the array when requests are made for contiguous chunks of the array (as in a large, sequential read, for example). We call the blocks in the same row a **stripe**; thus, blocks 0, 1, 2, and 3 are in the same stripe above.

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Table 37.1: RAID-0: Simple Striping

In the example, we have made the simplifying assumption that only 1 block (each of say size 4KB) is placed on each disk before moving on to the next. However, this arrangement need not be the case. For example, we could arrange the blocks across disks as in Table 37.2:

Disk 0	Disk 1	Disk 2	Disk 3	
0	2	4	6	chunk size:
1	3	5	7	2 blocks
8	10	12	14	
9	11	13	15	

Table 37.2: Striping with a Bigger Chunk Size

In this example, we place two 4KB blocks on each disk before moving on to the next disk. Thus, the **chunk size** of this RAID array is 8KB, and a stripe thus consists of 4 chunks or 32KB of data.

Chunk Sizes

Chunk size mostly affects performance of the array. For example, a small chunk size implies that many files will get striped across many disks, thus increasing the parallelism of reads and writes to a single file; however, the positioning time to access blocks across multiple disks increases, because the positioning time for the entire request is determined by the maximum of the positioning times of the requests across all drives.

A big chunk size, on the other hand, reduces such intra-file parallelism, and thus relies on multiple concurrent requests to achieve high throughput. However, large chunk sizes reduce positioning time; if, for example, a single file fits within a chunk and thus is

placed on a single disk, the positioning time incurred while accessing it will just be the positioning time of a single disk.

Thus, determining the "best" chunk size is hard to do, as it requires a great deal of knowledge about the workload presented to the disk system [CL95]. For the rest of this discussion, we will assume that the array uses a chunk size of a single block (4KB); most arrays use larger chunk sizes (say around 64 KB), but for the issues we discuss below, the exact chunk size does not matter and thus we use a single block for the sake of simplicity.

ASIDE: THE RAID MAPPING PROBLEM

Before studying the capacity, reliability, and performance characteristics of the RAID, we first present an aside on what we call **the mapping problem**. This problem arises in all RAID arrays; simply put, given a logical block to read or write, how does the RAID know exactly which physical disk and offset to access?

For these simple RAID levels, we do not need much sophistication in order to correctly map logical blocks onto their physical locations. Take the first striping example above (chunk size = 1 block = 4KB). In this case, given a logical block address A, the RAID can easily compute the desired disk and offset with two simple equations:

```
Disk   = A % number_of_disks
Offset = A / number_of_disks
```

Note that these are all integer operations (e.g., 4 / 3 = 1 not 1.33333...).

Let's see how these equations work for a simple example. Imagine in the first RAID above that a request arrives for block 14. Given that there are 4 disks, this would mean that the disk we are interested in is (14 The exact block is calculated as (14 / 4 = 3): block 3. Thus, block 14 should be found on the fourth block (block 3, starting at 0) of the third disk (disk 2, starting at 0), which is exactly where it is.

You can think about how these equations would be modified to support different chunk sizes. Try it! It's not too hard.

Back to RAID-0 Analysis

Let us now evaluate striping. From the perspective of capacity, it is perfect: given N disks, striping delivers N disks worth of useful capacity. From the standpoint of reliability, striping is also perfect, but in the bad way: any disk failure will lead to data loss. Finally, performance is excellent.

Evaluating RAID Performance

In analyzing RAID performance, one can consider two different performance metrics. The first is *single-request latency*. Understanding the latency of a single I/O request to a RAID is useful as it reveals how much parallelism can exist during a single logical I/O operation. The second is *steady-state throughput* of the RAID, i.e., the total bandwidth of many concurrent requests. Because RAIDs are often used in high-performance environments, the steady-state bandwidth is critical, and thus will be the main focus of our analyses.

To understand throughput in more detail, we need to put forth some workloads of interest. We will assume, for this discussion, that there are two types of workloads: **sequential** and **random**. With a sequential workload, we assume that requests to the array come in large contiguous chunks; for example, a request (or series of requests) that accesses 1 MB of data, starting at block (B) and ending at block (B + 1 MB), would be deemed sequential. Sequential workloads are common in many environments (think of searching through a large file for a keyword), and thus are considered important.

For random workloads, we assume that each request is rather small, and that each request is to a different random location on disk. For example, a random stream of requests may first access 4KB at logical address 10, then at logical address 55000, then at 20100, and so forth. Some important workloads, like transactional workloads on a database, exhibit this type of access pattern, and thus it is considered an important workload as well.

Of course, real workloads are not so simple, and often have a mix of sequential and random-seeming components as well as behaviors in-between the two. However, for now, we will just consider these two possibilities.

As you can tell, sequential and random workloads will result in

widely different performance characteristics from a disk. With sequential access, a disk operates in its most efficient mode, spending little time seeking and waiting for rotation and most of its time transferring data. With random access, just the opposite is true: most time is spent seeking and waiting for rotation and relatively little time is spent transferring data. To capture this difference in our analysis, we will assume that a disk can transfer data at S MB/s under a sequential workload, and R MB/s when under a random workload. In general, S is much greater than R.

To make sure we understand this difference, let's do a simple exercise. Specifically, lets calculate S and R given the following disk characteristics. Assume a sequential transfer of size 10 MB on average, and a random transfer of 10 KB on average. Also, assume the following disk characteristics:

Average seek time	7 ms
Average rotational delay	3 ms
Transfer rate of disk	50 MB/s

To compute S, we need to first figure out how time is spent in a typical 10 MB transfer. First, we spend 7 ms seeking, and then 3 ms rotating. Finally, transfer begins; 10 MB @ 50 MB/s leads to 1/5th of a second, or 200 ms, spent in transfer. Thus, for each 10 MB request, we spend 210 ms completing the request. To compute S, we just need to divide:

$$S = \frac{Amount\ of\ Data}{Time\ to\ access} = \frac{10\ MB}{210\ ms} = 47.62\ MB/s$$

As we can see, because of the large time spent transferirng data, S is very near the peak bandwidth of the disk (the seek and rotational costs have been amortized).

We can compute R similarly. Seek and rotation are the same; we then compute the time spent in transfer, which is 10 KB @ 50 MB/s, or 0.195 ms.

$$R = \frac{Amount\ of\ Data}{Time\ to\ access} = \frac{10\ KB}{10.195\ ms} = 0.981\ MB/s$$

As we can see, R is less than 1 MB/s, and S/R is almost 50.

Back to RAID-0 Analysis, Again

Let's now evaluate the performance of striping. As we said above, it is generally good. From a latency perspective, for example, the

latency of a single-block request should be just about identical to that of a single disk; after all, RAID-0 will simply redirect that request to one of its disks.

From the perspective of steady-state throughput, we'd expect to get the full bandwidth of the system. Thus, throughput equals N (the number of disks) multiplied by S (the sequential bandwidth of a single disk). For a large number of random I/Os, we can again use all of the disks, and thus obtain $N \cdot R$ MB/s. As we will see below, these values are both the simplest to calculate and will serve as an upper bound in comparison with other RAID levels.

37.5 RAID Level 1: Mirroring

Our first RAID level beyond striping is known as RAID level 1, or mirroring. With a mirrored system, we simply make more than one copy of each block in the system; each copy should be placed on a separate disk, of course. By doing so, we can tolerate disk failures.

In a typical mirrored system, we will assume that for each logical block, the RAID makes two physical copies of the block. Here is a simple example:

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Table 37.3: Simple RAID-1: Mirroring

In the example, disk 0 and disk 1 have identical contents, and disk 2 and disk 3 do as well; the data is striped across these mirror pairs. In fact, you may have noticed that there are a number of different ways to place block copies across the disks. The arrangement above is a common one and is sometimes called **RAID-10** or (**RAID 1+0**) because it uses mirrored pairs (RAID-1) and then stripes (RAID-0) on top of them; another common arrangement is **RAID-01** (or **RAID 0+1**), which contains two large striping (RAID-0) arrays, and then mirrors (RAID-1) on top of them. For now, we will just talk about mirroring assuming the above layout.

When reading a block from a mirrored array, the RAID has a choice: it can read either copy. For example, if a read to logical block

5 is issued to the RAID, it is free to read it from either disk 2 or disk 3. When writing a block, though, no such choice exists: the RAID must update *both* copies of the data, in order to preserve reliability. Do note, though, that these writes can take place in parallel; for example, a write to logical block 5 could proceed to disks 2 and 3 at the same time.

ASIDE: THE RAID CONSISTENT UPDATE PROBLEM

Before analyzing RAID-1, let us first discuss a problem that arises in any multi-disk RAID system, known as the **consistent update problem** [DAA05]. The problem occurs on a write to any RAID that has to update multiple disks during a single logical operation. In this case, let us assume we are considering a mirrored disk array.

Imagine the write is issued to the RAID, and then the RAID decides that it must be written to two disks, disk 0 and disk 1. The RAID then issues the write to disk 0, but just before the RAID can issue the request to disk 1, a power loss (or system crash) occurs. In this unfortunate case, let us assume that the request to disk 0 completed (but clearly the request to disk 1 did not, as it was never issued).

The result of this untimely power loss is that the two copies of the block are now **inconsistent**; the copy on disk 0 is the new version, and the copy on disk 1 is the old. What we would like to happen is for the state of both disks to change **atomically**, i.e., either both should end up as the new version or neither.

The general way to solve this problem is to use a **write-ahead log** of some kind to first record what the RAID is about to do (i.e., update two disks with a certain piece of data) before doing it. By taking this approach, we can ensure that in the presence of a crash, the right thing will happen; by running a **recovery** procedure that replays all pending transactions to the RAID, we can ensure that no two mirrored copies (in the RAID-1 case) are out of sync.

One last note: because logging to disk on every write is prohibitively expensive, most RAID hardware includes a small amount of non-volatile RAM (e.g., battery-backed) where it performs this type of logging. Thus, consistent update is provided without the high cost of logging to disk.

RAID-1 Analysis

Let us now assess RAID-1. From a capacity standpoint, RAID-1 is pretty expensive; with the mirroring level = 2, we only obtain half of our peak useful capacity. Thus, with N disks, the useful capacity of mirroring is N/2.

From a reliability standpoint, RAID-1 does well. It can tolerate the failure of any one disk. However, you may notice RAID-1 can actually do better than this, with a little luck. Imagine, in the figure above, that disk 0 and disk 2 both failed. In such a situation, there is still no data loss! More generally, a mirrored system (with mirroring level = 2) can tolerate 1 disk failure for certain, and up to N/2 failures depending on which disks fail. In real life, however, we generally don't like to leave things like this to chance, and thus most people consider mirroring to be good for handling a single failure.

Finally, we analyze performance. From the perspective of the latency of a single read request, we can see it is the same as the latency on a single disk; all the RAID-1 does is direct the read to one of its copies. A write is a little different: it requires two physical writes to complete before it is done. These two writes happen in parallel, and thus the time will be roughly equivalent to the time of a single write; however, because the logical write must wait for both physical writes to complete, it suffers from the worst-case seek and rotational delay of the two requests, and thus (on average) will be just a little bit higher than a single write to a single disk.

To analyze steady-state throughput, let us start with the sequential workload. When writing out to disk sequentially, each logical write must result in two physical writes; for example, when we write logical block 0 (in the figure above), the RAID internally would write it to both disk 0 and disk 1. Thus, we can conclude that the maximum bandwidth obtained during sequential writing to a mirrored array is $(\frac{N}{2} \cdot S)$, or half the peak bandwidth.

Unfortunately, we obtain the exact same performance during a sequential read. One might think that a sequential read could do better, because it only needs to read one copy of the data, not both. However, let's use an example to illustrate why this doesn't help much. Imagine we need to read blocks 0, 1, 2, 3, 4, 5, 6, and 7. Let's say we issue the read of 0 to disk 0, the read of 1 to disk 2, the read of 2 to disk 1, and the read of 3 to disk 3. We continue by issuing reads to 4, 5, 6, and 7 to disks 0, 2, 1, and 3, respectively. One might naively

think that because we are utilizing all the disks in this example, we are achieving the full bandwidth of the array.

To see that this is not the case, however, consider the requests a single disk receives (say disk 0). First, it gets a request for block 0; then, it gets a request for block 4 (skipping block 2). In fact, each disk receives a request for every other block. While it is rotating over the skipped block, it is not delivering useful bandwidth to the client. Thus, each disk will only deliver half its peak bandwidth. And thus, the sequential read will only obtain a bandwidth of ($\frac{N}{2} \cdot S$) MB/s.

Random reads are the best case for a mirrored RAID. In this case, we can distribute the reads across all the disks, and thus obtain the full possible bandwidth. Thus, for random reads, RAID-1 delivers $N \cdot R$ MB/s.

Finally, random writes perform as you might expect: $\frac{N}{2} \cdot R$ MB/s. Each logical write must turn into two physical writes, and thus while all the disks will be in use, the client will only perceive this as half the available bandwidth. Even though a write to logical block X turns into two parallel writes to two different physical disks, the bandwidth of many small requests only achieves half of what we saw with striping. As we will soon see, getting half the available bandwidth is actually pretty good!

37.6 RAID Level 4: Saving Space with Parity

We now present a different method of adding redundancy to a disk array known as **parity**. Parity-based approaches attempt to use less capacity and thus overcome the huge space penalty paid by mirrored systems. They do so at a cost, however: performance.

In a five-disk RAID-4 system, we might observe the following data layout:

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

As you can see, for each stripe of data, we have added a single **parity** block that stores the redundant information for that stripe of

blocks. For example, parity block P1 has redundant information that it calculated from blocks 4, 5, 6, and 7.

To compute parity, we need to use some kind of mathematical function that enables us to withstand the loss of any one block from our stripe. It turns out the simple function **XOR** does the trick quite nicely. For a given set of bits, the XOR of all of those bits returns a 0 if there are an even number of 1's in the bits, and a 1 if there are an odd number of 1's. For example:

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1) = 0
0	1	0	0	XOR(0,1,0,0) = 1

In the first row (0,0,1,1), there are two 1's (C2, C3), and thus XOR of all of those values will be 0 (P); similarly, in the second row there is only one 1 (C1), and thus the XOR must be 1 (P). You can remember this in a very simple way: that the number of 1's in any row must be an even (not odd) number; that is the **invariant** that the RAID must maintain in order for parity to be correct.

From the example above, you might also be able to guess how parity information can be used to recover from a failure. Imagine the column labeled C2 is lost. To figure out what values must have been in the column, we simply have to read in all the other values in that row (including the XOR'd parity bit) and **reconstruct** the right answer. Specifically, assume the first row's value in column C2 is lost (it is a 1); by reading the other values in that row (0 from C0, 0 from C1, 1 from C3, and 0 from the parity column P), we get the values 0, 0, 1, and 0. Because we know that XOR keeps an even number of 1's in each row, we know what the missing data must be: a 1. And that is how reconstruction works in a XOR-based parity scheme! Note also how we compute the reconstructed value: we just XOR the data bits and the parity bits together, in the same way that we calculated the parity in the first place.

Now you might be wondering: we are talking about XORing all of these bits, and yet above we know that the RAID places 4KB (or larger) blocks on each disk; how do we apply XOR to a bunch of blocks to compute the parity? It turns out this is easy as well. Simply perform a bitwise XOR across each bit of the data blocks; put the result of each bitwise XOR into the corresponding bit slot in the parity block. For example, if we had blocks of size 4 bits (yes, this is still quite a bit smaller than a 4KB block, but you get the picture), they

might look something like this:

Block0	Block1	Block2	Block3	Parity
00	10	11	10	11
10	01	00	01	10

As you can see from the figure, the parity is computed for each bit of each block and the result placed in the parity block.

RAID-4 Analysis

Let us now analyze RAID-4. From a capacity standpoint, RAID-4 uses 1 disk for parity information for every group of disks it is protecting. Thus, our useful capacity for a RAID group is (N-1).

Reliability is also quite easy to understand: RAID-4 tolerates 1 disk failure and no more. If more than one disk is lost, there is simply no way to reconstruct the lost data.

Finally, there is performance. This time, let us start by analyzing steady-state throughput. Sequential read performance can utilize all of the disks except for the parity disks, and thus deliver a peak effective bandwidth of $(N-1) \cdot S$ MB/s (an easy case).

To understand the performance of sequential writes, we must first understand how they are done. When writing a big chunk of data to disk, RAID-4 can perform a simple optimization known as a **full-stripe write**. For example, imagine the case where the blocks 0, 1, 2, and 3 have been sent to the RAID as part of a write request (Table 37.4).

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Table 37.4: Full-stripe Writes In RAID-4

In this case, the RAID can simply calculate the new value of P0 (by performing an XOR across the blocks 0, 1, 2, and 3) and then write all of the blocks (including the parity block) to the five disks above in parallel (highlighted in gray in the figure). Thus, full-stripe writes are the most efficient way for RAID-4 to write to disk.

Once we understand the full-stripe write, calculating the performance of sequential writes on RAID-4 is easy; the effective bandwidth is also $(N-1) \cdot S$ MB/s. Even though the parity disk is constantly in use during the operation, the client does not gain any performance advantage from it.

Now let us analyze the performance of random reads. As you can also see from the figure above, a set of 1-block random reads will be spread across the data disks of the system but not the parity disk. Thus, the effective performance is: $(N-1) \cdot R$ MB/s.

Random writes, which we have saved for last, present the most interesting case for RAID-4. Imagine we wish to overwrite block 1 in the example above. We could just go ahead and overwrite it, but that would leave us with a problem: the parity block P0 would no longer accurately reflect the correct parity value for the stripe. Thus, in this example, P0 must also be updated. But how can we update it both correctly and efficiently?

It turns out there are two methods. The first, known as **additive parity**, requires us to do the following. To compute the value of the new parity block, read in all of the other data blocks in the stripe in parallel (in the example, blocks 0, 2, and 3) and XOR those with the new block (1). The result is your new parity block. To complete the write, you can then write the new data and new parity to their respective disks, also in parallel.

The problem with this technique is that it scales with the number of disks, and thus in larger RAIDs requires a high number of reads to compute parity. Thus, the **subtractive parity** method.

For example, imagine this string of bits (4 data bits, and one parity bit):

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1) = 0

Let's imagine that we wish to overwrite bit C2 with a new value which we will call C2(new). The subtractive method works in three steps. First, we read in the old data at C2 (C2(old) = 1) and the old parity (P(old) = 0). Then, we compare the old data and the new data; if they are the same (e.g., C2(new) = C2(old)), then we know the parity bit will also remain the same (i.e., P(new) = P(old)). If, however, they are different, then we must flip the old parity bit to the opposite of its current state, that is, if (P(old) == 1), P(new) will be set to 0; if (P(old) == 0), P(new) will be set to 1. We can express this whole

mess neatly with XOR as it turns out (if you understand XOR, this will now make sense to you):

```
P(new) = (C(old) XOR C(new)) XOR P(old)
```

Because we are dealing with blocks, not bits, we perform this calculation over all the bits in the block (e.g., 4096 bytes in each block multipled by 8 bits per byte). Thus, in most cases, the new block will be different than the old block and thus the new parity block will too.

You should now be able to figure out when we would use the additive parity calculation and when we would use the subtractive method. Think about how many disks would need to be in the system so that the additive method performs fewer I/Os than the subtractive method, and vice-versa.

For this performance analysis, let us assume we are using the subtractive method. Thus, for each write, the RAID has to perform 4 physical I/Os (two reads and two writes). Now imagine there are lots of writes submitted to the RAID; how many can RAID-4 perform in parallel? To understand, let us again look at the RAID-4 layout (Figure 37.5).

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
*4	5	6	7	+P1
8	9	10	11	P2
12	*13	14	15	+P3

Table 37.5: Example: Writes To 4, 13, And Respective Parity Blocks

Now imagine there were 2 small writes submitted to the RAID-4 at about the same time, to blocks 4 and 13 (marked with * in the diagram). The data for those disks is on disks 0 and 1, and thus the read and write to data could happen in parallel, which is good. The problem that arises is with the parity disk; both the requests have to read the related parity blocks for 4 and 13, parity blocks 1 and 3 (marked with +). Hopefully, the issue is now clear: the parity disk is a bottleneck under this type of workload; we sometimes thus call this the **small-write problem** for parity-based RAIDs. Thus, even though the data disks could be access in parallel, the parity disk prevents any parallelism from materializing; all writes to the system will be serialized because of the parity disk. Because the parity disk has

to perform two I/Os (one read, one write) per logical I/O, we can compute the performance of small random writes in RAID-4 by computing the parity disk's performance on those two I/Os, and thus we achieve (R/2) MB/s. RAID-4 throughput under random small writes is terrible; it does not improve as you add disks to the system.

We conclude by analyzing I/O latency in RAID-4. As you now know, a single read (assuming no failure) is just mapped to a single disk, and thus its latency is equivalent to the latency of a single disk request. The latency of a single write requires two reads and then two writes; the reads can happen in parallel, as can the writes, and thus total latency is about twice that of a single disk (with some differences because we have to wait for both reads to complete and thus get the worst-case positioning time, but then the updates don't incur seek cost and thus may be a better-than-average positioning cost).

37.7 RAID Level 5: Rotating Parity

To address the small-write problem (at least, partially), Patterson, Gibson, and Katz introduced RAID-5. RAID-5 works almost identically to RAID-4, except that it **rotates** the parity block across the drives (Figure 37.6).

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Table 37.6: RAID-5 With Rotated Parity

As you can see in the figure, the parity block for each stripe is now rotated across the disks, in order to remove the parity-disk bottleneck for RAID-4.

RAID-5 Analysis

Much of the analysis for RAID-5 is identical to RAID-4. For example, the effective capacity and failure tolerance of the two levels are iden-

tical. So are sequential read and write performance. The latency of a single request (whether a read or a write) is also the same as RAID-4.

Random read performance is a little better, because we can utilize all of the disks. Finally, random write performance improves noticeably over RAID-4, as it allows for parallelism across requests. Imagine a write to block 1 and a write to block 10; this will turn into requests to disk 1 and disk 4 (for block 1 and its parity) and requests to disk 0 and disk 2 (for block 10 and its parity). Thus, they can proceed in parallel. In fact, we can generally assume that that given a large number of random requests, we will be able to keep all the disks about evenly busy. If that is the case, then our total bandwidth for small writes will be $\frac{N}{4} \cdot R$ MB/s; the factor of four loss is due to the fact that each RAID-5 write still generates 4 total I/O operations.

Because RAID-5 is basically identical to RAID-4 except in the few cases where it is better, it has almost completely replaced RAID-4 in the marketplace. The only place where it has not is in systems that know they will never perform anything other than a large write, thus avoiding the small-write problem altogether [HLM94]; in those cases, RAID-4 is sometimes used as it is slightly simpler to build.

37.8 RAID Comparison: A Summary

We now summarize our simplified comparison of RAID levels in Table 37.7. Note that we have omitted a number of details to simplify our analysis. For example, when writing in a mirrored system, the average seek time is a little higher than when writing to just a single disk, because the seek time is the max of two seeks (one on each disk). Thus, random write performance to two disks will generally be a little less than random write performance of a single disk. Also, when updating the parity disk in RAID-4/5, the first read of the old parity will likely cause a full seek and rotation, but the second write of the parity will only result in rotation.

However, our comparison does capture the essential differences, and thus is useful for understanding tradeoffs across RAID levels. We present a summary in the table below; for the latency analysis, we simply use D to represent the time that a request to a single disk would take.

Thus, if you strictly want performance and do not care about reliability, striping is obviously best. If, however, you want random

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	N	$N/2$	$N-1$	$N-1$
Reliability	0	1 (for sure) $\frac{N}{2}$ (if lucky)	1	1
Throughput				
Sequential Read	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Sequential Write	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Random Read	$N \cdot R$	$N \cdot R$	$(N-1) \cdot R$	$N \cdot R$
Random Write	$N \cdot R$	$(N/2) \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4} R$
Latency				
Read	D	D	D	D
Write	D	D	$2D$	$2D$

Table 37.7: RAID Capacity, Reliability, and Performance

I/O performance and reliability, mirroring is the best; the cost you pay is in lost capacity. If capacity and reliability are your main goals, then RAID-5 is the winner; the cost you pay is in small-write performance. Finally, if you are always doing sequential I/O and want to maximize capacity, RAID-5 also makes the most sense.

37.9 Other Interesting RAID Issues

There are a number of other interesting ideas that one could (and perhaps should) discuss when thinking about RAID. Here are some things we might eventually write about:

- Other RAID levels: Levels 2 and 3 from the original taxonomy, Level 6 to tolerate multiple disk faults.
- Reconstruction: What the RAID does when a disk fails and it has a **hot spare** sitting around to fill in for the failed disk. What happens to performance under failure, and performance during reconstruction?
- More realistic fault models: Our own work on partial failures, including latent sector errors and block corruption.
- Ways of tolerating more realistic faults: Checksums and the many different approaches there. Again some of our own work.
- Software RAID: How to build the RAID as a software layer underneath the file system. Cheaper, but less reliable?

37.10 Summary

We have discussed RAID. RAID transforms a number of independent disks into a large, more capacious, and more reliable single entity; importantly, it does so transparently, and thus hardware and software above is relatively oblivious to the change.

There are many possible RAID levels to choose from, and the exact RAID level to use depends heavily on what is important to the end-user. For example, mirrored RAID is simple, reliable, and generally provides good performance but at a high capacity cost. RAID-5, in contrast, is reliable and better from a capacity standpoint, but performs quite poorly when there are small writes in the workload. Picking a RAID and setting its parameters (chunk size, number of disks, etc.) properly for a particular workload is challenging, and thus still remains more of an art than a science.

References

[BJ88] "Disk Shadowing"
D. Bitton and J. Gray
VLDB 1988
One of the first papers to discuss mirroring, herein called "shadowing".

[CL95] "Striping in a RAID level 5 disk array"
Peter M. Chen, Edward K. Lee
SIGMETRICS 1995
A nice analysis of some of the important parameters in a RAID-5 disk array.

[DAA05] "Journal-guided Resynchronization for Software RAID"
Timothy E. Denehy, A. Arpaci-Dusseau, R. Arpaci-Dusseau
FAST 2005
Our own work on the consistent-update problem. Here we solve it for Software RAID by integrating the journaling machinery of the file system above with the software RAID beneath it.

[HLM94] "File System Design for an NFS File Server Appliance"
Dave Hitz, James Lau, Michael Malcolm
USENIX Winter 1994, San Francisco, California, 1994
The sparse paper introducing a landmark product in storage, the write-anywhere file layout or WAFL file system that underlies the NetApp file server.

[K86] "Synchronized Disk Interleaving"
M.Y. Kim.
IEEE Transactions on Computers, Volume C-35: 11, November 1986
Some of the earliest work on RAID is found here.

[K88] "Small Disk Arrays - The Emerging Approach to High Performance"
F. Kurzweil.
Presentation at Sping COMPCON '88, March 1, 1988, San Francisco, California
Another early RAID reference.

[P+88] "Redundant Arrays of Inexpensive Disks"
D. Patterson, G. Gibson, R. Katz.
SIGMOD 1988
This is considered the RAID paper, written by famous authors Patterson, Gibson, and Katz. The paper has since won many test-of-time awards and ushered in the RAID era, including the name RAID itself!

[PB86] "Providing Fault Tolerance in Parallel Secondary Storage Systems"
A. Park and K. Balasubramaniam
Department of Computer Science, Princeton, CS-TR-O57-86, November 1986
Another early work on RAID.

[SG86] "Disk Striping"
K. Salem and H. Garcia-Molina.
IEEE International Conference on Data Engineering, 1986
And yes, another early RAID work. There are a lot of these, which kind of came out of the woodwork when the RAID paper was published in SIGMOD.

[S84] "Byzantine Generals in Action: Implementing Fail-Stop Processors"
F.B. Schneider.
ACM Transactions on Computer Systems, 2(2):145154, May 1984
Finally, a paper that is not about RAID! This paper is actually about how systems fail, and how to make something behave in a fail-stop manner.

Homework

This section introduces `raid.py`, a simple RAID simulator you can use to shore up your knowledge of how RAID systems work. It has a number of options, as we see below:

```
Usage: raid2.py [options]

Options:
  -h, --help            show this help message and exit
  -s SEED, --seed=SEED  the random seed
  -D NUMDISKS, --numDisks=NUMDISKS
                        number of disks in RAID
  -C CHUNKSIZE, --chunkSize=CHUNKSIZE
                        chunk size of the RAID
  -n NUMREQUESTS, --numRequests=NUMREQUESTS
                        number of requests to simulate
  -S SIZE, --reqSize=SIZE
                        size of requests
  -W WORKLOAD, --workload=WORKLOAD
                        either "rand" or "seq" workloads
  -w WRITEFRAC, --writeFrac=WRITEFRAC
                        write fraction (100->all writes, 0->all reads)
  -R RANGE, --randRange=RANGE
                        range of requests (when using "rand" workload)
  -L LEVEL, --level=LEVEL
                        RAID level (0, 1, 4, 5)
  -5 RAID5TYPE, --raid5=RAID5TYPE
                        RAID-5 left-symmetric "LS" or left-asym "LA"
  -r, --reverse         instead of showing logical ops, show physical
  -t, --timing          use timing mode, instead of mapping mode
  -c, --compute         compute answers for me
```

In its basic mode, you can use it to understand how the different RAID levels map logical blocks to underlying disks and offsets. For example, let's say we wish to see how a simple striping RAID (RAID-0) with four disks does this mapping.

```
prompt> ./raid2.py -n 5 -L 0 -R 20
...
LOGICAL READ from addr:16 size:4096
  Physical reads/writes?

LOGICAL READ from addr:8 size:4096
  Physical reads/writes?

LOGICAL READ from addr:10 size:4096
  Physical reads/writes?
```

```
LOGICAL READ from addr:15 size:4096
  Physical reads/writes?

LOGICAL READ from addr:9 size:4096
  Physical reads/writes?
```

In this example, we simulate five requests (`-n 5`), specifying RAID level zero (`-L 0`), and restrict the range of random requests to just the first twenty blocks of the RAID (`-R 20`). The result is a series of random reads to the first twenty blocks of the RAID; the simulator then asks you to guess which underlying disks/offsets were accessed to service the request, for each logical read.

In this case, calculating the answers is easy: in RAID-0, recall that the underlying disk and offset that services a request is calculated via modulo arithmetic:

```
  disk   = address % number_of_disks
  offset = address / number_of_disks
```

Thus, the first request to 16 should be serviced by disk 0, at offset 4. And so forth. You can, as usual see the answers (once you've computed them!), by using the handy `-c` flag to compute the results.

```
prompt> ./raid2.py -R 20 -n 5 -L 0 -c
...
LOGICAL READ from addr:16 size:4096
  read  [disk 0, offset 4]

LOGICAL READ from addr:8 size:4096
  read  [disk 0, offset 2]

LOGICAL READ from addr:10 size:4096
  read  [disk 2, offset 2]

LOGICAL READ from addr:15 size:4096
  read  [disk 3, offset 3]

LOGICAL READ from addr:9 size:4096
  read  [disk 1, offset 2]
```

Because we like to have fun, you can also do this problem in reverse, with the `-r` flag. Running the simulator this way shows you the low-level disk reads and writes, and asks you to reverse engineer which logical request must have been given to the RAID:

```
prompt> ./raid2.py -R 20 -n 5 -L 0 -r
...
LOGICAL OPERATION is ?
  read  [disk 0, offset 4]

LOGICAL OPERATION is ?
  read  [disk 0, offset 2]

LOGICAL OPERATION is ?
  read  [disk 2, offset 2]

LOGICAL OPERATION is ?
  read  [disk 3, offset 3]

LOGICAL OPERATION is ?
  read  [disk 1, offset 2]
```

You can again use `-c` to show the answers. To get more variety, a different random seed (`-s`) can be given.

Even further variety is available by examing different RAID levels. In the simulator, RAID-0 (block striping), RAID-1 (mirroring), RAID-4 (block-striping plus a single parity disk), and RAID-5 (block-striping with rotating parity) are supported.

In this next example, we show how to run the simulator in mirrored mode. We show the answers to save space:

```
prompt> ./raid2.py -R 20 -n 5 -L 1 -c
...
LOGICAL READ from addr:16 size:4096
  read  [disk 0, offset 8]

LOGICAL READ from addr:8 size:4096
  read  [disk 0, offset 4]

LOGICAL READ from addr:10 size:4096
  read  [disk 1, offset 5]

LOGICAL READ from addr:15 size:4096
  read  [disk 3, offset 7]

LOGICAL READ from addr:9 size:4096
  read  [disk 2, offset 4]
```

You might notice a few things about this example. First, the mirrored RAID-1 assumes a striped layout (which some might call RAID-01), where logical block 0 is mapped to the 0th block of disks 0 and 1, logical block 1 is mapped to the 0th blocks of disks 2 and 3, and

so forth (in this four-disk example). Second, when reading a single block from a mirrored RAID system, the RAID has a choice of which of two blocks to read. In this simulator, we use a relatively silly way: for even-numbered logical blocks, the RAID chooses the even-numbered disk in the pair; the odd disk is used for odd-numbered logical blocks. This is done to make the results of each run easy to guess for you (instead of, for example, a random choice).

We can also explore how writes behave (instead of just reads) with the `-w` flag, which specifies the "write fraction" of a workload, i.e., the fraction of requests that are writes. By default, it is set to zero, and thus the examples so far were 100% reads. Let's see what happens to our mirrored RAID when some writes are introduced:

```
prompt> ./raid2.py -R 20 -n 5 -L 1 -w 100 -c
...
LOGICAL WRITE to  addr:16 size:4096
  write [disk 0, offset 8]     write [disk 1, offset 8]

LOGICAL WRITE to  addr:8 size:4096
  write [disk 0, offset 4]     write [disk 1, offset 4]

LOGICAL WRITE to  addr:10 size:4096
  write [disk 0, offset 5]     write [disk 1, offset 5]

LOGICAL WRITE to  addr:15 size:4096
  write [disk 2, offset 7]     write [disk 3, offset 7]

LOGICAL WRITE to  addr:9 size:4096
  write [disk 2, offset 4]     write [disk 3, offset 4]
```

With writes, instead of generating just a single low-level disk operation, the RAID must of course update both disks, and hence two writes are issued. Even more interesting things happen with RAID-4 and RAID-5, as you might guess; we'll leave the exploration of such things to you in the questions below.

The remaining options are discovered via the help flag. They are:

```
Options:
  -h, --help            show this help message and exit
  -s SEED, --seed=SEED  the random seed
  -D NUMDISKS, --numDisks=NUMDISKS
                        number of disks in RAID
  -C CHUNKSIZE, --chunkSize=CHUNKSIZE
                        chunk size of the RAID
  -n NUMREQUESTS, --numRequests=NUMREQUESTS
```

```
                        number of requests to simulate
-S SIZE, --reqSize=SIZE
                        size of requests
-W WORKLOAD, --workload=WORKLOAD
                        either "rand" or "seq" workloads
-w WRITEFRAC, --writeFrac=WRITEFRAC
                        write fraction (100->all writes, 0->all reads)
-R RANGE, --randRange=RANGE
                        range of requests (when using "rand" workload)
-L LEVEL, --level=LEVEL
                        RAID level (0, 1, 4, 5)
-5 RAID5TYPE, --raid5=RAID5TYPE
                        RAID-5 left-symmetric "LS" or left-asym "LA"
-r, --reverse           instead of showing logical ops, show physical
-t, --timing            use timing mode, instead of mapping mode
-c, --compute           compute answers for me
```

The `-C` flag allows you to set the chunk size of the RAID, instead of using the default size of one 4-KB block per chunk. The size of each request can be similarly adjusted with the `-S` flag. The default workload accesses random blocks; use `-W sequential` to explore the behavior of sequential accesses. With RAID-5, two different layout schemes are available, left-symmetric and left-asymmetric; use `-5 LS` or `-5 LA` to try those out with RAID-5 (`-L 5`).

Finally, in timing mode (`-t`), the simulator uses an incredibly simple disk model to estimate how long a set of requests takes, instead of just focusing on mappnings. In this mode, a "random" request takes 10 milliseconds, whereas a "sequential" request takes 0.1 milliseconds. The disk is assumed to have a tiny number of blocks per track (100), and a similarly small number of tracks (100). You can thus use the simulator to estimate RAID performance under some different workloads.

Questions

1. Use the simulator to perform some basic RAID mapping tests. Run with different levels (0, 1, 4, 5) and see if you can figure out the mappings of a set of requests. For RAID-5, see if you can figure out the difference between left-symmetric and left-asymmetric layouts. Use some different random seeds to generate different problems than above.

2. Do the same as the first problem, but this time vary the chunk size with `-C`. How does chunk size change the mappings?

3. Do the same as above, but use the `-r` flag to reverse the nature of each problem.

4. Now use the reverse flag but increase the size of each request with the `-S` flag. Try specifying sizes of 8k, 12k, and 16k, while varying the RAID level. What happens to the underlying I/O pattern when the size of the request increases? Make sure to try this with the sequential workload too (`-W sequential`); for what request sizes are RAID-4 and RAID-5 much more I/O efficient?

5. Use the timing mode of the simulator (`-t`) to estimate the performance of 100 random reads to the RAID, while varying the RAID levels, using 4 disks.

6. Do the same as above, but increase the number of disks. How does the performance of each RAID level scale as the number of disks increases?

7. Do the same as above, but use all writes (`-w 100`) instead of reads. How does the performance of each RAID level scale now? Can you do a rough estimate of the time it will take to complete the workload of 100 random writes?

8. Run the timing mode one last time, but this time with a sequential workload (`-W sequential`). How does the performance vary with RAID level, and when doing reads versus writes? How about when varying the size of each request? What size should you write to a RAID when using RAID-4 or RAID-5?