Simple, Robust Software RAID for Linux 2.6

Daniel Phillips

27th June 2003

Abstract

Linux's new Device Mapper subsystem provides efficient facilities for concatenating, striping and mirroring physical volumes into a single logical volume, but support of RAID level 5 is left to the existing Multiple Device subsystem. Though the Device Mapper and Multiple Device subsystems can be combined to work around this problem, this requires extra administration work, adds an extra level of processing overhead, and does not satisfy Device Mapper's original goal of simplicity. A new RAID plug-in for Device Mapper is introduced here to provide RAID5-like functionality for physical volume configurations consisting of 2^k + 1 disks, where each logical block is split across 2^k disks, and parity information is written to the remaining disk. This strategy resembles the old RAID 3 strategy, and avoids one of the greatest sources of complexity in RAID 5, which is the need to read before writing in order to update parity information. This in turn removes the need for an extra block cache in the RAID driver. With the IO path thus simplified, performance for largely sequential write loads can approach the combined IO bandwith of the physical devices. Versus RAID 5, random IO loads generate higher seeking and lower rotational latency penalties, so random IO performance remains acceptable.

1 Introduction

Linux's new Device Mapper subsystem provides efficient facilities for concatenating, striping and mirroring physical volumes into a single logical volume. Device-mapper currently supports a striped target (RAID 0), and by the time 2.6 is released, a mirrored target (RAID 1). However, device-mapper does not directly support block device redundancy beyond simple mirroring, that is, device-mapper has no RAID 5 or similar target. One way to provide such support is by running device-mapper underneath the existing Multiple Device subsystem, as a virtual block device. This is not entirely satisfactory due to the extra overhead introduced by a second level of virtualization. Such a combination would also require an extra level of configuration and administration. Finally, it would fall short of satisfying Device Mapper's original goal of simplicity.

A new RAID plug-in for Device Mapper is introduced here to provide funcionality similar to RAID5, but with a considerably simpler implementation. This is accomplished by restricting the allowed number of physical drives to 2^k + 1. For example, 3 drive, 5 drive and 9 drive arrays would be supported while 6 drive arrays would not be.

2 Terminology

There are many possible ways of distributing data amongst the drivers of a RAID array. A particular RAID scheme is typically called a "RAID level", as if each possible RAID level were a subset of the next higher

level. This is not strictly true. For example, RAID 0[1] simply distributes data sectors across an array of drives with no redundancy, while RAID 1[2] stores an identical copy of each sector on both drives of a two-drive array. RAID 0 is clearly not a subset of RAID 1, since a RAID 0 array can have many drives while a RAID 1 array can have only two. In practice, RAID level numbers tend to reflect the difficulty of implementing each scheme more than anything else.

There exists some confusion over the use of the term "stripe". Properly speaking, it's a continguous run of blocks on one drive. Sometimes taken to mean a set of sectors, one on each of (n-1) drives in the RAID set, each contributing to a parity sector on the remaining drive. To avoid confusion, I use the term "parity row" to mean a set of data sectors stored on N-1 drives of a RAID array, contributing to a parity sector stored on the remaining drive.

The term "interleave" was orginally used to describe the practice of stepping the physical sector number of the starting logical sector on each track of a hard disk, so that the time required to step the head from one track to another is less than the time for the first logical sector of the new track to rotate under the read head after the last logical sector of the previous track is read. This term was repurposed as part of RAID terminology to describe the practice of stepping the drive number of the starting sector of each parity row. We can call the latter "drive interleave" as opposed to the former "sector interleave". In this sense, RAID 5 has a drive interleave of 1 or -1, while RAID 4 has a driver interleave of 0, that is, it is not interleaved.

I now introduce a new term, "band", meaning a contiguous set of data sectors distributed across a raid array having the same interleave offset, together with their parity blocks. In other words, each time the interleave offset changes, we step to a new band.

We say that RAID 4 and RAID 5 are striped, while RAID 0, 1 and 2 are not.

RAID 3.5, which I introduce in this paper is, following the above terminology, a non-striped distributed parity scheme with band-wise interleaving.

3 Taxonomy of RAID Schemes

In this section we examine a few of the existing RAID schemes that are related more or less closely to the RAID 3.5 I propose, or which might in some sense compete with the new scheme. This includes obsolete RAID levels 3 and 4 (with RAID 2 making a cameo appearance) and the widely used RAID 5. RAID 0 is not discussed here, as it offers no redundancy. RAID 1 is not discussed because its approach to data redundancy - replicating data verbatum across multiple volumes - is clearly less efficient in terms of storage size and transfer bandwidth than most higher RAID levels.

The main question we intend to answer in this section is: is the RAID scheme proposed here actually novel, or is it simply a special case of an existing scheme? (Below, I conclude that it is in fact novel, i.e., it is not a subset of any existing RAID level.)

3.1 RAID 2 and RAID 3

RAID 3[3] is a simplified version of RAID 2 [ref http://www.acnc.com/04_01_02.html]. Like RAID 2[4], RAID 3 uses a stride of a N bytes, where N is the number of drives in the array. In places of RAID 2's ECC codes, RAID 3 uses a simple XOR. This RAID 3 drive array has N-1 drives for data, with the remaining drive storing exclusively parity. With RAID 3, all drives rotate in synchronization, which requires hardware support that not available on many drives. Hardware support is also required to divide blocks

up between the drives, and generate the parity bytewise. RAID 3 has traditionally been thought to suffer from performance bottlenecks under random access loads. Because of that and its specialized hardware requirements, it is no longer commonly used.

RAID 3.5 strongly resembles RAID 3, except that the bytes of each filesystem block are divided differently between the physical drive sectors, and the parity sectors differ as a result. In other words, given the restriction of 2^k+1 drives and band interleave of 0, RAID 3 and RAID 3.5 use the same locations on the drive array for each transfered block, but split the data differently between those locations. (RAID 3 uses a stride of N-1 whereas RAID 3.5 uses a stride of B/(N-1) bytes, where N is the number of drives and B is the transfer blocksize.) Thus, it is not possible to configure a RAID 3 system to read a volume formatted as RAID 3.5, and consequently, RAID 3.5 is not simply a software implementation of RAID 3.

A further difference between RAID 3 and RAID 3.5 is the provision for bandwise interleaving, mentioned in this paper but not described in detail.

In some sense, RAID 5 is the chief "competitor" to the RAID 3.5 scheme introduced here.

3.2 RAID 4

RAID 4 uses a data striping arrangement (as in RAID 0 configurations, or "striped targets" in device-mapper terminology) with parity stored on a dedicated parity drive. RAID 4 arrays tend to suffer from performance problems under many loads, due to the need to update the parity drive on every write.

RAID 3.5 resembles a specific configuration of RAID 4, where stripe size has been set to, e.g., 1K on a 5 drive system. However, as described below, RAID 3.5 supports band-level interleaving, which is not possible with RAID 4 as defined, so RAID 3.5 is not a subset of RAID 4. Furthermore, although in practical terms, a RAID 4 implementation could be configured to access a non-interleaved RAID 3.5 volume, performance would be very poor, as the algorithm used for IO transfers differs dramatically between RAID 4 and RAID 5. In theory, a RAID 4 implementation could be improved with cross-stripe optimization, in which case it could perform identically to RAID 3.5 in the cases where RAID 3.5 performs well. Band level interleaving could also be added to RAID 4, however what we would have at that point is a superset of a subset of the original RAID 4.

3.3 RAID 5

RAID5[7] uses both data striping (as in RAID 0) and parity (as in RAID 4). The difference between RAID level 5 and RAID level 4 is the interleave factor, that is, parity is stored on a different drive in each band, which distributes the parity update load across the entire array. In practice, this gives RAID 5 such a great advantage over RAID 4 that the latter has fallen into disuse, and is seldom even implemented on software or hardware RAID systems.

As with RAID 4, a RAID 5 array can be configured to resemble a RAID 3.5 array closely, by setting the stripe size to a binary fraction of the virtual volume's filesystem block size. The only remaining difference would be the interleaving: RAID 5 would store the parity information for each adjacent filesystem block on a different drive, while RAID 3.5 stores the parity information on the same drive, within each band.

4 RAID 3.5

A new RAID scheme, RAID 3.5 is introduced here. RAID 3.5 is restricted to RAID arrays consisting of 2^k + 1 driver, where each logical block is split evenly across 2^k drives, and a correspondingly sized block of parity information is written to the remaining drive. This strategy resembles the obsolete RAID 3. However, it avoids one of the greatest sources of complexity in RAID 5, which is the need to read before writing in order to update parity information. This in turn removes the need for an extra block cache in the RAID driver. With the IO path thus simplified, performance for largely sequential write loads can approach the combined IO bandwidth of the physical devices. Versus RAID 5, random IO loads generate higher seeking and lower rotational latency penalties, so random IO performance remains acceptable.

[add: gives best parallel transfer rate with no alignment problems at expense of worst transaction parallelizing. Alignment problems force read-before-write which introduces full-rotation delays.]

Why is it called RAID 3.5? Because it is very similar to both RAID 3 and RAID 4, but it is neither identical to or a subset of either, hence RAID 3.5. Specifically, RAID 3.5 has the data blocking scheme of RAID 3 but the data is distributed within a row like RAID 4. This scheme turn out to have certain practical advantages, among them simplicity and parsimonious use of drive bus bandwidth.

Why has this arrangement not been considered before? First, because it is no means certain that performance at or near the level of RAID 5 will be achieved. However, as I illustrate below, it does perform much better than one would expect.

An defining characteristic of RAID 3 and RAID 3.5 is that all drives seek in unison, since each filesystem block (including parity) is split across all drives. In the case of RAID 3, it was thought that this property would introduce an unacceptable performance loss in the form of nearly an entire rotation (about 6 ms) per read or write transfer. This is incorrect: in a sequence of transfers, only the first is likely to incur nearly a full revolution penalty; the rotational latency of succeeding seeks will tend to overlap.[1] It was also thought that hardware synchronization would be required to avoid the aforementioned rotational latency penalty, but we see now that this is not necessary, though it would indeed be useful.

A further performance problem previously thought to be associated with RAID 3 is the fact that it is unable to process more than a single transfer at a time, since all (or all but one in the read case) of the drives in the array are needed for each transfer. As we shall see, RAID 5 suffers different, but nearly as severe performance problems of its own, the end result being that this is not nearly as big a problem as it appears.

RAID 3.5 has a number of advantages over RAID, not the least being that it is very simple, and so a reliable implementation can be built using much less code and effort than for RAID 5. Once it is working, there is very little that can go wrong with it: it either works or it doesn't, and has far fewer corner cases to handle then (an efficient implementation of) RAID 5 does.

RAID 3.5 is an optimal strategy for loads that involve mainly large, serial transfers. As such, a RAID 3.5 array would be an attractive replacement for a traditional tape backup system, delivering far higher throughput and much better data safety.

Finally, though it is not discussed here, there is a small embellishment to the RAID 3.5 strategy which uses bandwise interleaving to achieve steady state read throughput equal to the combined bandwidth of all the drives in the array, as opposed to RAID 5, which can achieve the bandwidth of all the drives in the array, less one, at most.

[1] This performance characteristic depends on keeping the device queue full, which is normal behaviour with SCSI disk systems under load. IDE systems with writeback caching enabled can approximate this behaviour as well. Even when no transaction queuing is available on the device, the block IO approximates it by operating its own queue asynchronously with respect to applications, so that delays between transfers are kept as short as possible, given the hardware limitations.

5 Implementation

The RAID 3.5 target is just another target, much like the "striped" target, and is in fact derived from it. An implemenation of RAID 3.5 is in progress, and should be available by the time this paper is presented.

5.1 Efficient BIO generation

A naive implementation of RAID 3.5 would simply generate T*2^k physical transfers for every logical transfer, where T is the number of blocks in the transfer. This would generate an unacceptable amount of device bus traffic. Instead, we want to make scatter-gather DMA hardware do most of the work for us. The new BIO (Block IO) layer in Linux 2.6 supports this well. Each BIO object defines one transfer, possibly consisting of many regions contiguous at the destination but disjoint at the source. Thus, each source BIO object received by the logical device generates N BIO transfers, one to each of the target drives, and each of these BIO objects contains T transfer regions of size B/2^k, where B is the filesystem blocksize. The low-level device driver will ultimately translate these BIO regions into scatter-gather regions, effecting the transfer as efficiently as possible.

5.2 Partial block transfers

With most filesystems, transfers having sub-block size or alignment almost never occur. With suitable attention to the alignment of volume partitions, we might use the word "never" here. However, it is possible that, for example, when performing the initial load of the superblock, a non-aligned transfer might be required. It is not necessary to support such and operation efficiently, however, the RAID3.5 implementation would be more robust if the operation simply ran slowly, rather than failing outright. This is easily accomplished: when a subblock transfer is called for, a slow path is used....

Ext2, Ext3 and ReiserFS never perform subblock transfers[except possibly for the superblock read/write?], the former because it does not support UFS-style fragments, and the latter because it uses tailmerging as an alternative to fragment processing, which works in units of full blocks.

On the other hand, XFS currently works in extents, with a granularity of 512 bytes. However, a new mkfs option to specify a coarser XFS granularity has already been implemented and is currently being tested, so new versions of XFS should also work well with RAID 3.5. [2]

Even UFS, if it were implemented on Linux, would work in certain configurations. For example, with a 2K fragment size on a five drive array, or 1K fragment size with three drives.An implementation of RAID 3.5 is in progress. At time of writing, the necessary XOR processing has been written and benchmarked, and found to perform satisfactorily. The author's test system consists of an array of 5 scsi drives each capable of transferring 40 MB/s, connected to an ultra scsi bus supporting up to 160 MB/s of parallel transfers. Raw throughput without any raid processing on this system was found to approach 133 MB/s. Thus, and aggressive performance level of greater than 100 MB/s, read or write, has been set as the goal for The target performance level for the finished RAID 3.5 system.

5.3 Examples

Here are some concrete examples of how data is laid according to various RAID configurations. For the RAID 3.5 and RAID 5 examples, the overhead involved in carrying out a small write transfer is examined.

[2] Some operating environments, particularly mainframes, do not support a transfer granularity as fine as 512 bytes.

The write example is intentionally concocted not only to be favorable to RAID 3.5, but to draw attention to the unintuitive result that sometimes RAID 3.5 is faster than RAID 5, by a wide margin.

5.3.1 Example 1: Raid 3

For the purpose of the examples below, we assume a 5 disk array for now, with 4KB transfer units.

5.3.2 Example 1: Raid 3

Each row has four bytes of date, one byte of parity:

Drive0	Drive1	Drive2	Drive3	Drive4
a:1/4096	a:1/4096	a:1/4096	a:1/4096	parity(a0..a3)
a:1/4096	a:5/4096	a:6/4096	a:7/4096	parity(a3..a7)
...				
b:0/4096	b:1/4096	b:2/4096	b:2/4096	parity(b0..b3)
...				
c:0/4096	...			
...				
d:0/4096	...			

But in practice, it's really stored on the drive in sectors. So assuming a 512 byte sector, we have:

5.3.3 Example 2: Raid 3

Each row has 2048 bytes and 512 bytes of parity:

Drive0	Drive1	Drive2	Drive3	Drive4
a:0/8	a:1/8	a:2/8	a:3/8	parity(a0/8..a3/8)
a:3/8	a:4/8	a:5/8	a:6/8,	parity(a3/8..a7/8)
b:0/8	b:1/8	b:2/8	b:3/8	parity(b0/8..b3/8)
b:3/8	b:4/8	b:5/8	b:6/8	parity(b3/8..b7/8)
c:0/8	c:1/8	c:2/8	c:3/8	parity(c0/8..c3/8)
c:3/8	c:4/8	c:5/8	c:6/8	parity(c3/8..c7/8)
d:0/8	d:1/8	d:2/8	d:3/8	parity(d0/8..d3/8)
d:3/8	d:4/8	d:5/8	d:6/8	parity(d3/8..d7/8)

Each 512 byte data sector has every fourth byte of the original data, in other words, there is no change from Example 1 except for the grouping, which just reflects how the data is actually transfered to the drive.

On a (less common) hard disk with 1024 byte sectors, we would have:

5.3.4 Example 3: Raid 3

Each row has 4098 bytes of data and 1024 bytes of parity:

Drive0	Drive1	Drive2	Drive3	Drive4
a:0/4	a:1/4	a:2/4	a:3/4	parity(a)
b:0/4	b:1/4	b:2/4	b:3/4	parity(b)
c:0/4	c:1/4	c:2/4	c:3/4	parity(c)
d:0/4	d:1/4	d:2/4	d:3/4	parity(d)

where a0/4 has every fourth byte of a modulo 0, a1/4 has every fourth byte of a modulo 1, and so on.

5.3.5 Example 4: Raid 3.5

Each row has one filesystem block and parity for that block:

Drive0	Drive1	Drive2	Drive3	Drive4
a:0/4	a:1/4	a:2/4	a:3/4	parity(a)
b:0/4	b:1/4	b:2/4	b:3/4	parity(b)
c:0/4	c:1/4	c:2/4	c:3/4	parity(c)
d:0/4	d:1/4	d:2/4	d:3/4	parity(d)

where a0/4 has the first 1/4 block block a, b0/4 has the second 1/4 block block b and so on.

Now, the only difference between raid 3.5 and raid 3 is how the bytes of one block are divided between the drives, and how the parity is calculated. The diagrams are identical. So when we look at it this way, raid 3.5 is almost the same as raid3, except for partitioning the bytes of the block differently between the data drives, and the parity also being different as a result. Therefore, the actual data on the drives won't be compatible between raid 3 and raid 3.5, though the data is shared between the drives in an almost identical way.

Say we want to write blocks b and c. The operations are:

Drive0	Drive1	Drive2	Drive3	Drive4
*write b:0/4	*write b:1/4	*write b:1/4	*write b:1/4	*write b:1/4
write c:0/4	write c:0/4	write c:0/4	write c:0/4	write c:0/4

* Latency due to seek time and 1/2 rotation on average

Assuming seek time is 3 ms and rotational period is 6 ms, the transaction finishes in 6 ms. Total bus traffic is two blocks.

5.3.6 Example 5: Raid 4

Each row has four filesystem blocks and parity for those four blocks.

If stripe size is one block:

Drive0	Drive1	Drive2	Drive3	Drive4
a	b	c	d	parity(a, b, c, d)

To illustrate how the pattern continues with a stripe size of two blocks:

Drive0	Drive1	Drive2	Drive3	Drive4
a	c	e	g	parity(a, c, e, g)
b	d	f	h	parity(b, d, f, h)

5.3.7 Example 6: Raid 5

Each row has four filesystem blocks and parity for those four blocks

With one block stripes:

Drive0	Drive1	Drive2	Drive3	Drive4
a	b	c	d	parity(a, b, c, d)
parity(e, f, g, h)	e	f	g	h

The first row is identical to RAID 4; the second is shifted over by one drive.

With negative interleave the shift is in the other direction:

Drive0	Drive1	Drive2	Drive3	Drive4
a	b	c	d	parity(a, b, c, d)
e	f	g	parity(e, f, g, h)	h

Finally, a stripe size of two blocks shows how the pattern develops:

Drive0	Drive1	Drive2	Drive3	Drive4
a	c	e	g	parity(a, c, e, g)
b	d	f	h	parity(b, d, f, h)
i	k	m	parity(i, k, m, o)	o
j	l	n	parity(j, l, n, p)	p

With RAID 5, stripe size is very important in determining performance characteristics under different loads.A small stripe size is better for random transactions because it increases the chance that full parity rows can be written, avoiding the need to read before writing. However, a small stripe size is not very good for larger transfers because the transfer will not lie entirely on one disk, and more of the disk heads will need to seek to perform the transfer. Clearly, there is no one best stripe size for all loads. It is apparent that when the stripe size happens to be inappropriate for a particular load that either seeking will increase and extra reads will be performed, which one again shows that RAID 5 does not have as much of an advantage over RAID 3.5 as it would first appear.

Say a transfer writes blocks b and c. The operations performed are:

Drive0	Drive1	Drive4
*read b	*read c	*read parity(a, c, e, g)
**write b	**write c	read parity(b, d, f, h)
		**write parity(a, c, e, g)
		write parity(b, d, f, h)

* Delay of seek time plus 1/2 rotation on average

** Delay of seek time plus a full rotation

Assuming seek time is 3 ms and rotation time is 6 ms, the transaction is finished in 9 ms. Total bus traffic is eight blocks. So in this case, RAID 3.5 soundly trounces RAID 5, which is the surprising result promised above.

Note that RAID 3.5 will by no means always perform better than RAID 5. In fact, I expect that when the benchmark results come in, we will see that a good RAID 5 implementation does in fact perform as well or better than a RAID 3.5 implementation under typical loads. However, the difference will not be nearly as great as one would expect. The RAID 3.5 implementation will surely be far simpler than the RAID 5 implementation, and thus both more robust and easier to optimize.

6 Further work

Besides completing and benchmarking the current design, a number of incremental improvements are possible. First, the restriction of 2^k + 1 drives can be removed. Second, read transfers can be optimized as described above in the text. Third, a procedural interface for scatter-gather region generation could be implemented in the block layer and device drivers, reducing the amount of storage required for the BIO transfers.

References

[1] http://osr5doc.ca.caldera.com:457/OSAdminG/vdmC.vdtypes.html#vdmC.raid0

[2] http://osr5doc.ca.caldera.com:457/OSAdminG/vdmC.vdtypes.html#vdmC.raid1

[3] http://www.robustdigitalsolutions.com/html/raid-3.html

[4] http://www.acnc.com/04_01_02.html

[5] http://osr5doc.ca.caldera.com:457/OSAdminG/vdmC.vdtypes.html#vdmC.raid4

[7] http://osr5doc.ca.caldera.com:457/OSAdminG/vdmC.vdtypes.html#vdmC.raid5