Software RAID in 2.4

Neil Brown

December 20, 2000

Abstract

With 2.4.0, software RAID in Linux will come of age. The functionality that previously was only available through a separate patch set will finally be part of a "Stable" kernel release. However for much (hopefully not all) of the 2.4.0-test series, RAID5 performance was extremely poor.

This talk will look at how differences between 2.2 and 2.4 affected software RAID, and how the various issues were (finally) dealt with. In particular, it will look at RAID5 performance and what slowing things down.

1 Introduction

The Linux Kernel contains a device driver call "MD" for "Multiple Disk". It allows several disc drives (or other devices) to be joined together to give the appearance of a single devices that is either larger, or more reliable, or both. The virtual device presented is often referred to as a RAID or Redundant Array of Independent Drives.

When the then-latest version of the MD driver (also referred to as Software Raid) was incorporated into the 2.3 development series of the Linux kernel, it had some functionality problems and the RAID5 module also had substantial performance problems. These were largely due to changes in other parts of the Linux kernel that were made early in the 2.3 series.

The rest of this paper takes a brief look at the history of Software Raid in the Linux kernel, and then looks at the specific changes that causes problems, why they caused problems, and what was done about it.

2 History

The MD (multiple disk) driver which provides the Software RAID functionality first appeared in the Linux kernel in release 1.3.69. This was version 0.34 of the MD driver and only supported the `RAID0` and `linear` (drive concatenation) personalities. The version number increased to 0.35 in 1.3.93 with some substantial changes to the interface between the MD driver and the block-device subsystem (`ll_rw_blk.c`).

In release 2.1.63, and then 2.0.35 (for the stable series), RAID1 and RAID5 personalities were added as part of an MD driver upgrade to 0.36. The modules provided basic support for these RAID arrays including survival of a drive failure, but they did not include online rebuilding of redundancy. To rebuild a drive or check parity for RAID5, the array would have to be taken off-line, and a user-level program run to for the work.

For some time after this, Ingo Molnar maintained a set of RAID patches separate from the main kernels. This patch set gained many improvements including on-line rebuild, a more robust super-block format, and other structural improvements. However it was never included in the 2.2 series kernels, partly because it made some changes to the interface with user-level tools which was not backwards compatible. Nevertheless, many people used this patch set and were very happy with it. When discussing 2.2 RAID performance below, it will always be with reference to 2.2 with this patch set applied. This RAID code is version 0.90 of the MD driver.

Much of Ingo's work was incorporated into the development kernel in 2.3.43. However on-line rebuilding did not work for reasons which are explained below. In 2.3.99pre8, new rebuild code that I had written was incorporated, and a fairly mature RAID subsystem was finally available in a `kernel.org` kernel.

Since them, numerous small improvements and bug fixes have gone in, and it is very likely that we will have a fully functional and efficient RAID subsystem in 2.4.0-final.

3 The buffer cache gives way to the page cache

One of the significant changes between 2.2 and 2.4 happened quite early in the 2.3 development series. This involved the cache in which filesystems stored data.

In 2.2, filesystem data was stored in the buffer cache. This cache is indexed by device number and physical block on that device. To find a particular block of a file in the buffer cache, you have to look up the file's meta-data to find where on the device that block is, and then look up that address in the buffer cache.

In 2.4, filesystem data has largely been moved into the page cache. The page cache is indexed by an "address_space" and an offset within that address_space. An address_space typically corresponds to a file. Each inode contains a pointer to the associated address space. This allows direct access to a particular block of a file, rather than having to look up addresses in the file's meta-data.

Some filesystem data, particularly meta-data does not fit neatly into the address_space concept, and this is still kept in the buffer cache. However meta-data is only a small fraction of the data that a filesystem caches, so in 2.4 very little filesystem data is stored in the physically addressable buffer cache.

This has two effects on the RAID code, one that relates to RAID5 only, and one that relates to both RAID1 and RAID5 — the two RAID personalities that provide redundancy.

3.1 RAID5 snooping for write performance

In RAID5, blocks of data are logically gathered, one from each device, into stripes. For each stripe, one block is deemed the "parity" block. All other blocks store real data. The parity block stores a logical "exclusive-or" of the data blocks. Thus if any one device is lost, the data that was on that device can be discovered using the data on the working drives, and the parity block.

Thus, whenever updating any data block, the parity block in the same stripe must be updated as well. A bit simplisticly, this leads to two physical up-dates for each logical update.

If several logical updates within the same stripe can be gathered together and performed at the same time, then only one parity update is needed for the whole stripe. Though this is a bit of an over-simplification, it is clear that gathering updates for a given stripe together will improve write performance.

The RAID5 code for 2.2 did this by looking in (snooping on) the buffer cache. As the buffer cache is physically addressed, when RAID5 needed to write a block in a given stripe, it could first check the buffer cache for any other blocks in that stripe which were marked dirty — but not yet scheduled for writing — and proactively write them. This lead to substantial speed-ups in write throughput.

While the buffer cache is still present in 2.4, and the RAID5 code can still look in it for dirty data, not much is actually stored in the buffer cache, so RAID5 will not often find anything.

This means that an optimisation that was very effective in 2.2, has become effectively useless in 2.4. A different approach to clustering writes needs to be found.

3.2 Rebuilding redundancy through the buffer cache

The second way that the RAID code for 2.2 utilised the buffer cache was when rebuilding redundancy, whether copying a mirror to a spare, or re-calculating parity on a RAID5 array, or when reconstructing a failed drive onto a spare for RAID5.

The basic approach that was taken for rebuilding redundancy was to read and re-write the entire array. The read process would find the correct data, either by reading from a valid drive or by reading all other drives in a RAID5 stripe and calculating the missing block. The write process would then update all drives appropriately.

During this process there is obviously a time lag between reading the data and writing it back out again. If, during that time lag, an active file system that was using the RAID array chose to write out new data, there is a potential for the old data written by the rebuilding process to over-write the new data written by the filesystem. That would obviously be a problem.

In 2.2, this potential problem was not realised because the buffer cache provided the necessary synchronisation. If a filesystem chose to write to a block that was being used in the rebuild process, it would write to exactly the same buffer that the rebuild process was using, so there would be no risk of over-writing new data.

At least, that was true for filesystems. For RAID arrays used for swapping the same synchronisation was not available. Even in 2.2, swap I/O did not go through the buffer cache, so swapping to a RAID array that was in the process of rebuilding could cause data loss. This lead to a strong recommendation that RAID NOT be used for swap, or if it was, swap should not be enabled on a RAID array that was being rebuilt.

For 2.4, the rebuild process has been substantially re-written. Much of the detail of the process has been pushed down into the individual personalities (RAID1.c and RAID5.c) which are in a better position to synchronised things properly.

For RAID1, the possibility of over-writing new data with old is avoided by creating a small sliding window in which the re-build happens. Normal I/O that is targeted within the window is blocked until the window has moved on. The leading edge of the window does not progress until it can be sure that there is no outstanding I/O requests in the next section, and the trailing edge does not progress until all rebuild requests in the tail section have completed.

For RAID5, the synchronisation was somewhat easier to achieve, though possibly not as easy to describe. RAID5 maintains a "stripe cache" which buffers some or all of the data in recently accessed stripes. Each new I/O request is attached to the appropriate stripe, and that stripe is handled as a whole. In the same way, rebuild requests are attached to the appropriate stripe, and all reading, writing, and rebuilding of a stripe are handled together, thus avoiding possible conflict.

4 The Big Kernel Lock gives way to finer grains

The second big change that came during the 2.3 development series was the reduction in using the Big Kernel Lock (BKL) to protect code against concurrent execution on SMP machines.

In 2.2, all file I/O was, by default, executed under the BKL so that there were no SMP related concurrency issues. All that changed in 2.3, and it became required for each filesystem and each device driver to do its own locking as required.

This was not much of an issue for the linear, RAID0, and RAID1 personalities, as they treat each request independently. For the most part, the only data-structures that are shared by different requests are read-only data-structures that describe the whole array. Even where shared write access is required, such as allocating buffer_heads from a pool in RAID1.c, a simple spin lock around the code is adequate.

However with RAID5, the situation is much more complex. As mentioned earlier, RAID5 maintains a cache of recently accessed stripes, and attaches new requests to the appropriate stripe. This means that access to each stripe needs to be suitably protected with a lock of some sort.

Further, each stripe may be in one of various states, possibly reading old data to perform a parity calculation, or reading data to service requests, or writing out new data and parity. Establishing the best model to accommodate all of these states and all possible transitions is not straight forward.

At the time when all these changes were happening in 2.3, Ingo Molnar, the maintainer of the RAID patches for 2.2, was busy with other interesting things, particular the Tux http server. Though he obviously wanted to keep RAID5 working in the 2.3 kernels, he did not (as far as I can tell) have the time to give it the attention that it needed. While he managed to successfully port the RAID code into 2.3, the locking scheme for the RAID5 stripe was imperfect, as we shall see in the next section.

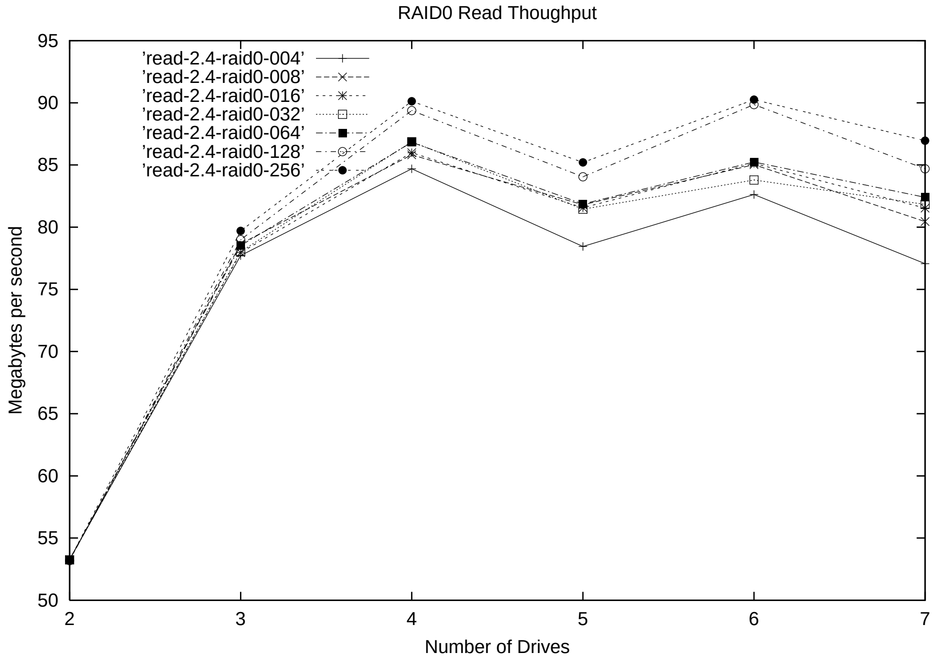

Figure 1: RAID0 Read Throughput in 2.4 with various chunk sizes

5 What went wrong with RAID5

Though the software RAID code was successful ported to the 2.3 development kernels, and was included in the kernel.org releases from about 2.3.50 onwards, performance for RAID5 was quite poor. In this section I will look at RAID5 performance, why it was poor, and my experience in trying to improve it.

But first, it will pay to look a bit at how I chose to measure performance.

Ultimately, the only measure of performance that is important is "How quickly does it do the job that I want it to do?" However that is hard to measure or control reliably. Instead, I followed that example of Gary Murakami[1] and ran a few simple tests over a variety of configurations. This allowed me so see how different aspects of the configuration affected different aspects of performance.

The different configurations that I tested were arrays with between 2 and 7 drives, and using a chunk size ranging from 4k to 256K in powers of 2. For each of these configurations I measured sequential read and write performance using bonnie and random filesystem performance using dbench. I did this on both RAID5 configurations and RAID0 configurations. As RAID0 uses a similar layout to RAID5, it provided a means to isolate performance issues that were specific to RAID5, rather than general to the hardware.

To give an idea of the sort of data that resulted, figure 1 shows a graph of RAID0 sequential read throughput for 2.4.0-test10 using varying chunk sizes, and figure 2 shows RAID0 and RAID5 sequential write throughput for 2.4.0-test10 and 2.2.18 plus RAID patches for two different chunk sizes.

[1] See `http://www.research.att.com/g̃jm/linux/ide-i75RAID.html`

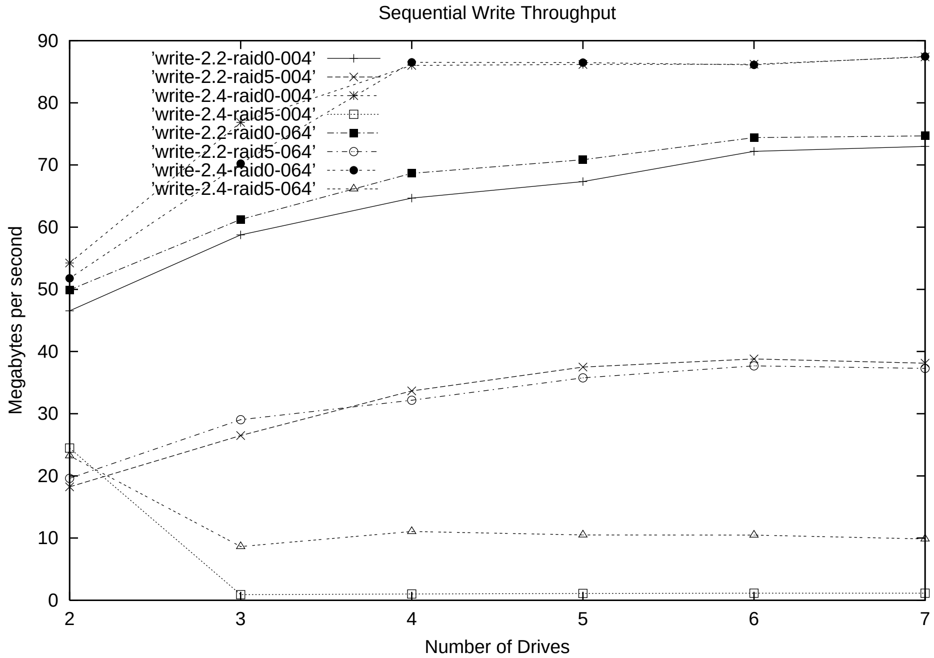

Figure 2: RAID0 and RAID5 Write throughput in various kernels with a 4K and a 64K chunk size

5.1 Over contentious locks

As can be seen from figure 2, while RAID0 write performance has gone up for 2.4, RAID5 write performance has dropped down badly. Further, while the performance for all other configurations tends to improve with more drives, the performance for RAID5 in 2.4 is either stable or drops off slightly for more drives.

A large part of the reason for this is the over cautious stripe locking that was introduced in the 2.3 port of the RAID5 code. The effect of this locking on the RAID5 code was that as soon as one I/O request was attached to a stripe, that stripe was locked until the request had been completely dealt with. This substantially reduced opportunities for parallelism between the multiple drives.

For write requests, the RAID code would still snoop on the buffer cache to find more buffers to write in a single stripe, but for reasons mentioned above, it would almost never find them. This essentially meant that RAID5 would write a single chunk, and then wait for that chunk to complete before writing the next chunk. Read performance is affected in a similar way for similar reasons.

The first performance improvement I needed to do was to free up the locking on the stripe cache.

Though it is hard to be certain, I believe that the reason that it had been difficult to get locking right on the stripe cache was related to the explicit state-machine nature of the code.

When writing to a RAID5 array, a number of steps are needed to complete the write:

begin-write This is the initial state when buffers with pending write requests are attached to the stripe.

read-old This is an optional, though common, state where old data or parity blocks are read from the devices to enable parity calculations.

write In this state, data and parity is actually being written to the devices.

complete This is the final state of writing when the write requests are returned to the callers.

These states for writing, and similar states for reading, were encoded using two per-stripe variables, `cmd` which could be `STRIPE_NONE`, `STRIPE_WRITE`, or `STRIPE_READ` and `phase` which could be one of `PHASE_BEGIN`, `PHASE_READ_OLD`, `PHASE_WRITE`, `PHASE_READ`, or `PHASE_COMPLETE`.

Having explicit state like this suggested that well defined state transitions would be appropriate. This means that new requests could not simply be added at any time, as that would require an irregular state change. I believe that this fact made the model for locking the stripes too complicated.

The approach that I took was to get rid of the explicit recording of state altogether. There is still a number of states that needed to be handled, and a number of state transitions, but these are now less explicit, and more natural.

The state of a stripe is implied by the state of all the cache buffers, whether they are up-to-date, or dirty, or locked. Each time a stripe is considered for action, the next action is determined by the composite state of the various buffers. This means that new requests can be accepted at any time, except during very small window that is protected by a spinlock.

The result of these changes is that read throughput is much improved, and that write throughput is somewhat improved, as can be seen in figure 3.

5.2 No more snooping

Both the read and write throughput increased markedly with the fixing of the locking of the stripe cache. This is not surprising as it is now possible for multiple requests on a single stripe to be handled concurrently. However the throughput still hasn't reached 2.2 levels.

The shortfall in read throughput for larger numbers of drives (> 3) will be dealt with in the next section. The shortfall in write throughput — the topic of this section — is due to the loss of buffer cache snooping.

As mentioned earlier, buffer cache snooping in 2.2 allowed RAID5 to often collect a whole stripe worth of write requests together and to process them as a whole. This greatly improves performance as no pre-reading phase is needed. In 2.4, this is not possible, and a different strategy is needed.

The need to collect several requests together to make a full stripe before commencing I/O is similar to the need to collect multiple contiguous requests together when accessing a normal rotating disc drive, to allow full tracks (or at least, larger extents) to be read in a single request. This is currently achieved with an approach known as plugging.

For normal drives, a new request on an idle drive causes the drive queue to be "plugged". This means that the request is queued normally, but that all

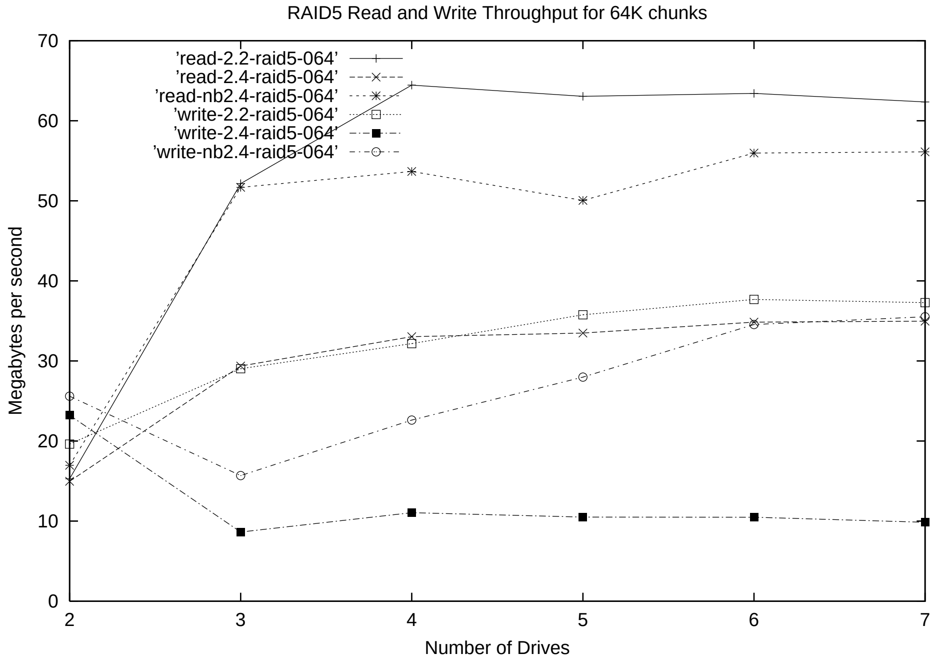

Figure 3: RAID5 Read and Write Throughput for 2.2, 2.4, and patched 2.4

processing of the queue is temporarily disabled. This allows multiple requests to be collected together and merged so that fewer, larger requests get created.

At some point, the queue needs to be unplugged, and this is currently done in Linux using a simple "un-queue-all-devices" call whenever a kernel thread needs to wait for a specific buffer to be completed, or when it is noticed that too much memory is being used by buffers with pending I/O.

If a device is busy, then it is expected that newly added requests will not be processed for a little while. This means that there will be a natural delay in processing requests which will allow neighbouring requests to be merged.

This concept can translate to RAID5, but not perfectly. As an initial test of the idea, I implemented a simple plugging scheme where-by stripes with write requests added to them were added to a separate queue which would only be processes after an unplug call. As the figure 4 shows, this made a substantial difference for write throughput to arrays with a 4K chunk size. It made some difference for other configurations, but not as much, as figure 5 shows. The key feature about the 4k-chunk configuration is that it consecutive requests are likely to get attached to the same stripe (given a 4K filesystem and hence a 4K stripe size). Other configurations need to collect requests on more than two stripes at once, and this didn't seem to happen very often. I believe this is because the queue was being unplugged too often.

One cause of frequent unplugging is the RAID5 code itself. The RAID5 code maintains a fixed size stripe cache. Whenever a request arrives for a stripe that is not in the cache, it much wait for a stripe to become available. Before waiting, it must unplug any underlying devices. Because of the paucity of the interface of unplugging devices, it must unplug all devices, including itself! This

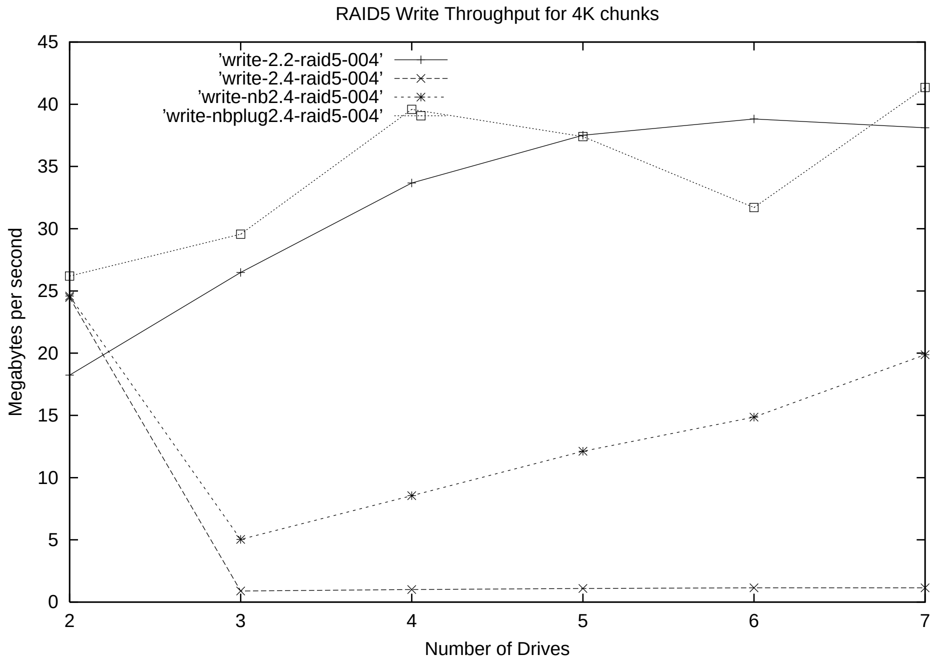

Figure 4: RAID5 Write Throughput showing the effect of plugging with a 4K chunk size

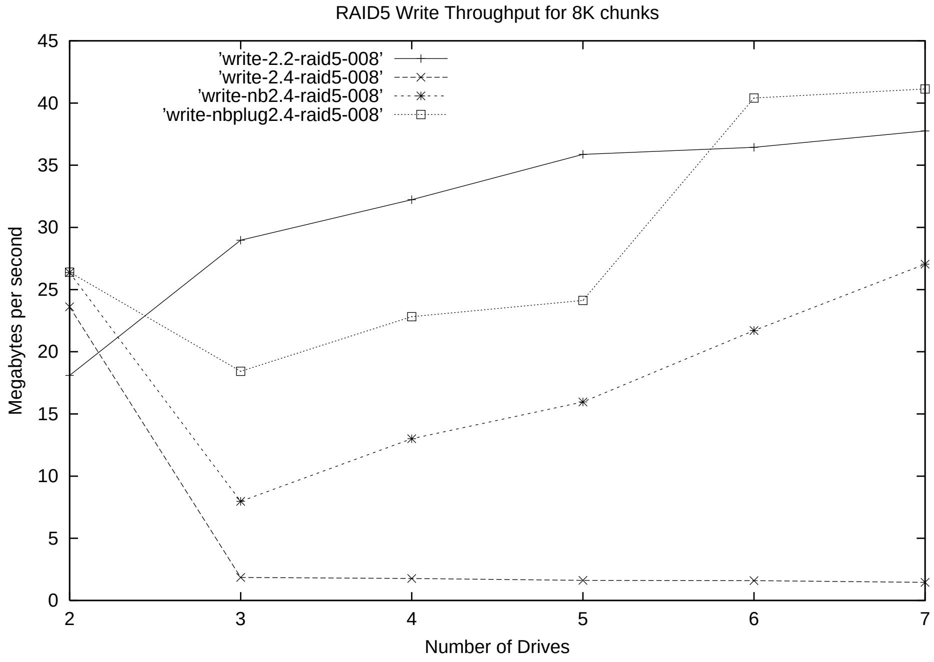

Figure 5: RAID5 Write Throughput showing the effect of plugging with a 8K chunk size

means that for every request that goes to a new stripe, any previous stripes will be unplugged.

There are strategies available to reduce the problem of frequent unplugging. One is to free-up stripes in large groups, instead of just one at a time. This means that a series of requests can arrive for several new stripes, and they can all be served without forcing a unplug.

Another strategy which holds more promise I believe is to simulate the delay of a busy drive.

Currently as soon as a stripe has a request attached to it, it will cause a matching request to be submitted to lower level devices (unless it is a plugged write request). This is done without any regard to how busy the underlying device is, as RAID5 has no way to know. If, however, we limit the number of stripes which can have outstanding requests on lower level devices to, say, half of the cache, then we will get a natural delay in processing that is independent of plugging (which as we have seen is not alway effective).

This strategy has not yet been implemented.

5.3 Loss of zero copy reads

The shortfall in read throughput observed in figure 3 only affects arrays with more than 3 drives, and seems to be general lowering of the ceiling imposed by some bottle neck in the whole system.

It will be helpful at this point to consider the relationship between read throughput for RAID0 and RAID5. The read throughput of a fully working RAID5 array should be comparable to that of a RAID0 array, except for the effect of the parity blocks. Though the parity blocks do not have to be read, they do have to be skipped over by the drive head, and so they do affect the throughput. In general for an N drive array, the RAID5 read throughput should be about the $\frac{N-1}{N}$ of the RAID0 read throughput.

Figure 6 shows that this is true for small values of N, it quickly ceases to be true as N grows.

The main difference between the handling of reading in RAID5 and in RAID0 is the stripe cache in RAID5. The RAID5 read must both wait for a free stripe, if there isn't one, and must copy the data out of the cache and into the target buffer on completion. Both of these could slow things down.

I implemented a simple cache-bypass scheme for read requests on stripes that were not engaged in writing or synching and got throughput must closer to the theoretical goal as shown in figure 6. This shows that memory-to-memory copying was slowing things down for high throughputs.

The original RAID code already treated read requests this way, and this functionality was lost when I freed up the locking on the stripe cache, so it is not surprising that it needed to be put back.

However I am quite happy that I left it out and then put it back in later as

- it meant I could have an objective measure of how important it was and
- I am fairly sure that the code was much neater than it would have been had I tried to include it from the beginning. It is a special case fast-path and is best treated as an optional extra that is only used when convenient.

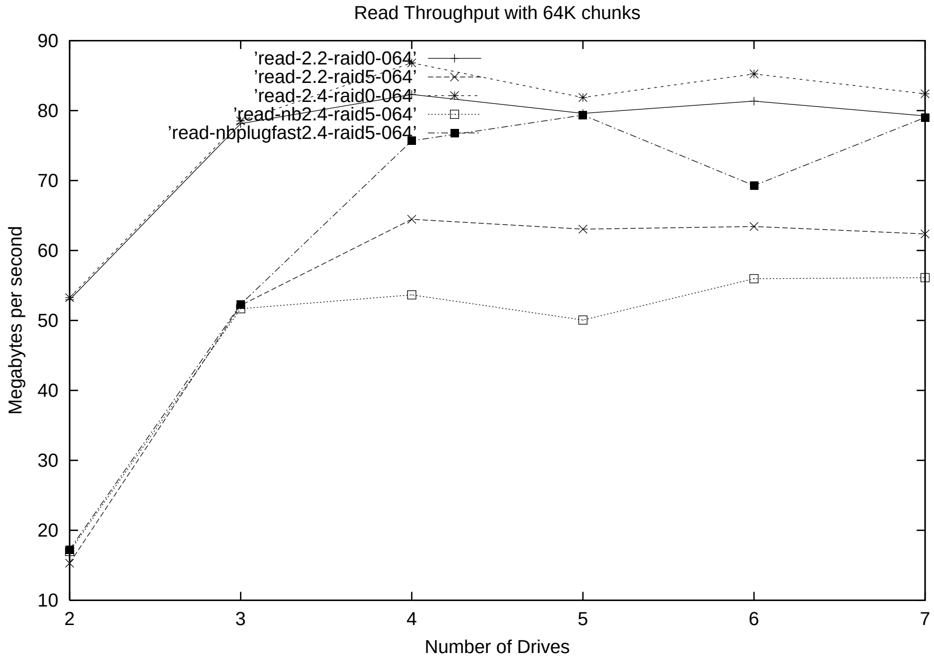

Figure 6: Read throughput comparing RAID0 and RAID5 and particularly showing the effect of bypassing the cache for reading

6 Future directions

With the changes described above, software RAID, and RAID5 in particular, is quite usable in 2.4 kernels and as usable or more so than the software RAID in patched 2.2 kernels. If the scheme for clustering writes by limiting the number of concurrent lower level requests works, this will be especially true.

However there is more work that can be done to make soft RAID work even better. Some directions are outlined below.

6.1 Partitions

Most storage devices can be partitioned into separate sub-devices to allow convenient storage management. However RAID devices cannot currently be partitioned. It is true that subdivision can be done using LVM, the logical volume manager. However that approach is more heavy weight that traditional partitioning, and may not always provide what is required.

A particular use for partition is for providing a mirrored pair of devices as the boot device. A typical boot device contains a partition for the root filesystem, a partition for swap, and another partition for `/usr` or similar. It would be nice to be able to mirror the whole boot device onto another device, and then partition the whole in the same way that a single device is partitioned. This would particularly make adding a mirror to an already configured machine much easier.

I have initial patches to do this, but more work is needed.

6.2 Reorganise recovery

Currently the recovery processes is a bit awkward.

- Recovery after an unclean shutdown is handled quite differently to recovery after a failed drive.
- Only one RAID array in a system can recover from a failed drive at a time.
- It is not possible to have a spare drive that can be attached as needed to either of two RAID arrays.

None of these are particularly hard issues to solve, but they point to an untidiness in the implementation that should be dealt with.

6.3 RAID5 with variable buffer sizes

Currently the stripe size of the RAID5 stripe cache is set to the block size of the filesystem that is mounted on it. If the block size changes (as it does, for example, when first mounting the filesystem) the stripe cache is flushed and rebuilt.

This assumes that a filesystem has a fixed block size. However many newer filesystem would benefit from not having a fixed block size and it is likely that Linux will drop this restriction soon. When it does, RAID5 will be left out in the cold.

Something needs to be done to allow requests of various sizes to be handled by RAID5. This means having a fixed stripe size for the stripe cache, and either mapping all requests to this size, pre-reading for small writes, and over-reading for small reads, or allowing each buffer in a stripe to be only partially valid.

This is as yet completely unimplemented.

6.4 Random other stuff

1. Currently, some status information is made available in `/proc/md`. It would be nice to create a directory structure instead which made it easier to access and parse information about different devices.
2. Currently, writes to the RAID super block are NOT CHECKED, and a write error there will not be noticed.
3. There is a lot of un-necessary data stored in the super block. It would be possible to choose a new superblock layout that is only 1K (instead of 4K) and allows for hundreds of devices per array instead of 29.