The Vinum Volume Manager

Greg Lehey
Nan Yang Computer Services Ltd.
PO Box 460
Echunga SA 5153
grog@lemis.com

ABSTRACT

The *Vinum Volume Manager* is a block device driver which implements virtual disk drives. It isolates disk hardware from the block device interface and maps data in ways which result in an increase in flexibility, performance and reliability compared to the traditional slice view of disk storage. Vinum implements the RAID-0, RAID-1 and RAID-5 models, both individually and in combination.

Introduction

Disk hardware is evolving rapidly, and the current UNIX disk abstraction is inadequate for a number of modern applications. In particular, file systems must be stored on a single disk partition, and there is no kernel support for redundant data storage. In addition, the direct relationship between disk volumes and their location on disk make it generally impossible to enlarge a disk volume once it has been created. Performance can often be limited by the maximum data rate which can be achieved with the disk hardware.

The largest modern disks store only about 50 GB, but large installations, in particular web sites, routinely have more than a terabyte of disk storage, and it is not uncommon to see disk storage of several hundred gigabytes even on PCs. Storage-intensive applications such as Internet World-Wide Web and FTP servers have accelerated the demand for high-volume, reliable storage systems which deliver high performance in a heavily concurrent environment.

The problems

Various solutions to these problems have been proposed and implemented:

Disks are too small

The *ufs* file system can theoretically span more than a petabyte (2^{50} or 10^{15} bytes) of storage, but no current disk drive comes close to this size. Although this problem is not as acute as it was ten years ago, there is a simple solution: the disk driver can create an abstract device which stores its data on a number of disks. A number of such implementations exist, though none appear to have become mainstream.

Access bottlenecks

Modern systems frequently need to access data in a highly concurrent manner. For example, *wcarchive.cdrom.com* maintains up to 3,600 concurrent *FTP* sessions and has a 100 Mbit/s connection to the outside world, corresponding to about 12 MB/s.

Current disk drives can transfer data sequentially at up to 30 MB/s, but this value is of little importance in an environment where many independent processes access a drive, where they may achieve only a fraction of these values. In such cases it's more interesting to view the problem from the viewpoint of the disk subsystem: the important parameter is the load that a transfer places on the subsystem, in other words the time for which a transfer occupies the drives involved in the transfer.

In any disk transfer, the drive must first position the heads, wait for the first sector to pass under the read head, and then perform the transfer. These actions can be considered to be atomic: it doesn't make any sense to interrupt them.

Consider a typical transfer of about 10 kB: the current generation of high-performance disks can position the heads in an average of 6 ms. The fastest drives spin at 10,000 rpm, so the average rotational latency (half a revolution) is 3 ms. At 30 MB/s, the transfer itself takes about 350 μs, almost nothing compared to the positioning time. In such a case, the effective transfer rate drops to a little over 1 MB/s and is clearly highly dependent on the transfer size.

The traditional and obvious solution to this bottleneck is "more spindles": rather than using one large disk, it uses several smaller disks with the same aggregate storage space. Each disk is capable of positioning and transferring independently, so the effective throughput increases by a factor close to the number of disks used.

The exact throughput improvement is, of course, smaller than the number of disks involved: although each drive is capable of transferring in parallel, there is no way to ensure that the requests are evenly distributed across the drives. Inevitably the load on one drive will be higher than on another.

The evenness of the load on the disks is strongly dependent on the way the data is shared across the drives. In the following discussion, it's convenient to think of the disk storage as a large number of data sectors which are addressable by number, rather like the pages in a book. The most obvious method is to divide the virtual disk into groups of consecutive sectors the size of the individual physical disks and store them in this manner, rather like taking a large book and tearing it into smaller sections. This method is called *concatenation* and has the advantage that the disks do not need to have any specific size relationships. It works well when the access to the virtual disk is spread evenly about its address space. When access is concentrated on a smaller area, the improvement is less marked. Figure 1 illustrates the sequence in which storage units are allocated in a concatenated organization.

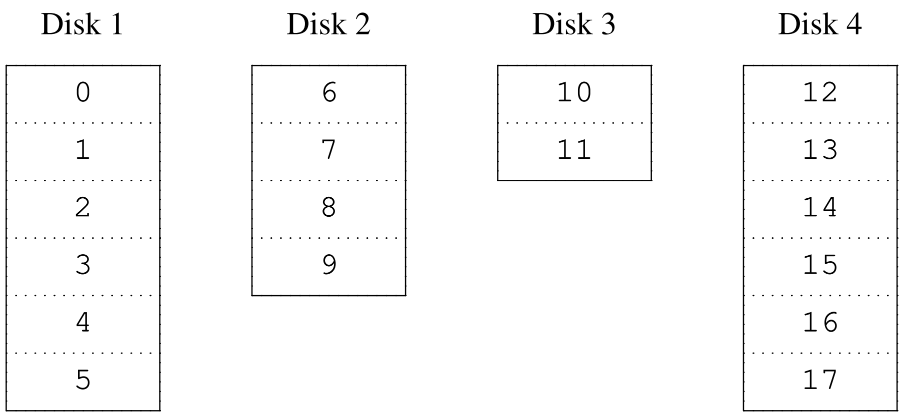

Figure 1: Concatenated organization

An alternative mapping is to divide the address space into smaller, even-sized components and store them sequentially on different devices. For example, the first 256 sectors may be stored on the first disk, the next 256 sectors on the next disk and so on. After filling the last disk, the process repeats until the disks are full. This mapping is called *striping* or RAID-0, though the latter term is somewhat misleading: it provides no redundancy. Striping requires somewhat more effort to locate the data, and it can cause additional I/O load where a transfer is spread over multiple disks, but it can also provide a more constant load across the disks. Figure 2 illustrates the sequence in which storage units are allocated in a striped organization.

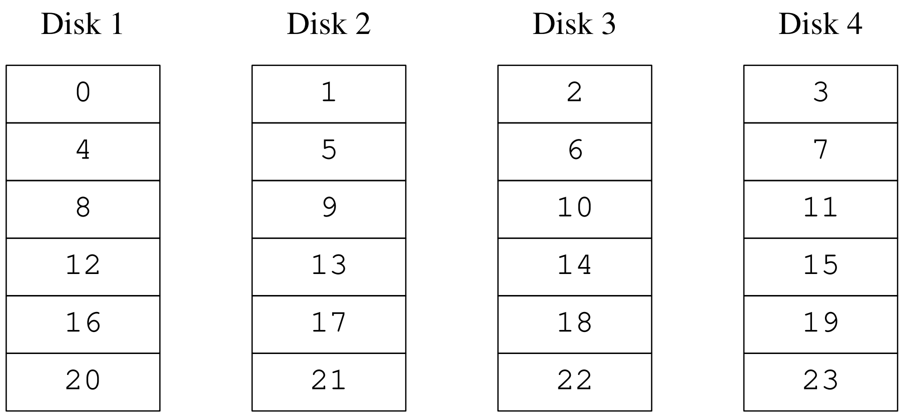

Figure 2: Striped organization

Data integrity

The final problem with current disks is that they are unreliable. Although disk drive reliability has increased tremendously over the last few years, they are still the most likely core component of a server to fail. When they do, the results can be catastrophic: replacing a failed disk drive and restoring data to it can take days.

The traditional way to approach this problem has been *mirroring*, keeping two copies of the data on different physical hardware. Since the advent of the RAID levels, this technique has also been called *RAID level 1* or *RAID-1*. Any write to the volume writes to both locations; a read can be satisfied from either, so if one drive fails, the data is still available on the other drive.

Mirroring has two problems:

- The price. It requires twice as much disk storage as a non-redundant solution.
- The performance impact. Writes must be performed to both drives, so they take up twice the bandwidth of a non-mirrored volume. Reads do not suffer from a performance penalty: it even looks as if they are faster. This issue will be discussed on page 20.

An alternative solution is *parity*, implemented in the RAID levels 2, 3, 4 and 5. Of these, RAID-5 is the most interesting. As implemented in Vinum, it is a variant on a striped organization which dedicates one block of each stripe to parity of the other blocks: As implemented by Vinum, a *RAID-5* plex is similar to a striped plex, except that it implements RAID-5 by including a parity block in each stripe. As required by RAID-5, the location of this parity block changes from one stripe to the next. The numbers in the data blocks indicate the relative block numbers.

Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	Parity
3	4	Parity	5
6	Parity	7	8
Parity	9	10	11
12	13	14	Parity
15	16	Parity	17

Figure 3: RAID-5 organization

Compared to mirroring, RAID-5 has the advantage of requiring significantly less storage space. Read access is similar to that of striped organizations, but write access is significantly slower, approximately 25% of the read performance. If one drive fails, the array can continue to operate in degraded mode: a read from one of the remaining accessible drives continues normally, but a read from the failed drive is recalculated from the corresponding block from all the remaining drives.

Current implementations

The three requirements discussed above have solutions that are only partially compatible. In particular, redundant data storage and performance improvements require different solutions and affect each other negatively.

The current trend is to realize such systems in *disk array* hardware, which looks to the host system like a very large disk. Disk arrays have a number of advantages:

- They are portable. Since they have a standard interface, usually SCSI, but increasingly also IDE, they can be installed on almost any system without kernel modifications.
- They have the potential to offer impressive performance: they offload the calculations (in particular, the parity calculations for RAID-5) to the array, and in the case of replicated data, the aggregate transfer rate to the array is less than it would be to local disks. RAID-0 (striping) and RAID-5 organizations also spread the load more evenly over the physical disks, thus improving performance. Nevertheless, an array is typically connected via a single SCSI connection, which can be a bottleneck, and some implementations show surprisingly poor performance. Installing a disk array does not guarantee better performance.
- They are reliable. A good disk array offers a large number of features designed to enhance reliability, including enhanced cooling, hot-plugging (the ability to replace a drive while the array is running) and automatic failure recovery.

On the other hand, disk arrays are relatively expensive and not particularly flexible. An alternative is a software-based *volume manager* which performs similar functions in software. A number of these systems exist, notably the VERITAS® volume manager [Veritas], Solaris *DiskSuite* [Solstice], IBM's *Logical Volume Facility* [IBM] and SCO's *Virtual Disk Manager* [SCO]. An implementation of RAID software is also available for Linux [Linux].

Vinum

Vinum is an open source [OpenSource] volume manager implemented under FreeBSD [FreeBSD]. It was inspired by the VERITAS® volume manager and implements many of the concepts of VERITAS®. Its key features are:

- Vinum implements RAID-0 (striping), RAID-1 (mirroring) and RAID-5 (rotated block-interleaved parity). In RAID-5, a group of disks are protected against the failure of any one disk by an additional disk with block checksums of the other disks.[1]
- Drive layouts can be combined to increase robustness, including striped mirrors (so-called "RAID-10").

1. The RAID-5 functionality is currently available under license from Cybernet, Inc. [Cybernet]. It will be released as open source at a later date.

- Vinum implements only those features which appear useful. Some commercial volume managers appear to have been implemented with the goal of maximizing the size of the spec sheet. Vinum does not implement ‘‘ballast’’ features such as RAID-4. It would have been trivial to do so, but the only effect would have been to further confuse an already confusing topic.
- Volume managers initially emphasized reliability and performance rather than ease of use. The results are frequently down time due to misconfiguration, with consequent reluctance on the part of operational personnel to attempt to use the more unusual features of the product. Vinum attempts to provide an easier-to-use non-GUI interface.

How Vinum addresses the Three Problems

As mentioned above, Vinum addresses three main deficiencies of traditional disk hardware. This section examines them in more detail.

Vinum objects

In order to address these problems, vinum implements a four-level hierarchy of objects:

- The most visible object is the virtual disk, called a *volume*. Volumes have essentially the same properties as a UNIX disk drive, though there are some minor differences. They have no size limitations.
- Volumes are composed of *plexes*, each of which represent the total address space of a volume. This level in the hierarchy thus provides redundancy.
- Since Vinum exists within the UNIX disk storage framework, it would be possible to use UNIX partitions as the building block for multi-disk plexes, but in fact this turns out to be too inflexible: UNIX disks can have only a limited number of partitions. Instead, Vinum subdivides a single UNIX partition (the *drive*) into contiguous areas called *subdisks*, which it uses as building blocks for plexes.
- Subdisks reside on Vinum *drives*, currently UNIX partitions. Vinum drives can contain any number of subdisks. With the exception of a small area at the beginning of the drive, which is used for storing configuration and state information, the entire drive is available for data storage.

The following sections describe the way these objects provide the functionality required of Vinum.

Volume size considerations

Plexes can include multiple subdisks spread over all drives in the Vinum configuration. As a result, the size of an individual drive does not limit the size of a plex, and thus of a volume.

Redundant data storage

Vinum provides both mirroring and RAID-5. It implements mirroring by attaching multiple plexes to a volume. Each plex is a representation of the data in a volume. A volume may contain between one and eight plexes.

Although a plex represents the complete data of a volume, it is possible for parts of the representation to be physically missing, either by design (by not defining a subdisk for parts of the plex) or by accident (as a result of the failure of a drive). As long as at least one plex can provide the data for the complete address range of the volume, the volume is fully functional.

From an implementation standpoint, it is not practical to represent a RAID-5 organization as a collection of plexes. This issue is discussed below.

Performance issues

Vinum implements both concatenation and striping at the plex level:

- A *concatenated plex* uses the address space of each subdisk in turn.
- A *striped plex* stripes the data across each subdisk. The subdisks must all have the same size, and there must be at least two subdisks in order to distinguish it from a concatenated plex.
- Like a striped plex, a *RAID-5 plex* stripes the data across each subdisk. The subdisks must all have the same size, and there must be at least three subdisks, since otherwise mirroring would be more efficient.

RAID-5

Conceptually, RAID-5 is used for redundancy, but in fact the implementation is a kind of striping. This poses problems for the implementation of Vinum: should it be a kind of plex or a kind of volume? It would have been possible to implement it either way, but it proved to be simpler to implement RAID-5 as a plex type. This means that there are two different ways of ensuring data redundancy: either have more than one plex in a volume, or have a single RAID-5 plex. These methods can be combined.

Which plex organization?

Vinum implements only that subset of RAID organizations which make sense in the framework of the implementation. It would have been possible to implement all RAID levels, but there was no reason to do so. Each of the chosen organizations has unique advantages:

- Concatenated plexes are the most flexible: they can contain any number of subdisks, and the subdisks may be of different length. The plex may be extended by adding additional subdisks. They require less CPU time than striped or RAID-5 plexes, though the difference in CPU overhead from striped plexes is not measurable. On the other hand, they are most susceptible to hot spots, where one disk is very active and others are idle.

- The greatest advantage of striped (RAID-0) plexes is that they reduce hot spots: by choosing an optimum sized stripe (empirically determined to be in the order of 256 kB), the load on the component drives can be made more even. The disadvantages of this approach are (fractionally) more complex code and restrictions on subdisks: they must be all the same size, and extending a plex by adding new subdisks is so complicated that Vinum currently does not implement it. Vinum imposes an additional, trivial restriction: a striped plex must have at least two subdisks, since otherwise it is indistinguishable from a concatenated plex.

- RAID-5 plexes are effectively an extension of striped plexes. Compared to striped plexes, they offer the advantage of fault tolerance, but the disadvantages of higher storage cost and significantly higher CPU overhead, particularly for writes. The code is an order of magnitude more complex than for concatenated and striped plexes. Like striped plexes, RAID-5 plexes must have equal-sized subdisks and cannot currently be extended. Vinum enforces a minimum of three subdisks for a RAID-5 plex, since any smaller number would not make any sense.

Table 4 summarizes the advantages and disadvantages of each plex organization.

Plex type	Minimum subdisks	can add subdisks	must be equal size	application
concatenated	1	yes	no	Large data storage with maximum placement flexibility and moderate performance.
striped	2	no	yes	High performance in combination with highly concurrent access.
RAID-5	3	no	yes	Highly reliable storage, primarily read access.

Figure 4: Vinum plex organizations

These are not the only possible organizations. In addition, the following could have been implemented:

- RAID-4, which differs from RAID-5 only by the fact that all parity data is stored on a specific disk. This simplifies the algorithms somewhat at the expense of drive utilization: the activity on the parity disk is a direct function of the read to write ratio. Since Vinum implements RAID-5, RAID-4's only advantage is nullified.

- RAID-3, effectively an implementation of RAID-4 with a stripe size of one byte. Each transfer requires reading each disk (with the exception of the parity disk for reads). Without spindle synchronization (where the corresponding sectors pass the heads of each drive at the same time), RAID-3 would be very inefficient. In a multiple-access system, it also causes high latency.

An argument for RAID-3 does exist where a single process requires very high data rates. With spindle synchronization, this would be a potentially useful addition to Vinum.

- RAID-2, which uses two subdisks to store a Hamming code, and which otherwise resembles RAID-3. Compared to RAID-3, it offers a lower data density, higher CPU usage and no compensating advantages.

In addition, RAID-5 can be interpreted in two different ways: the data can be striped, as in the Vinum implementation, or it can be written serially, exhausting the address space of one subdisk before starting on the other, effectively a modified concatenated organization. There is no recognizable advantage to this approach, since it does not provide any of the other advantages of concatenation.

Some examples

Vinum maintains a *configuration database* which describes the objects known to an individual system. Initially, the user creates the configuration database from one or more configuration files with the aid of the *vinum(8)* utility program. Vinum stores a copy of its configuration database on each disk slice (which Vinum calls a *device*) under its control. This database is updated on each state change, so that a restart accurately restores the state of each Vinum object.

The configuration file

The configuration file describes individual Vinum objects. The definition of a simple volume might be:

```
drive a device /dev/da3h
volume myvol
  plex org concat
    sd length 512m drive a
```

This file describes a four Vinum objects:

- The `drive` line describes a disk partition (*drive*) and its location relative to the underlying hardware. It is given the symbolic name *a*. This separation of the symbolic names from the device names allows disks to be moved from one location to another without confusion.
- The `volume` line describes a volume. The only required attribute is the name, in this case `myvol`.
- The `plex` line defines a plex. The only required parameter is the organization, in this case `concat`. No name is necessary: the system automatically generates a name from the volume name by adding the suffix `.p`***x***, where ***x*** is the number of the plex in the volume. Thus this plex will be called *myvol.p0*.

- The `sd` line describes a subdisk. The minimum specifications are the name of a drive on which to store it, and the length of the subdisk. As with plexes, no name is necessary: the system automatically assigns names derived from the plex name by adding the suffix **.** s***x***, where ***x*** is the number of the subdisk in the plex. Thus Vinum gives this subdisk the name *myvol.p0.s0*

After processing this file, *vinum(8)* produces the following output:

```
vinum -> create config1
Configuration summary

Drives:         1 (4 configured)
Volumes:        1 (4 configured)
Plexes:         1 (8 configured)
Subdisks:       1 (16 configured)

D a                     State: up       Device /dev/da3h        Avail: 2061/2573 MB (80%)

V myvol                 State: up       Plexes:       1 Size:        512 MB

P myvol.p0            C State: up       Subdisks:     1 Size:        512 MB

S myvol.p0.s0           State: up       PO:        0  B Size:        512 MB
```

This output shows the brief listing format of *vinum(8)*. It is represented graphically in Figure 5.

This figure, and the ones which follow, represent a volume, which contains the plexes, which in turn contain the subdisks. In this trivial example, the volume contains one plex, and the plex contains one subdisk.

This particular volume has no specific advantage over a conventional disk partition. It contains a single plex, so it is not redundant. The plex contains a single subdisk, so there is no difference in storage allocation from a conventional disk partition. The following sections illustrate various more interesting configuration methods.

Increased resilience: mirroring

The resilience of a volume can be increased either by mirroring or by using RAID-5 plexes. When laying out a mirrored volume, it is important to ensure that the subdisks of each plex are on different drives, so that a drive failure will not take down both plexes. The following configuration mirrors a volume:

```
drive b device /dev/da4h
volume mirror
  plex org concat
    sd length 512m drive a
  plex org concat
    sd length 512m drive b
```

In this example, it was not necessary to specify a definition of drive *a* again, since Vinum keeps track of all objects in its configuration database. After processing this definition, the configuration looks like:

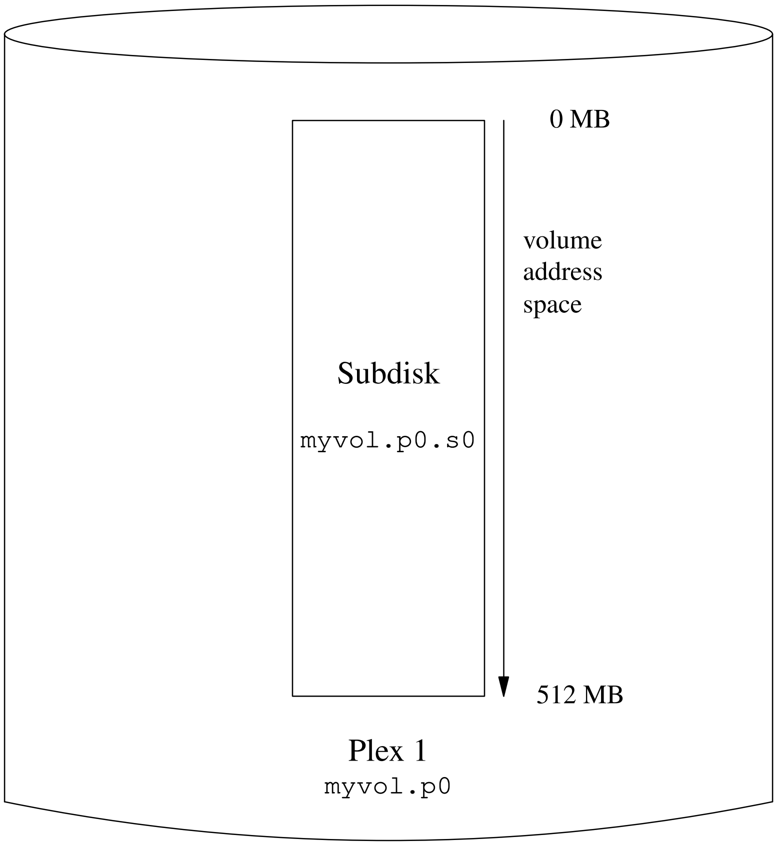

Figure 5: A simple Vinum volume

```
Drives:         2 (4 configured)
Volumes:        2 (4 configured)
Plexes:         3 (8 configured)
Subdisks:       3 (16 configured)

D a                     State: up       Device /dev/da3h        Avail: 1549/2573 MB (60%)
D b                     State: up       Device /dev/da4h        Avail: 2061/2573 MB (80%)

V myvol                 State: up       Plexes:       1 Size:        512 MB
V mirror                State: up       Plexes:       2 Size:        512 MB

P myvol.p0            C State: up       Subdisks:     1 Size:        512 MB
P mirror.p0           C State: up       Subdisks:     1 Size:        512 MB
P mirror.p1           C State: initializing     Subdisks:     1 Size:        512 MB

S myvol.p0.s0           State: up       PO:        0  B Size:        512 MB
S mirror.p0.s0          State: up       PO:        0  B Size:        512 MB
S mirror.p1.s0          State: empty    PO:        0  B Size:        512 MB
```

Figure 6 shows the structure graphically.

In this example, each plex contains the full 512 MB of address space. As in the previous example, each plex contains only a single subdisk.

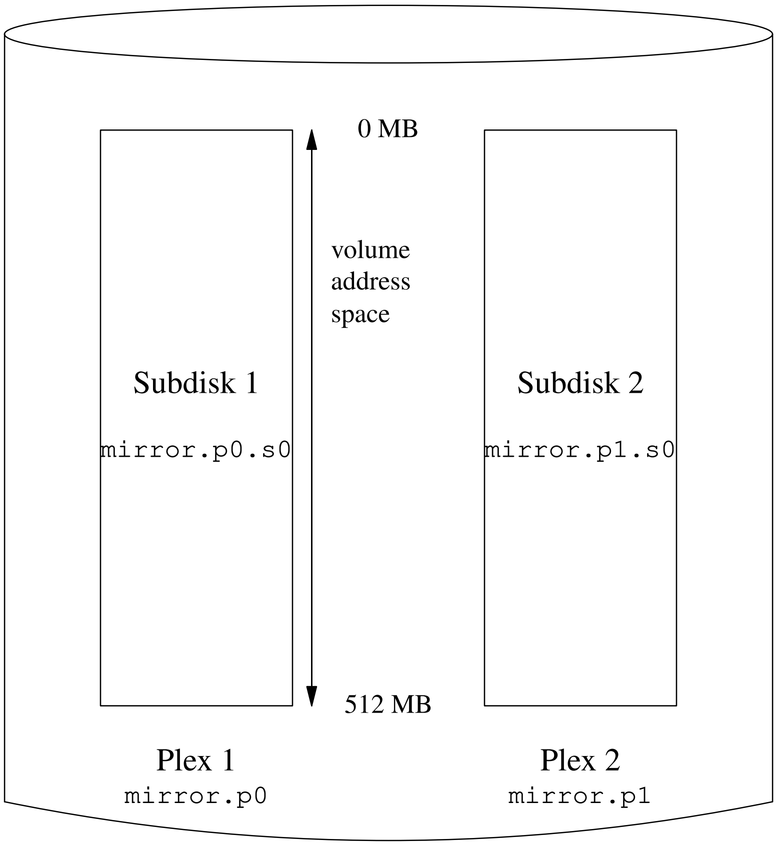

Figure 6: A mirrored Vinum volume

Optimizing performance

The mirrored volume in the previous example is more resistant to failure than an unmirrored volume, but its performance is less: each write to the volume requires a write to both drives, using up a greater proportion of the total disk bandwidth. Performance considerations demand a different approach: instead of mirroring, the data is striped across as many disk drives as possible. The following configuration shows a volume with a plex striped across four disk drives:

```
drive c device /dev/da5h
drive d device /dev/da6h
volume stripe
  plex org striped 512k
    sd length 128m drive a
    sd length 128m drive b
    sd length 128m drive c
    sd length 128m drive d
```

As before, it is not necessary to define the drives which are already known to Vinum. After processing this definition, the configuration looks like:

```
Drives:         4 (4 configured)
Volumes:        3 (4 configured)
Plexes:         4 (8 configured)
Subdisks:       7 (16 configured)

D a                     State: up       Device /dev/da3h        Avail: 1421/2573 MB (55%)
D b                     State: up       Device /dev/da4h        Avail: 1933/2573 MB (75%)
D c                     State: up       Device /dev/da5h        Avail: 2445/2573 MB (95%)
D d                     State: up       Device /dev/da6h        Avail: 2445/2573 MB (95%)

V myvol                 State: up       Plexes:       1 Size:        512 MB
V mirror                State: up       Plexes:       2 Size:        512 MB
V striped               State: up       Plexes:       1 Size:        512 MB

P myvol.p0            C State: up       Subdisks:     1 Size:        512 MB
P mirror.p0           C State: up       Subdisks:     1 Size:        512 MB
P mirror.p1           C State: initializing     Subdisks:     1 Size:        512 MB
P striped.p1            State: up       Subdisks:     1 Size:        512 MB

S myvol.p0.s0           State: up       PO:        0  B Size:        512 MB
S mirror.p0.s0          State: up       PO:        0  B Size:        512 MB
S mirror.p1.s0          State: empty    PO:        0  B Size:        512 MB
S striped.p0.s0         State: up       PO:        0  B Size:        128 MB
S striped.p0.s1         State: up       PO:      512 kB Size:        128 MB
S striped.p0.s2         State: up       PO:     1024 kB Size:        128 MB
S striped.p0.s3         State: up       PO:     1536 kB Size:        128 MB
```

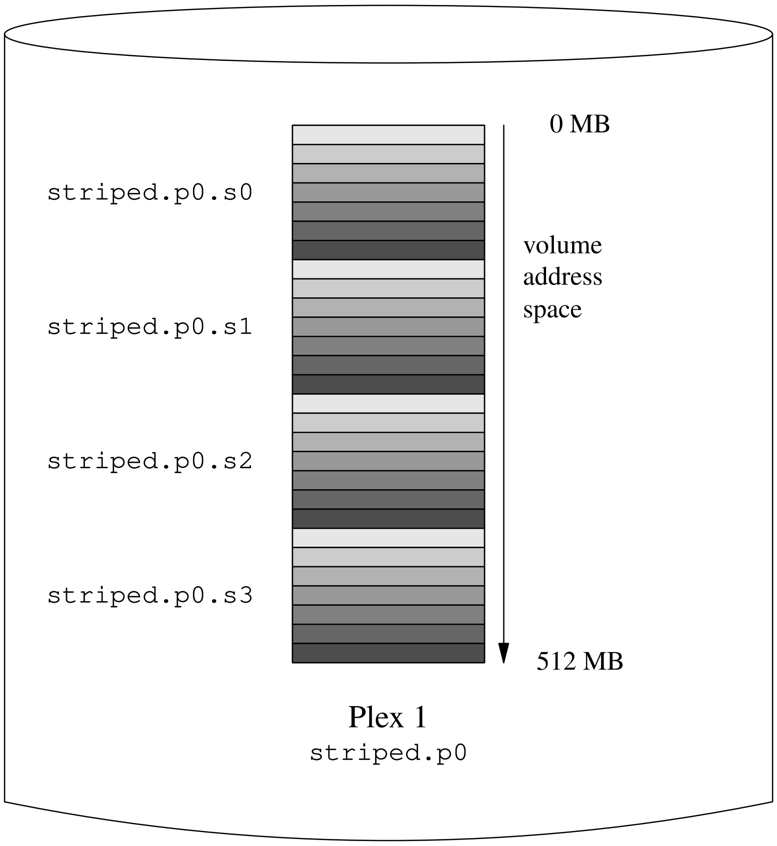

Figure 7: A striped Vinum volume

This volume is represented in Figure 7. The darkness of the stripes indicates the position within the plex address space: the lightest stripes come first, the darkest last.

Increased resilience: RAID-5

The alternative approach to resilience is RAID-5. A RAID-5 configuration might look like:

```
drive e device /dev/da6h
volume raid5
  plex org raid5 512k
    sd length 128m drive a
    sd length 128m drive b
    sd length 128m drive c
    sd length 128m drive d
    sd length 128m drive e
```

Although this plex has five subdisks, its size is the same as the plexes in the other examples, since the equivalent of one subdisk is used to store parity information. After processing the configuration, the system configuration is:

```
Drives:         5 (8 configured)
Volumes:        4 (4 configured)
Plexes:         5 (8 configured)
Subdisks:       12 (16 configured)

D a                     State: up       Device /dev/da3h        Avail: 1293/2573 MB (50%)
D b                     State: up       Device /dev/da4h        Avail: 1805/2573 MB (70%)
D c                     State: up       Device /dev/da5h        Avail: 2317/2573 MB (90%)
D d                     State: up       Device /dev/da6h        Avail: 2317/2573 MB (90%)
D e                     State: up       Device /dev/da6h        Avail: 2445/2573 MB (95%)

V myvol                 State: up       Plexes:       1 Size:        512 MB
V mirror                State: up       Plexes:       2 Size:        512 MB
V striped               State: up       Plexes:       1 Size:        512 MB
V raid5                 State: up       Plexes:       1 Size:        512 MB

P myvol.p0            C State: up       Subdisks:     1 Size:        512 MB
P mirror.p0           C State: up       Subdisks:     1 Size:        512 MB
P mirror.p1           C State: initializing     Subdisks:     1 Size:        512 MB
P striped.p0          S State: up       Subdisks:     1 Size:        512 MB
P raid5.p0            R State: up       Subdisks:     1 Size:        512 MB

S myvol.p0.s0           State: up       PO:        0  B Size:        512 MB
S mirror.p0.s0          State: up       PO:        0  B Size:        512 MB
S mirror.p1.s0          State: empty    PO:        0  B Size:        512 MB
S striped.p0.s0         State: up       PO:        0  B Size:        128 MB
S striped.p0.s1         State: up       PO:      512 kB Size:        128 MB
S striped.p0.s2         State: up       PO:     1024 kB Size:        128 MB
S striped.p0.s3         State: up       PO:     1536 kB Size:        128 MB
S raid5.p0.s0           State: init     PO:        0  B Size:        128 MB
S raid5.p0.s1           State: init     PO:      512 kB Size:        128 MB
S raid5.p0.s2           State: init     PO:     1024 kB Size:        128 MB
S raid5.p0.s3           State: init     PO:     1536 kB Size:        128 MB
S raid5.p0.s4           State: init     PO:     1536 kB Size:        128 MB
```

Figure 8 represents this volume graphically. As with striped plexes, the darkness of the stripes indicates the position within the plex address space: the lightest stripes come first, the darkest last. The completely black stripes are the parity stripes.

On creation, RAID-5 plexes are in the *init* state: before they can be used, the parity data must be created. Vinum currently initializes RAID-5 plexes by writing binary zeros to all subdisks, though a probably future alternative is to rebuild the parity blocks, which allows better recovery of crashed plexes.

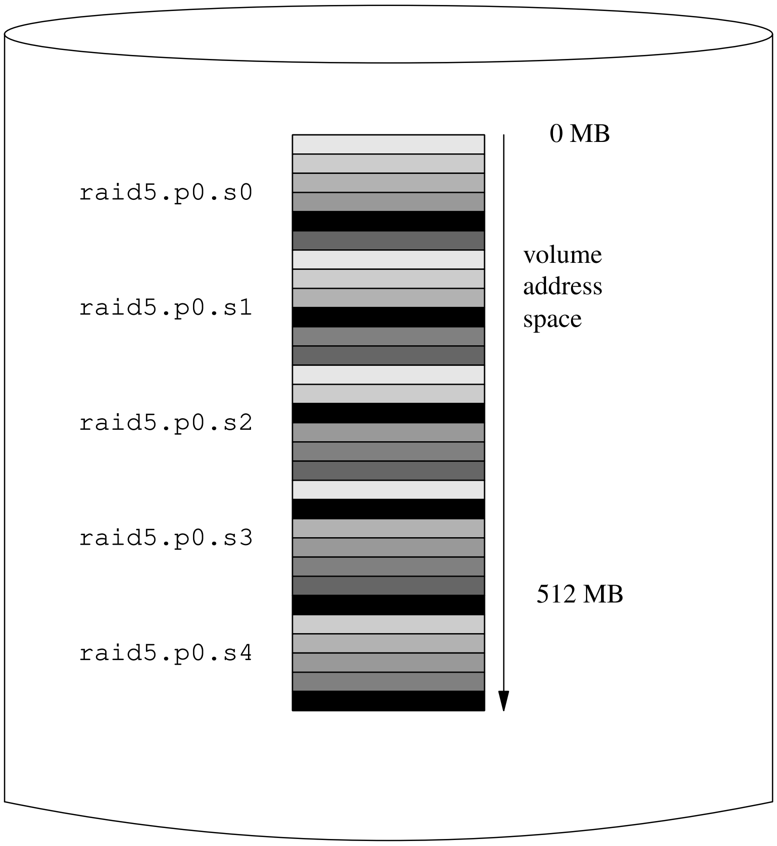

Figure 8: A RAID-5 Vinum volume

Resilience and performance

With sufficient hardware, it is possible to build volumes which show both increased resilience and increased performance compared to standard UNIX partitions. Mirrored disks will always give better performance than RAID-5, so a typical configuration file might be:

```
volume raid10
  plex org striped 512k
    sd length 102480k drive a
    sd length 102480k drive b
    sd length 102480k drive c
    sd length 102480k drive d
    sd length 102480k drive e
  plex org striped 512k
    sd length 102480k drive c
    sd length 102480k drive d
    sd length 102480k drive e
    sd length 102480k drive a
    sd length 102480k drive b
```

The subdisks of the second plex are offset by two drives from those of the first plex: this helps ensure that writes do not go to the same subdisks even if a transfer goes over two drives.

Figure 9 represents the structure of this volume.

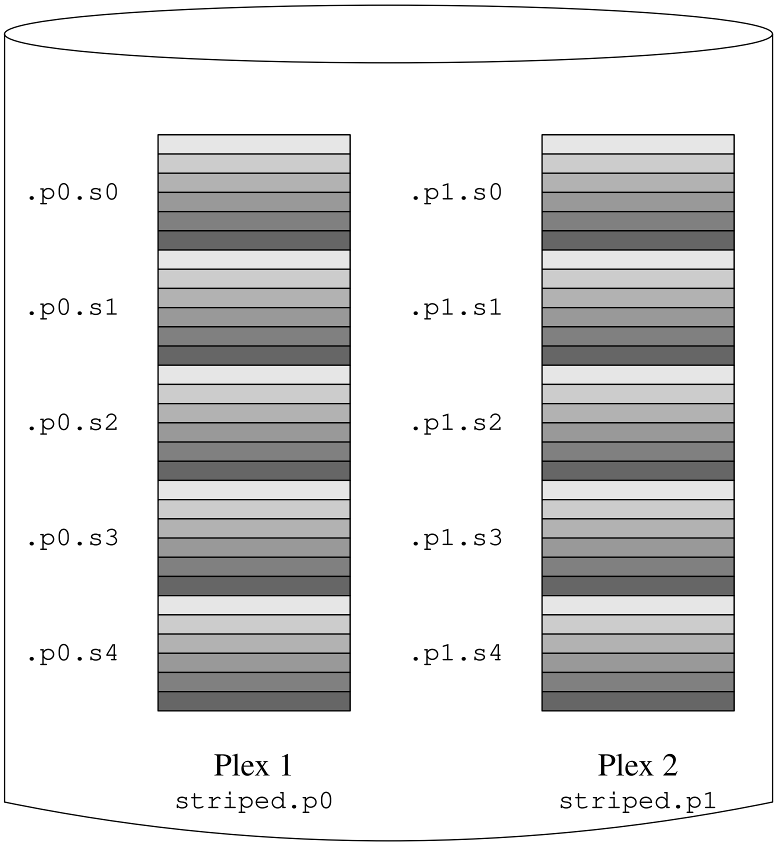

Figure 9: A mirrored, striped Vinum volume

Object naming

As described above, Vinum assigns default names to plexes and subdisks, although they may be overridden. Overriding the default names is not recommended: experience with the VERITAS volume manager, which allows arbitary naming of objects, has shown that this flexibility does not bring a significant advantage, and it can cause confusion.

Names may contain any non-blank character, but it is recommended to restrict them to letters, digits and the underscore characters. The names of volumes, plexes and subdisks may be up to 64 characters long, and the names of drives may up to 32 characters long.

Vinum objects are assigned device nodes in the hierarchy */dev/vinum*. The configuration shown above would cause Vinum to create the following device nodes:

- The control devices */dev/vinum/control* and */dev/vinum/controld*, which are used by *vinum(8)* and the Vinum dæmon respectively.

- Block and character device entries for each volume. These are the main devices used by Vinum. The block device names are the name of the volume, while the character device names follow the BSD tradition of prepending the letter `r` to the name. Thus the configuration above would include the block devices */dev/vinum/myvol*, */dev/vinum/mirror*, */dev/vinum/striped*, */dev/vinum/raid5* and */dev/vinum/raid10*, and the character devices */dev/vinum/rmyvol*, */dev/vinum/rmirror*, */dev/vinum/rstriped*, */dev/vinum/rraid5* and */dev/vinum/rraid10*. There is obviously a problem here: it is possible to have two volumes called *r* and *rr*, but there will be a conflict creating the device node */dev/vinum/rr*: is it a character device for volume *r* or a block device for volume *rr*? Currently Vinum does not address this conflict: the first-defined volume will get the name.
- A directory */dev/vinum/drive* with entries for each drive. These entries are in fact symbolic links to the corresponding disk nodes.
- A directory */dev/vinum/volume* with entries for each volume. It contains subdirectories for each plex, which in turn contain subdirectories for their component subdisks.
- The directories */dev/vinum/plex* and */dev/vinum/sd*, */dev/vinum/rsd*, which contain block device nodes for each plex and block and character device nodes respectively for subdisk.

For example, consider the following configuration file:

```
drive drive1 device /dev/sd1h
drive drive2 device /dev/sd2h
drive drive3 device /dev/sd3h
drive drive4 device /dev/sd4h
volume s64 setupstate
 plex org striped 64k
   sd length 100m drive drive1
   sd length 100m drive drive2
   sd length 100m drive drive3
   sd length 100m drive drive4
```

After processing this file, *vinum(8)* creates the following structure in */dev/vinum*:

```
brwx------  1 root  wheel   25, 0x40000001 Apr 13 16:46 Control
brwx------  1 root  wheel   25, 0x40000002 Apr 13 16:46 control
brwx------  1 root  wheel   25, 0x40000000 Apr 13 16:46 controld
drwxr-xr-x  2 root  wheel       512 Apr 13 16:46 drive
drwxr-xr-x  2 root  wheel       512 Apr 13 16:46 plex
crwxr-xr--  1 root  wheel   91,   2 Apr 13 16:46 rs64
drwxr-xr-x  2 root  wheel       512 Apr 13 16:46 rsd
drwxr-xr-x  2 root  wheel       512 Apr 13 16:46 rvol
brwxr-xr--  1 root  wheel   25,   2 Apr 13 16:46 s64
drwxr-xr-x  2 root  wheel       512 Apr 13 16:46 sd
drwxr-xr-x  3 root  wheel       512 Apr 13 16:46 vol

/dev/vinum/drive:
total 0
lrwxr-xr-x  1 root  wheel  9 Apr 13 16:46 drive1 -> /dev/sd1h
lrwxr-xr-x  1 root  wheel  9 Apr 13 16:46 drive2 -> /dev/sd2h
lrwxr-xr-x  1 root  wheel  9 Apr 13 16:46 drive3 -> /dev/sd3h
lrwxr-xr-x  1 root  wheel  9 Apr 13 16:46 drive4 -> /dev/sd4h

/dev/vinum/plex:
total 0
brwxr-xr--  1 root  wheel   25, 0x10000002 Apr 13 16:46 s64.p0

/dev/vinum/rsd:
```

```
total 0
crwxr-xr--  1 root  wheel   91, 0x20000002 Apr 13 16:46 s64.p0.s0
crwxr-xr--  1 root  wheel   91, 0x20100002 Apr 13 16:46 s64.p0.s1
crwxr-xr--  1 root  wheel   91, 0x20200002 Apr 13 16:46 s64.p0.s2
crwxr-xr--  1 root  wheel   91, 0x20300002 Apr 13 16:46 s64.p0.s3

/dev/vinum/rvol:
total 0
crwxr-xr--  1 root  wheel   91,   2 Apr 13 16:46 s64

/dev/vinum/sd:
total 0
brwxr-xr--  1 root  wheel   25, 0x20000002 Apr 13 16:46 s64.p0.s0
brwxr-xr--  1 root  wheel   25, 0x20100002 Apr 13 16:46 s64.p0.s1
brwxr-xr--  1 root  wheel   25, 0x20200002 Apr 13 16:46 s64.p0.s2
brwxr-xr--  1 root  wheel   25, 0x20300002 Apr 13 16:46 s64.p0.s3

/dev/vinum/vol:
total 1
brwxr-xr--  1 root  wheel   25,   2 Apr 13 16:46 s64
drwxr-xr-x  3 root  wheel       512 Apr 13 16:46 s64.plex

/dev/vinum/vol/s64.plex:
total 1
brwxr-xr--  1 root  wheel   25, 0x10000002 Apr 13 16:46 s64.p0
drwxr-xr-x  2 root  wheel       512 Apr 13 16:46 s64.p0.sd

/dev/vinum/vol/s64.plex/s64.p0.sd:
total 0
brwxr-xr--  1 root  wheel   25, 0x20000002 Apr 13 16:46 s64.p0.s0
brwxr-xr--  1 root  wheel   25, 0x20100002 Apr 13 16:46 s64.p0.s1
brwxr-xr--  1 root  wheel   25, 0x20200002 Apr 13 16:46 s64.p0.s2
brwxr-xr--  1 root  wheel   25, 0x20300002 Apr 13 16:46 s64.p0.s3
```

Although it is recommended that plexes and subdisks should not be allocated specific names, Vinum drives must be named. This makes it possible to move a drive to a different location and still recognize it automatically. Drive names may be up to 32 characters long.

Creating file systems

Volumes appear to the system to be identical to disks, with one exception. Unlike UNIX drives, Vinum does not partition volumes, which thus do not contain a partition table. This has required modification to some disk utilities, notably *newfs*, which previously tried to interpret the last letter of a Vinum volume name as a partition identifier. For example, a disk drive may have a name like */dev/wd0a* or */dev/da2h*. These names represent the first partition (`a`) on the first (0) IDE disk (`wd`) and the eight partition (`h`) on the third (2) SCSI disk (`da`) respectively. By contrast, a Vinum volume may be called */dev/vinum/concat*, a name which has no relationship with a partition name.

Normally, *newfs(8)* interprets the name of the disk and complains if it cannot understand it. For example:

```
# newfs /dev/vinum/concat
newfs: /dev/vinum/concat: can't figure out file system partition
```

In order to create a file system on this volume, use the `-v` option to *newfs(8)*:

```
# newfs -v /dev/vinum/concat
```

Startup

Vinum stores configuration information on the disk slices in essentially the same form as in the configuration files. When reading from the configuration database, Vinum recognizes a number of keywords which are not allowed in the configuration files. For example, a disk configuration might contain to following text, which has been wrapped to fit on the page:

```
volume myvol state up
volume bigraid state down
plex name myvol.p0 state up org concat vol myvol
plex name myvol.p1 state up org concat vol myvol
plex name myvol.p2 state init org striped 512b vol myvol
plex name bigraid.p0 state initializing org raid5 512b vol bigraid
sd name myvol.p0.s0 drive a plex myvol.p0 state up len 1048576b driveoffset 265b plexoffs
et 0b
sd name myvol.p0.s1 drive b plex myvol.p0 state up len 1048576b driveoffset 265b plexoffs
et 1048576b
sd name myvol.p1.s0 drive c plex myvol.p1 state up len 1048576b driveoffset 265b plexoffs
et 0b
sd name myvol.p1.s1 drive d plex myvol.p1 state up len 1048576b driveoffset 265b plexoffs
et 1048576b
sd name myvol.p2.s0 drive a plex myvol.p2 state init len 524288b driveoffset 1048841b ple
xoffset 0b
sd name myvol.p2.s1 drive b plex myvol.p2 state init len 524288b driveoffset 1048841b ple
xoffset 524288b
sd name myvol.p2.s2 drive c plex myvol.p2 state init len 524288b driveoffset 1048841b ple
xoffset 1048576b
sd name myvol.p2.s3 drive d plex myvol.p2 state init len 524288b driveoffset 1048841b ple
xoffset 1572864b
sd name bigraid.p0.s0 drive a plex bigraid.p0 state initializing len 4194304b driveoffset
 1573129b plexoffset 0b
sd name bigraid.p0.s1 drive b plex bigraid.p0 state initializing len 4194304b driveoffset
 1573129b plexoffset 4194304b
sd name bigraid.p0.s2 drive c plex bigraid.p0 state initializing len 4194304b driveoffset
 1573129b plexoffset 8388608b
sd name bigraid.p0.s3 drive d plex bigraid.p0 state initializing len 4194304b driveoffset
 1573129b plexoffset 12582912b
sd name bigraid.p0.s4 drive e plex bigraid.p0 state initializing len 4194304b driveoffset
 1573129b plexoffset 16777216b
```

The obvious differences here are the presence of explicit location information and naming (both of which are also allowed, but discouraged, for use by the user) and the information on the states (which are not available to the user). Vinum does not store information about drives in the configuration information: it finds the drives by scanning the configured disk drives for partitions with a Vinum label. This enables Vinum to identify drives correctly even if they have been assigned different UNIX drive IDs.

At system startup, Vinum reads the configuration database from one of the Vinum drives. Under normal circumstances, each drive contains an identical copy of the configuration database, so it does not matter which drive is read. After a crash, however, Vinum must determine which drive was updated most recently and read the configuration from this drive.

Performance issues

At present only superficial performance measurements have been made. They show that the performance is very close to what could be expected from the underlying disk driver performing the same operations as Vinum performs: in other words, the overhead of Vinum itself is negligible. This does not mean that Vinum has perfect performance: the choice of requests has a strong impact on the overall subsystem performance, and there are some known areas which could be improved upon. In addition, the user can influence performance by the design of the volumes.

The following sections examine some factors which influence performance.

> **Note:**The performance measurements in this section were done on some very old pre-SCSI-1 disk drives. The absolute performance is correspondingly poor. The intention of the following graphs is to show relative performance, not absolute performance. Other tests indicate that the performance relationships also apply to modern high-end hardware.

The influence of stripe size

In striped and RAID-5 plexes, the stripe size has a significant influence on performance. In all plex structures except a single-subdisk plex (which by definition is concatenated), the possibility exists that a single transfer to or from a volume will be remapped into more than one physical I/O request. This is never desirable, since the average latency for multiple transfers is always larger than the average latency for single transfers to the same kind of disk hardware. Spindle synchronization does not help here, since there is no deterministic relationship between the positions of the data blocks on the different disks. Within the bounds of the current BSD I/O architecture (maximum transfer size 128 kB) and current disk hardware, this increase in latency can easily offset any speed increase in the transfer.

In the case of a concatenated plex, this remapping occurs only when a request overlaps a subdisk boundary. In a striped or RAID-5 plex, however, the probability is an inverse function of the stripe size. For this reason, a stripe size of 256 kB appears to be optimum: it is small enough to create a relatively random mapping of file system hot spots to individual disks, and large enough to ensure than 95% of all transfers involve only a single data subdisk. Figure 10 shows the effect of stripe size on read and write performance, obtained with *rawio* [rawio]. This measurement used eight concurrent processes to access volumes with striped plexes with different stripe sizes. The graph shows the disadvantage of small stripe sizes, which can cause a significant performance degradation even compared to a single disk.

The influence of RAID-1 mirroring

Mirroring has different effects on read and write throughput. A write to a mirrored volume causes writes to each plex, so write performance is less than for a non-mirrored volume. A read from a mirrored volume, however, reads from only one plex, so read performance can improve.

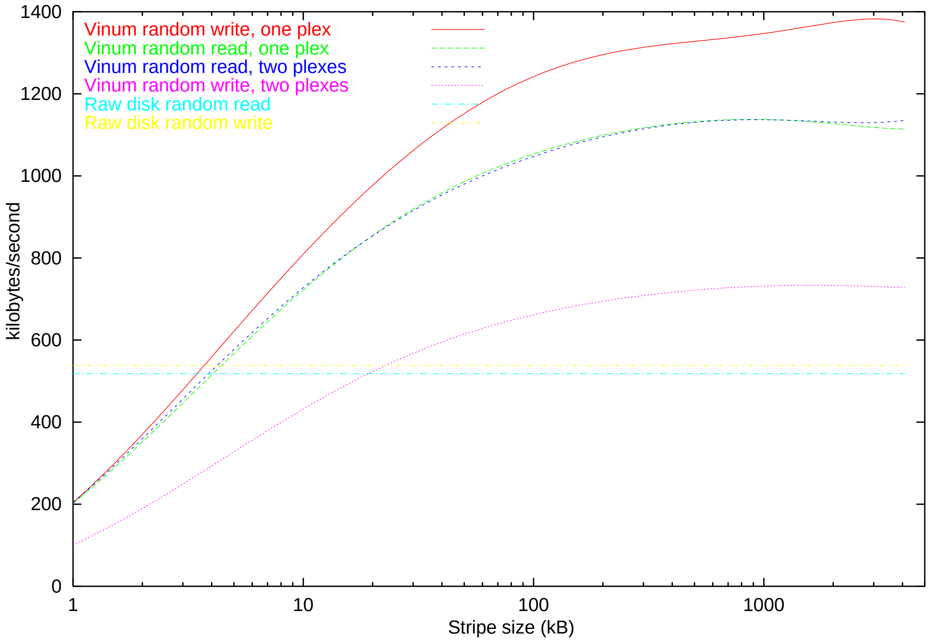

Figure 10: The influence of stripe size and mirroring

There are two different scenarios for these performance changes, depending on the layout of the subdisks comprising the volume. Two basic possiblities exist for a mirrored, striped plex.

One disk per subdisk

The optimum layout, both for reliability and for performance, is to have each subdisk on a separate disk. An example might be the following configuration, similar to the configuration on page 15:

```
volume raid10
  plex org striped 512k
    sd length 102480k drive a
    sd length 102480k drive b
    sd length 102480k drive c
    sd length 102480k drive d
  plex org striped 512k
    sd length 102480k drive e
    sd length 102480k drive f
    sd length 102480k drive g
    sd length 102480k drive h
```

In this case, the volume is spread over a total of eight disks. This has the following effects:

- Read access: by default, read accesses will alternate across the two plexes, giving a performance improvement close to 100%.

- Write access: writes must be performed to both disks, doubling the bandwidth requirement. Since the available bandwidth is also double, there should be little difference in througput.

At present, due to lack of hardware, no tests have been made of this configuration.

Both plexes on the same disks

An alternative layout is to spread the subdisks of each plex over the same disks:

```
volume raid10
  plex org striped 512k
    sd length 102480k drive a
    sd length 102480k drive b
    sd length 102480k drive c
    sd length 102480k drive d
  plex org striped 512k
    sd length 102480k drive c
    sd length 102480k drive d
    sd length 102480k drive a
    sd length 102480k drive b
```

This has the following effects:

- Read access: by default, read accesses will alternate across the two plexes. Since there is no increase in bandwidth, there will be little difference in performance through the second plex.
- Write access: writes must be performed to both disks, doubling the bandwidth requirement. In this case, the bandwidth has not increase, so write throughput will decrease by approximately 50%.

Figure 10 also shows the effect of mirroring in this manner. The results are very close to the theoretical predictions.

The influence of request size

As seen on page 2, the throughput of a disk subsystem is the sum of the latency (the time taken to position the disk hardware over the correct part of the disk) and the time to transfer the data to or from the disk. Since latency is independent of transfer size, overall throughput is strongly dependent on the size of the transfer, as Figure 11 shows. Unfortunately, there is little that can be done to influence the transfer size. In FreeBSD, it tends to be closer to 10 kB than to 30 kB.

The influence of concurrency

Vinum aims to give best performance for a large number of concurrent processes performing random access on a volume. Figure 12 shows the relationship between number of processes and throughput for a raw disk volume and a Vinum volume striped over four such disks with between one and 128 concurrent processes with an average transfer size of 16 kB. The actual transfers varied between 512 bytes and 32 kB, which roughly corresponds to UFS usage.

This graph clearly shows the differing effects of multiple concurrent processes on the Vinum volume and the relative lack of effect on a single disk. The single disk is saturated

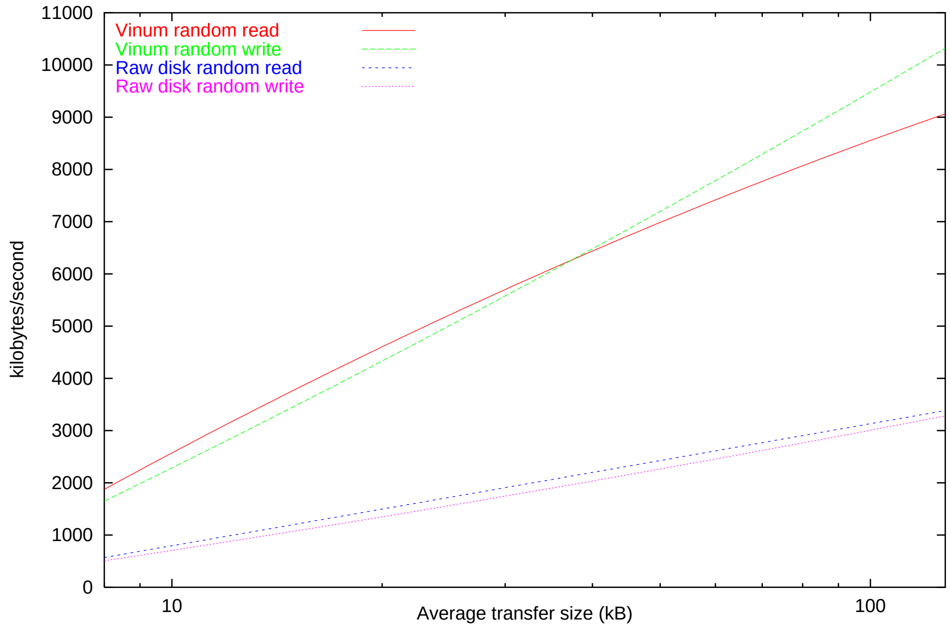

Figure 11: Throughput as function of transfer size

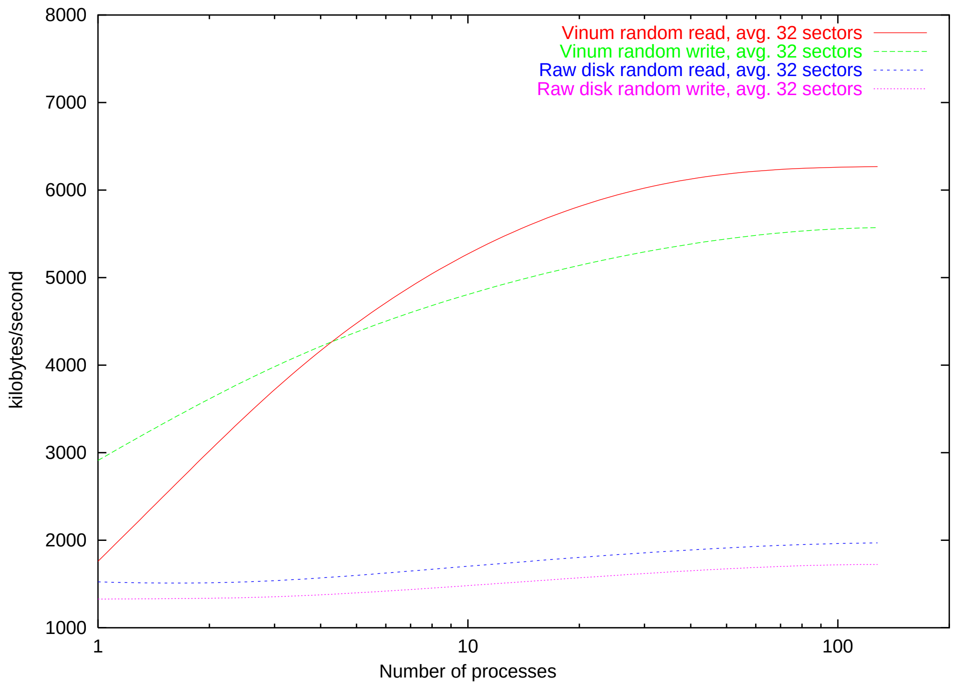

Figure 12: Concurrent random access with 32 sector transfers

even with one process, while Vinum shows a continual throughput improvement with up to 128 processes, by which time it has practically leveled off.

The influence of request structure

For concatenated and striped plexes, Vinum creates request structures which map directly to the user-level request buffers. The only additional overhead is the allocation of the request structure, and the possibility of improvement is correspondingly small.

With RAID-5 plexes, the picture is very different. The strategic choices described above work well when the total request size is less than the stripe width. By contrast, consider the following transfer of 32.5 kB:

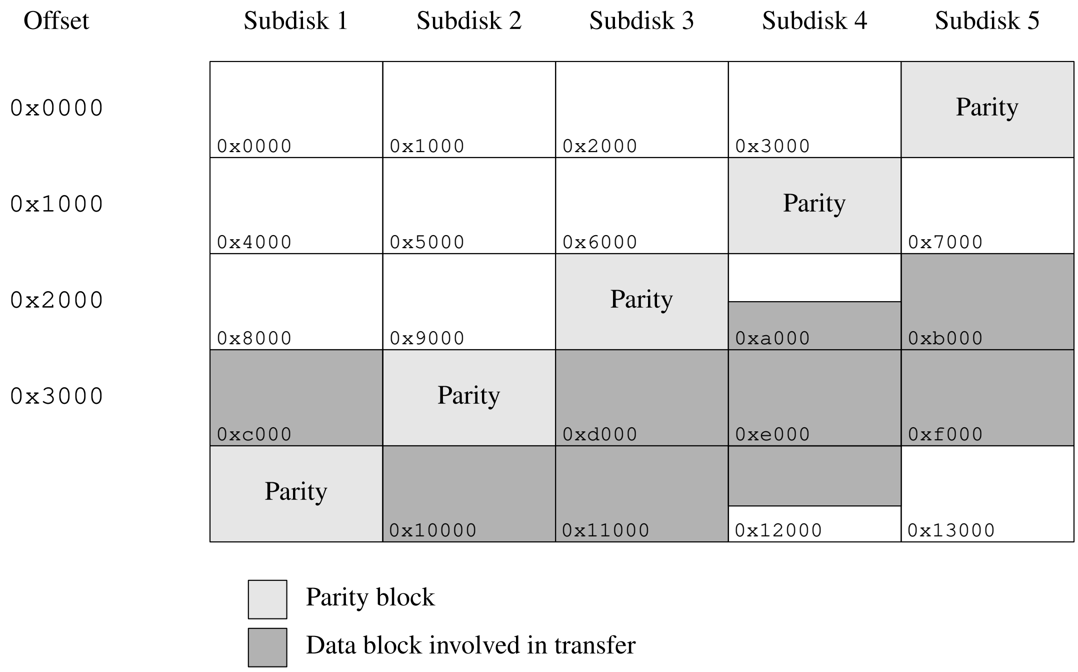

Figure 13: A sample RAID-5 transfer

An optimum approach to reading this data performs a total of 5 I/O operations, one on each subdisk. By contrast, Vinum treats this transfer as three separate transfers, one per stripe, and thus performs a total of 9 I/O transfers.

In practice, this inefficiency should not cause any problems: as discussed above, the optimum stripe size is larger than the maximum transfer size, so this situation does not arise when an appropriate stripe size is chosen.

These considerations are shown in figure 14, which clearly shows the RAID-5 tradeoffs:

- The RAID-5 write throughput is approximately half of the RAID-1 throughput in figure 10, and one-quarter of the write throughput of a striped plex.
- The read throughput is similar to that of striped volume of the same size.

Although the random access performance increases continually with increasing stripe size, the sequential access performance peaks at about 20 kB for writes and 35 kB for

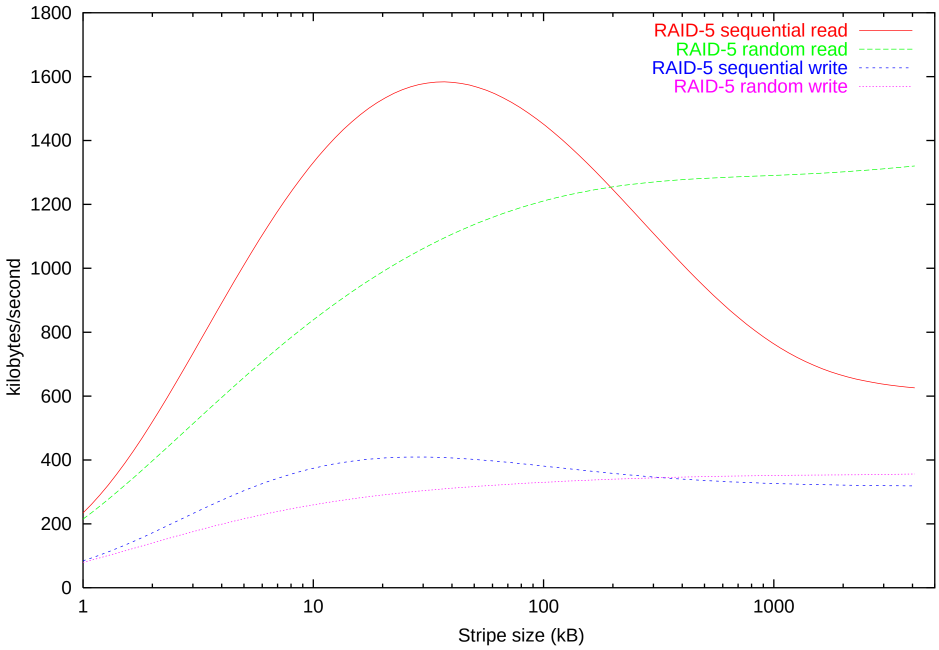

Figure 14: RAID-5 performance against stripe size

reads. This effect has not yet been adequately explained, but may be due to the nature of the test (8 concurrent processes writing the same data at the same time).

The implementation

At the time of writing, some aspects of the implementation are still subject to change. This section examines some of the more interesting tradeoffs in the implementation.

Where the driver fits

To the operating system, Vinum looks like a block device, so it is normally be accessed as a block device. Instead of operating directly on the device, it creates new requests and passes them to the real device drivers. Conceptually it could pass them to other Vinum devices, though this usage makes no sense and would probably cause problems. The following figure, borrowed from [McKusick],[1] shows the standard 4.4BSD I/O structure:

1. This figure is © 1996 Addison-Wesley, and is reproduced with permission.

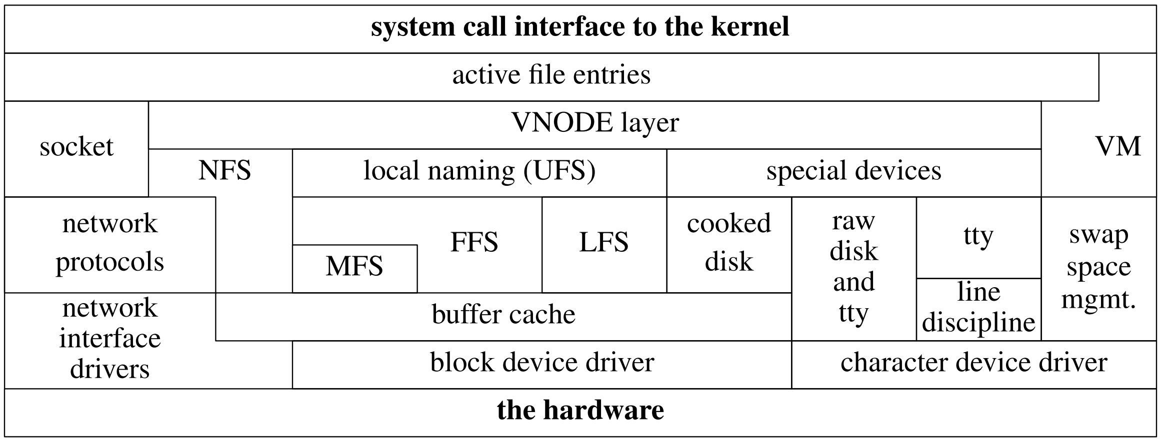

Figure 15: Kernel I/O structure, after McKusick et. al.

The following figure shows the I/O structure in FreeBSD after installing Vinum. Apart from the effect of Vinum, it shows the gradual lack of distinction between block and character devices that has occurred since the release of 4.4BSD: FreeBSD implements disk character devices in the corresponding block driver.

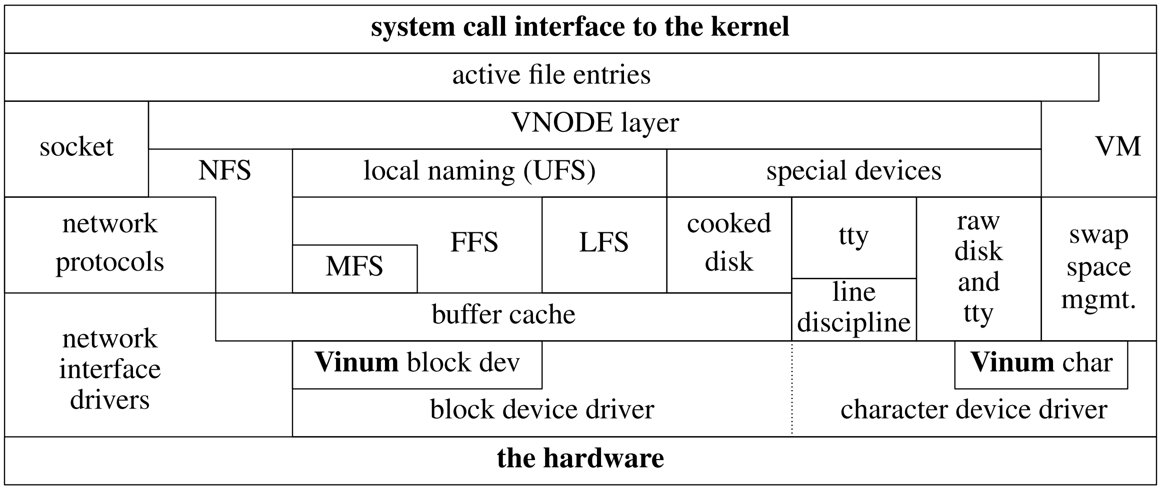

Figure 16: Kernel I/O structure with Vinum

Design limitations

Vinum was intended to have as few arbitrary limits as possible consistent with an efficient implementation. Nevertheless, a number of limits were imposed in the interests of efficiency, mainly in connection with the device minor number format. These limitations will no longer be relevant after the introduction of a device file system.

Restriction	Reasoning
Fixed maximum number of volumes per system.	In order to maintain compatibility with other versions of UNIX, it was considered desirable to keep the device numbers of volume in the traditional 8+8 format (8 bits major number, 8 bits minor number). This restricts the number of volumes to 256. In view of the fact that Vinum provides for larger volumes than disks, and current machines are seldom able to control more than 64 disk drives, this restriction seems unlikely to become severe for some years to come.
Fixed number of plexes per volume	Plexes supply redundancy according to RAID-1. For this purpose, two plexes are sufficient under normal circumstances. For rebuilding and archival purposes, additional plexes can be useful, but it is difficult to find a situation where more than four plexes are necessary or useful. On the other hand, additional plexes beyond four bring little advantage for reading and a significant disadvantage for writing. I believe that eight plexes are ample.
Fixed maximum number of subdisks per plex.	For similar reasons, the number of subdisks was limited to 256. It seldom makes sense to have more than about 10 subdisks per plex, so this restriction does not currently appear severe. There is no specific overall limitation on the number of subdisks.
Minimum device size	A device must contain at least 1 MB of storage. This assumption makes it possible to dispense with some boundary condition checks. Vinum requires 133 kB of disk space to store the header and configuration information, so this restriction does not appear serious.

Memory allocation

In order to perform its functionality, Vinum allocates a large number of dynamic data structures. Currently these structures are allocated by calling kernel `malloc`. This is a potential problem, since `malloc` interacts with the virtual memory system and may trigger a page fault. The potential for a deadlock exists if the page fault requires a transfer to a Vinum volume. It is probable that Vinum will modify its allocation strategy by reserving a small number of buffers when it starts and using these if a `malloc` request fails.

To cache or not to cache

Traditionally, UNIX block devices are accessed from the file system via caching routines such as *bread* and *bwrite*. It is also possible to access them directly, but this facility is seldom used. The use of caching enables significant improvements in performance.

Vinum does not cache the data it passes to the lower-level drivers. It would also seem counterproductive to do so: the data is available in cache already, and the only effect of caching it a second time would be to use more memory, thus causing more frequent cache misses.

RAID-5 plexes pose a problem to this reasoning. A RAID-5 write normally first reads the parity block, so there might be some advantage in caching at least the parity blocks. This issue has been deferred for further study.

Access optimization

The algorithms for RAID-5 access are surprisingly complicated and require a significant amount of temporary data storage. To achieve reasonable performance, they must take error recovery strategies into account at a low level. A RAID 5 access can require one or more of the following actions:

- *Normal read*. All participating subdisks are up, and the transfer can be made directly to the user buffer.
- *Recovery read*. One participating subdisk is down. To recover data, all the other subdisks, including the parity subdisk, must be read. The data is recovered by exclusive-oring all the other blocks.
- *Normal write*. All the participating subdisks are up. This write proceeds in four phases:
  1. Read the old contents of each block and the parity block.
  2. ‘‘Remove’’ the old contents from the parity block with exclusive or.
  3. ‘‘Insert’’ the new contents of the block in the parity block, again with exclusive or.
  4. Write the new contents of the data blocks and the parity block. The data block transfers can be made directly from the user buffer.
- *Degraded write* where the data block is not available. This requires the following steps:
  1. Read in all the other data blocks, excluding the parity block.

2. Recreate the parity block from the other data blocks and the data to be written.

3. Write the parity block.

- *Parityless write*, a write where the parity block is not available. This is in fact the simplest: just write the data blocks. This can proceed directly from the user buffer.

Combining access strategies

In practice, a transfer request may combine the actions above. In particular:

- A read request may request reading both available data (normal read) and non-available data (recovery read). This can be a problem if the address ranges of the two reads do not coincide: the normal read must be extended to cover the address range of the recovery read, and must thus be performed out of malloced memory.
- Combination of degraded data block write and normal write. The address ranges of the reads may also need to be extended to cover all participating blocks.

An exception exists when the transfer is shorter than the width of the stripe and is spread over two subdisks. In this case, the subdisk addresses do not overlap, so they are effectively two separate requests.

Examples

The following examples illustrate these concepts:

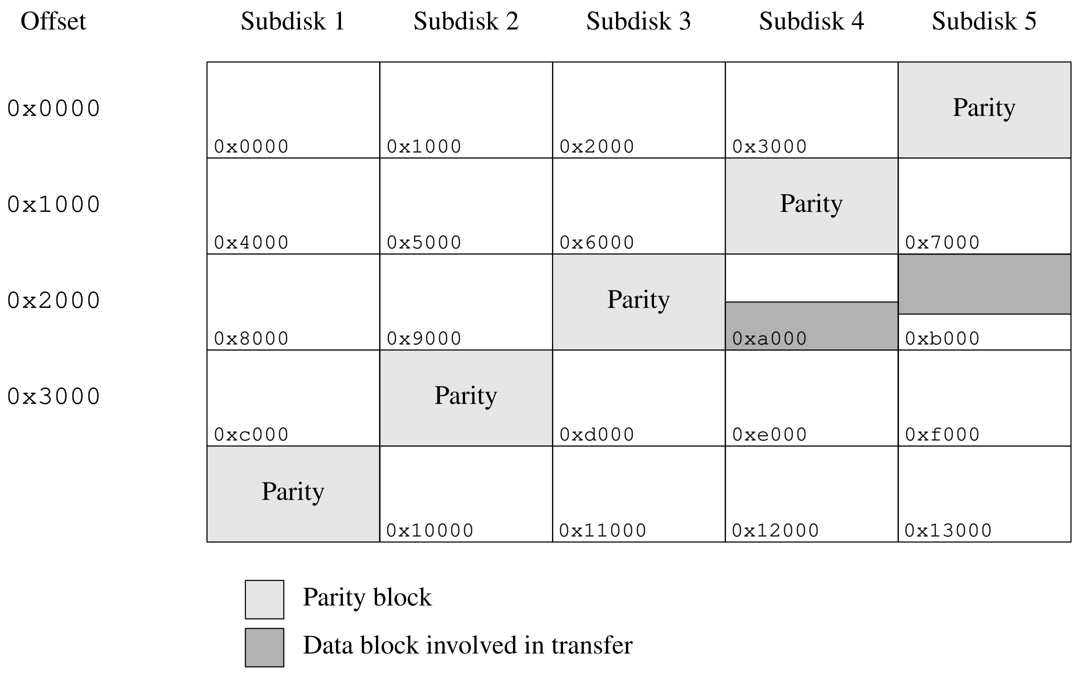

Figure 17: A sample RAID-5 transfer

Figure 17 illustrates a number of typical points about RAID-5 transfers. It shows the beginning of a plex with five subdisks and a stripe size of 4 kB. The shaded area shows the area involved in a transfer of 4.5 kB (9 sectors), starting at offset `0xa800` in the plex. A read of this area generates two requests to the lower-level driver: 4 sectors from subdisk 4, starting at offset `0x2800`, and 5 sectors from subdisk 5, starting at offset `0x2000`.

Writing this area is significantly more complicated. From a programming standpoint, the simplest approach is to consider the transfers individually. This would create the following requests:

- Read the old contents of 4 sectors from subdisk 4, starting at offset `0x2800`.
- Read the old contents of 4 sectors from subdisk 3 (the parity disk), starting at offset `0x2800`.
- Perform an exclusive OR of the data read from subdisk 4 with the data read from subdisk 3, storing the result in subdisk 3's data buffer. This effectively "removes" the old data from the parity block.
- Perform an exclusive OR of the data to be written to subdisk 4 with the data read from subdisk 3, storing the result in subdisk 3's data buffer. This effectively "adds" the new data to the parity block.
- Write the new data to 4 sectors of subdisk 4, starting at offset `0x2800`.
- Write 4 sectors of new parity data to subdisk 3 (the parity disk), starting at offset `0x2800`.
- Read the old contents of 5 sectors from subdisk 5, starting at offset `0x2000`.
- Read the old contents of 5 sectors from subdisk 3 (the parity disk), starting at offset `0x2000`.
- Perform an exclusive OR of the data read from subdisk 5 with the data read from subdisk 3, storing the result in subdisk 3's data buffer. This effectively "removes" the old data from the parity block.
- Perform an exclusive OR of the data to be written to subdisk 5 with the data read from subdisk 3, storing the result in subdisk 3's data buffer. This effectively "adds" the new data to the parity block.
- Write the new data to 5 sectors of subdisk 5, starting at offset `0x2000`.
- Write 5 sectors of new parity data to subdisk 3 (the parity disk), starting at offset `0x2000`.

This approach is clearly suboptimal. The operation involves a total of 8 I/O operations and transfers 36 sectors of data. In addition, the two halves of the operation block each other, since each must access the same data on the parity subdisk. Vinum optimizes this access in the following manner:

- Read the old contents of 4 sectors from subdisk 4, starting at offset `0x2800`.

- Read the old contents of 5 sectors from subdisk 5, starting at offset `0x2000`.
- Read the old contents of 8 sectors from subdisk 3 (the parity disk), starting at offset `0x2000`. This represents the complete parity block for the stripe.
- Perform an exclusive OR of the data read from subdisk 4 with the data read from subdisk 3, starting at offset `0x800` into the buffer, and storing the result in the same place in subdisk 3's data buffer.
- Perform an exclusive OR of the data read from subdisk 5 with the data read from subdisk 3, starting at the beginning of the buffer, and storing the result in the same place in subdisk 3's data buffer offset.
- Perform an exclusive OR of the data to be written to subdisk 4 with the modified parity block, starting at offset `0x800` into the buffer, and storing the result in the same place in subdisk 3's data buffer.
- Perform an exclusive OR of the data to be written to subdisk 5 with the modified parity block, starting at the beginning of the buffer, and storing the result in the same place in subdisk 3's data buffer offset.
- Write the new data to 4 sectors of subdisk 4, starting at offset `0x2800`.
- Write the new data to 5 sectors of subdisk 5, starting at offset `0x2000`.
- Write the 8 sectors of new parity data to subdisk 3 (the parity disk), starting at offset `0x2000`.

This is still a lot of work, but by comparison with the non-optimized version, the number of I/O operations has been reduced to 6, and the number of sectors transferred is reduced by 2. The larger the overlap, the greater the saving. If the request had been for a total of 17 sectors, starting at offset `0x9800`, the unoptimized version would have performed 12 I/O operations and moved a total of 68 sectors, while the optimized version would perform 8 I/O operations and move a total of 50 sectors.

Degraded read

Figure 18 illustrates the situation where a data subdisk fails, in this case subdisk 4.
In this case, reading the data from subdisk 5 is trivial. Recreating the data from subdisk 4, however, requires reading all the remaining subdisks. Specifically,

- Read 4 sectors each from subdisks 1, 2 and 3, starting at offset `0x2800` in each case.
- Read 8 sectors from subdisk 5, starting at offset `0x2800`.
- Clear the user buffer area for the data corresponding to subdisk 4.
- Perform an "exclusive or" operation on this data buffer with data from subdisks 1, 2, 3, and the last four sectors of the data from subdisk 5.
- Transfer the first 5 sectors of data from the data buffer for subdisk 5 to the corresponding place in the user data buffer.

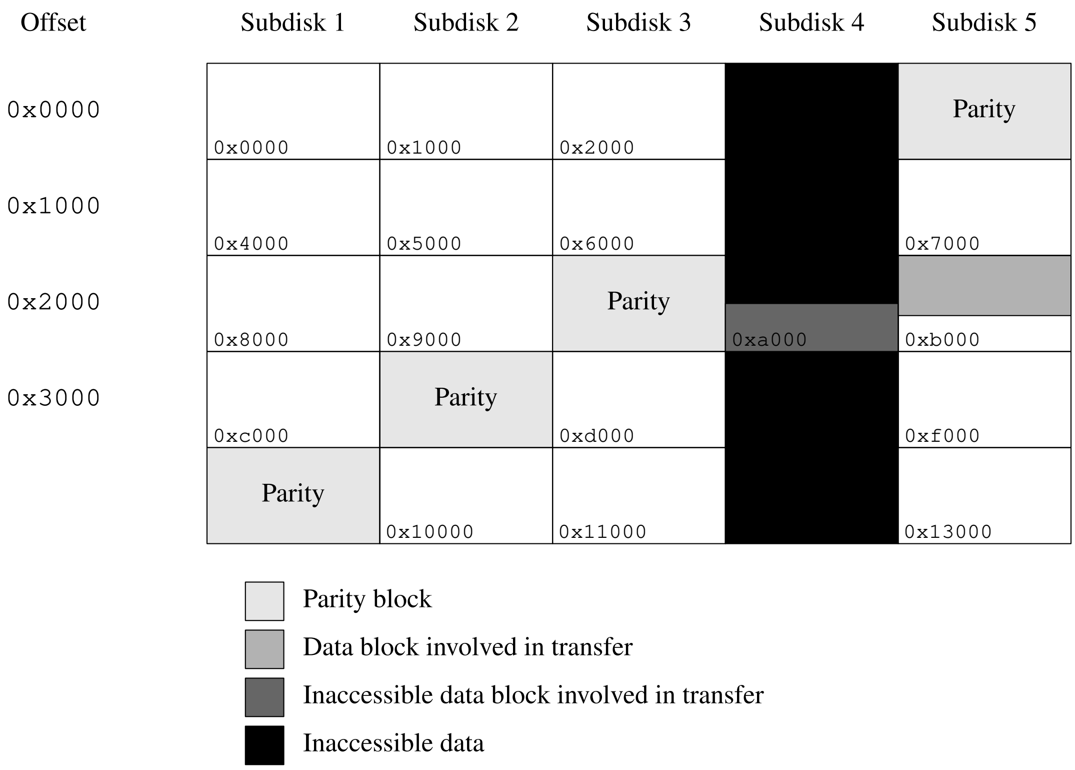

Figure 18: RAID-5 transfer with inaccessible data block

Degraded write

There are two different scenarios to be considered in a degraded write. Referring to the previous example, the operations required are a mixture of normal write (for subdisk 5) and degraded write (for subdisk 4). In detail, the operations are:

- Read 4 sectors each from subdisks 1 and 2, starting at offset `0x2800`, into temporary storage.
- Read 5 sectors from subdisk 3 (parity block), starting at offset `0x2000`, into the beginning of an 8 sector temporary storage buffer.
- Clear the last 3 sectors of the parity block.
- Read 8 sectors from subdisk 5, starting at offset `0x2000`, into temporary storage.
- ‘‘Remove’’ the first 5 sectors of subdisk 5 data from the parity block with exclusive or.
- Rebuild the last 3 sectors of the parity block by exclusive or of the corresponding data from subdisks 1, 2, 5 and the data to be written for subdisk 4.
- Write the parity block back to subdisk 3 (8 sectors).
- Write 5 sectors user data to subdisk 5.

Parityless write

Another situation arises when the subdisk containing the parity block fails, as shown in

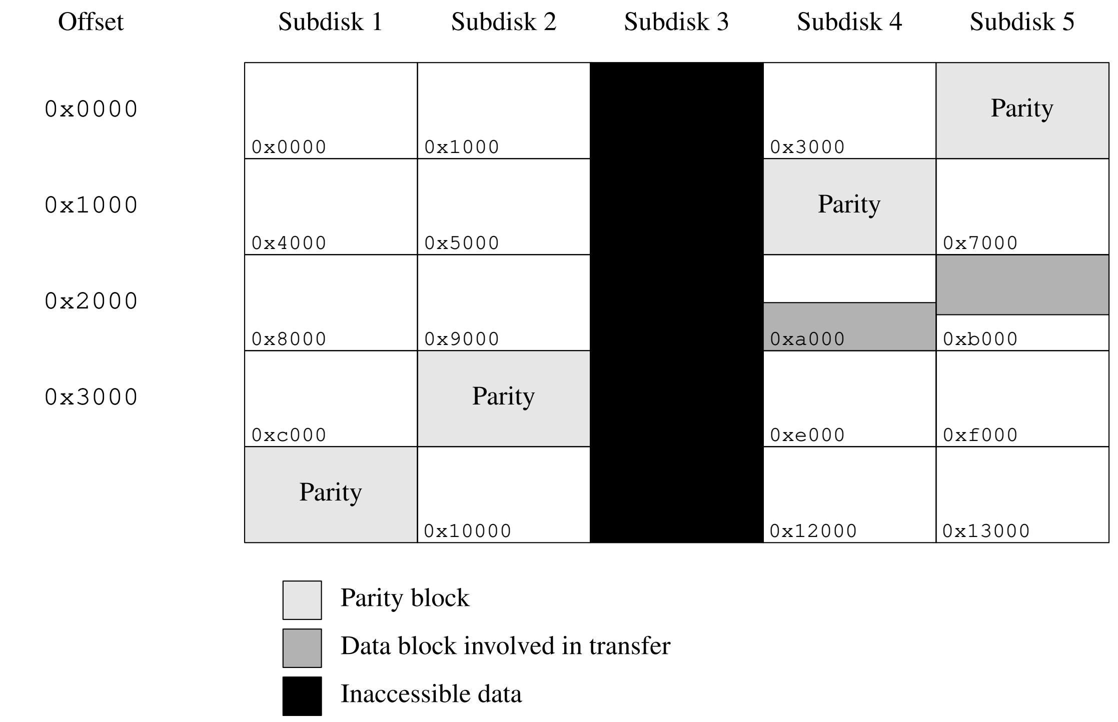

Figure 19: RAID-5 transfer with inaccessible parity block

figure 19. This configuration poses no problems on reading, since all the data is accessible. On writing, however, it is not possible to write the parity block. It is not possible to recover from this problem at the time of the write, so the write operation simplifies to writing only the data blocks. The parity block will be recreated when the subdisk is brought up again.

Driver structure

Vinum can issue multiple disk transfers for a single I/O request:

- As the result of striping or concatenation, the data for a single request may map to more than one drive. In this case, Vinum builds a request structure which issues all necessary I/O requests at one time. This behaviour has had the unexpected effect of highlighting problems with dubious SCSI hardware by imposing heavy activity on the bus.
- As seen above, many RAID-5 operations require a second set of I/O transfers after the initial transfers have completed.

- In case of an I/O failure on a resilient volume, Vinum must reschedule the I/O to a different plex.

The second set of RAID-5 operations and I/O recovery do not match well with the design of UNIX device drivers: typically, the ‘‘top half’’[1] of a UNIX device driver issues I/O commands and returns to the caller. The caller may choose to wait for completion, but one of the most frequent uses of a block device is where the virtual memory subsystem issues writes and does not wait for completion.

This poses a problem: who issues the second set of requests? The following possibilities, listed in order of increasing desirability, exist:

1. The top half can wait for completion of the first set of requests and then launch the second set before returning to the caller. This approach can seriously impact system performance and possibly cause deadlocks.

2. In a threaded kernel, the strategy routine can create a thread which waits for completion of the first set of requests and starts the second set without impacting the main thread of the process. At the moment this approach is not possible, since FreeBSD currently does not provide kernel thread support. It also appears likely that it could cause a number of problems in the areas of thread synchronization and performance.

3. Ownership of the requests can be ‘‘given’’ to another process, which will be awakened when they complete. This process can then issue the second set of requests. This approach is feasible, and it is used by some subsystems, notably NFS. It does not pose the same severe performance penalty of the previous possibility, but it does require that another process be scheduled twice for every I/O.

4. The second set of requests can be launched from the ‘‘bottom half’’ of the driver. This is potentially dangerous: the interrupt routine must call the `start` routine. While this is not expressly prohibited, the `start` routine is normally used by the top half of a driver, and may call functions which are prohibited in the bottom half.

Initially, Vinum used the fourth solution. This worked for most drivers, but some drivers required functions only available in the ‘‘top half’’, such as `malloc` for ISA bounce buffers. Current FreeBSD drivers no longer call these functions, but it is possible that the situation will arise again.

On the other hand, this method does not allow I/O recovery. Vinum now uses a dæmon process for I/O recovery and a couple of other housekeeping activities, such as saving the configuration database. The additional scheduling overhead for these activities is negligible, but it is the reason that the RAID-5 second stage does not use the dæmon.

---

1. UNIX device drivers run in two separate environments. The ‘‘top half’’ runs in the process context, while the ‘‘bottom half’’ runs in the interrupt context. There are severe restrictions on the functions that the bottom half of the driver can perform.

Availability

Vinum is available without RAID-5 functionality under a Berkeley-style copyright as part of the FreeBSD 3.1 distribution. It is also available at [vinum]. The RAID-5 functionality is available under licence from Cybernet, Inc. [Cybernet], and is included in their *NetMAX* Internet connection package.

Future directions

The current version of Vinum implements the core functionality. A number of additional features are under consideration:

- *Hot spare* capability: on the failure of a disk drive, the volume manager automatically recovers the data to another drive.
- *Logging* changes to a degraded volume. Rebuilding a plex usually requires copying the entire volume. In a volume with a high read to write, if a disk goes down temporarily and then becomes accessible again (for example, as the result of controller failure), most of the data is already present and does not need to be copied. Logging pinpoints which blocks require copying in order to bring the stale plex up to date.
- *Snapshots* of a volume. It is often useful to freeze the state of a volume, for example for backup purposes. A backup of a large volume can take several hours. It can be inconvenient or impossible to prohibit updates during this time. A snapshot solves this problem by maintaining *before images*, a copy of the old contents of the modified data blocks. Access to the plex reads the blocks from the snapshot plex if it contains the data, and from another plex if it does not.

  Implementing snapshots in Vinum alone would solve only part of the problem: there must also be a way to ensure that the data on the file system is consistent from a user standpoint when the snapshot is taken. This task involves such components as file systems and databases and is thus outside the scope of Vinum.
- A *SNMP interface* for central management of Vinum systems.
- A *GUI* interface is currently *not* planned, though it is relatively simple to program, since no kernel code is needed. As the number of failures testify, a good GUI interface is apparently very difficult to write, and it tends to gloss over important administrative aspects, so it's not clear that the advantages justify the effort. On the other hand, a graphical output of the configuration could be of advantage.
- An *extensible UFS*. It is possible to extend the size of some modern file systems after they have been created. Although UFS (the *UNIX File System*, previously called the Berkeley *Fast File System*) was not designed for such extension, it is trivial to implement extensibility. This feature would allow a user to add space to a file system which is approaching capacity by first adding subdisks to the plexes and then extending the file system.

- *Remote data replication* is of interest either for backup purposes or for read-only access at a remote site. From a conceptual viewpoint, it could be achieved by interfacing to a network driver instead of a local disk driver.
- *Extending striped and RAID-5 plexes* is a slow complicated operation, but it is feasible.

References

[CMD] CMD Technology, Inc June 1993, *The Need For RAID, An Introduction*. *http://www.fdma.com/info/raidinto.html*

[Cybernet] *The NetMAX Station*, *http://www.cybernet.com/netmax/index.html*. The first product using the Vinum Volume Manager.

[FreeBSD] FreeBSD home page, *http://www.FreeBSD.org/*

[IBM] *AIX Version 4.3 System Management Guide: Operating System and Devices, Logical Volume Storage Overview*
http://www.austin.ibm.com/doc_link/en_US/a_doc_lib/aixbman/baseadmn/lvm_overview.htm

[Linux] *Logical Volume Manager for Linux* , *http://linux.msede.com/lvm/*.

[McKusick] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*, Addison Wesley, 1996.

[OpenSource] *The Open Source Page*, *http://www.opensource.org/*

[rawio] A raw disk I/O benchmark. *ftp://ftp.lemis.com/pub/rawio.tar.gz*

[SCO] *SCO Virtual Disk Manager*, *http://www.sco.com/products/layered/ras/virtual.html*.

[Solstice] *http://www.sun.com/solstice/em-products/system/disksuite.html*

[veritas] The VERITAS® Volume manager,
http://www.veritas.com/product-info/vm/index.htm .

[vinum] Greg Lehey, *The Vinum Volume Manager*, *http://www.lemis.com/vinum.html*

[Wong] Brian Wong, *RAID: What does it mean to me?*, SunWorld Online, September 1995. *http://www.sunworld.com/sunworldonline/swol-09-1995/swol-09-raid5.html*