The Vinum Volume Manager

Greg Lehey
Nan Yang Computer Services Ltd.
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 the size 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, the FTP server *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.

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

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.

Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23

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, of all the core components of a server they are still the most likely 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 in the "Performance issues" section.

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. In RAID-5, the location of this parity block changes from one stripe to the next. Figure 3 shows the RAID-5 organization. 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 problems of size, performance and reliability 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. Striping (''RAID-0'') 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 which cannot be explained by the hardware configuration. 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'').
- 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.

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.

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

By spreading data across multiple disks, Vinum can deliver much higher performance than a single disk. This issue will be discussed in more detail below.

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 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 be extended. Vinum enforces a minimum of three subdisks for a RAID-5 plex, since any smaller number would not make any sense.

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 drive 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*

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 configurations.

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.

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.

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.

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.

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 probable future alternative is to rebuild the parity blocks, which allows better recovery of crashed plexes.

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.

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 eighth partition (h) on the third (2) SCSI disk (da) respectively. By contrast, a Vinum volume might 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 relating to object state which are not allowed in the configuration files. 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 with more than one subdisk, 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, which is seldom enough to be negligible. 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 99% of all transfers involve only a single data subdisk. Figure 4 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.

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 sample "RAID-10" configuration seen above.

```
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
```

In this configuration, the subdisks covering a specific plex address space have been placed on different drives, thus improving both performance and resilience This configuration has the following properties:

- 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 4 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 above, 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 and much larger than the transfer time for typical transfers, overall throughput is strongly dependent on the size of the transfer, as Figure 5 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 6 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 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, it does not perform optimally when a request is larger than the size of all blocks in a stripe. The requests map to contiguous space on the disk but non-contiguous space in the user buffer. An optimal implementation would perform one I/O request per drive and map to the user buffer in software. By contrast, Vinum performs separate I/O requests for each stripe.

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 arises only when an inappropriately small stripe size is chosen.

Figure 7 shows the RAID-5 tradeoffs:

- The RAID-5 write throughput is approximately half of the RAID-1 throughput in figure 4, 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 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).

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 ratio, 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 correct 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*. An extended version of this paper.

[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*

Illustrations

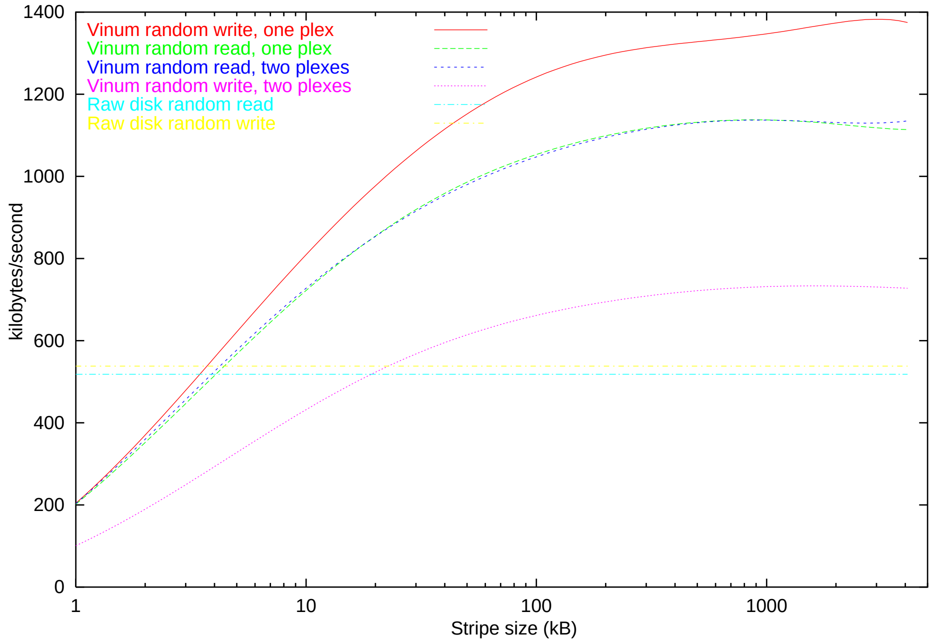

Figure 4: The influence of stripe size and mirroring

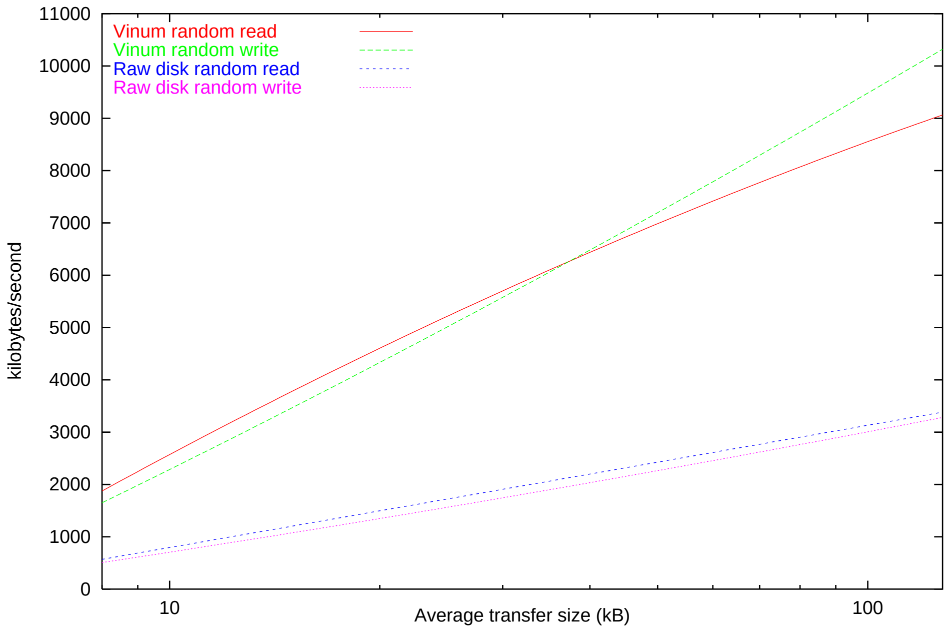

Figure 5: Throughput as function of transfer size

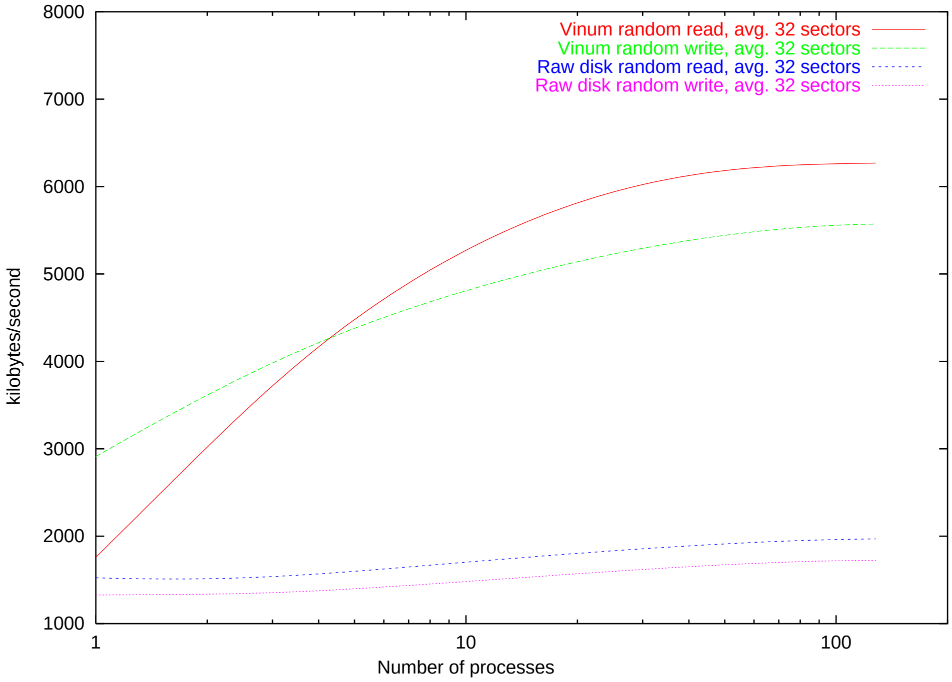

Figure 6: Concurrent random access

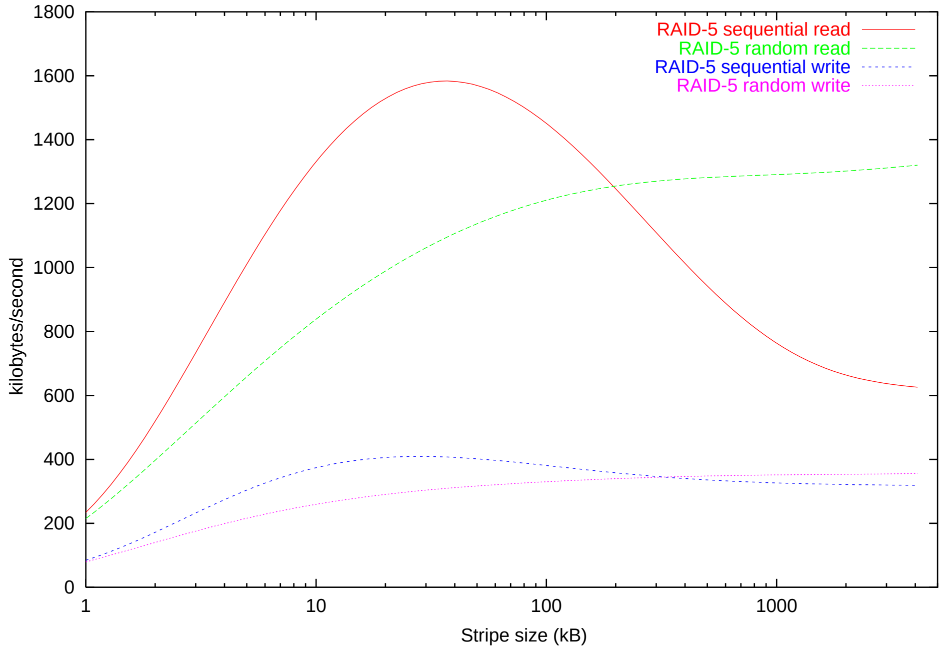

Figure 7: RAID-5 performance against stripe size