CHAPTER 13: TUNING STORAGE FOUNDATION

by Volker Herminghaus

13.1 Basics About Tuning Storage Foundation

Having read the chapter about SAN storage, Moore's law and the advancements in disk performance your expectations about **performance** tuning Storage Foundation should be low. After all, optimizing a resource that is aggressively used by hundreds of different hosts for thousands of different volumes, all accessing the same overloaded mechanically limited disk hardware at the same time is almost impossible because many optimizations that improve our application performance decrease everybody else's. Fortunately, VxVM and VxFS are extensively self-tuning. It is not usually necessary to tweak individual volumes or file systems to get very good performance out of them because they know some of the features of the underlying level and use them as best they can, usually without degrading aggregate performance. For instance, VxVM knows about your storage array's sweet spots because it identifies the type of LUN using its extensive array support libraries (ASLs). VxFS in turn knows about the layout of the VxVM volume and adapts its optimization parameters (parallelity, I/O-size etc.) to the volume layout as much as possible.

However it is still possible to base one's volume layouts on wrong assumptions which might lead to very poor performance. This is the part where we can help. We therefore limit this chapter to two areas that would not be considered **performance** tuning in the classical sense, but that certainly classify as **tuning** in the sense of adapting the product to best suit your requirements as well as to prevent anything that would actually hurt performance. The main points for the Easy Sailing part are:

V. Herminghaus and A. Sriba, *Storage Management in Data Centers*,
DOI: 10.1007/978-3-540-85023-6_13, © Springer-Verlag Berlin Heidelberg 2009

- Using reasonable volume parameters
- Setting these as defaults

As you can see, this is a rather short introduction into VxVM and VxFS tuning. Much more about performance tuning in general, benchmarking, and the limits of optimization can be found in this chapter's Technical Deep Dive section beginning on page 468.

13.1.1 Tuning VxVM by Using Reasonable Parameters

Remember what we said in the chapter about SAN storage arrays? Storage arrays are typically used by a massive amount of hosts at the same time. Very often hundreds and sometimes thousands of machines access the same storage array. Most arrays have a very large buffer cache, so they can buffer any writes coming in to the array. These writes will be received at very high speed by the storage array, so the host is freed from these write I/Os rather rapidly. One could say that any kind of write I/O is probably the best case for accessing a storage array, since no disk seek time must be waited for by the requesting machine. The limiting factor for write I/O are the latency of the array. This latency is in turn defined by two parameters:

1) The basic latency of the unloaded array. This one given by the array hardware processing capabilities
2) The amount of load on the array's front end controllers that insert the writes into the write queue and onto the back-end controllers that distribute the writes to the individual disks onto which the data is ultimately persisted.

We obviously cannot change the basic hardware features of the array (at least not by software tuning ;), so we'll have to concentrate on the second point: the load on the array's front end and back end controllers. How can we influence that parameter? The simple, and rather obvious, point here is to reduce the absolute number of I/Os as much as possible. If the array's controller only has to deal with 1,000 I/Os per second instead of 5,000 I/Os per second, the algorithms internal to the array will have less objects to handle, will have shorter queues to process, and subsequently will work more efficiently. This increased efficiency results in shorter latency from the storage array and therefore can increase performance for our host **without** hurting other participants.

How to Reduce the Number of I/Os?

The obvious point, and actually the point where the most performance could be gained, is: modify the application to read and write more wisely. Unfortunately, most application developers couldn't care less about storage; they are too far removed from the storage hardware. The application developers usually rely on the database staff and it is the database staff who the application developers will typically resort to when performance problems arise. But database people tend to be technically a little on the weak side, and from our experience many of them will still fall for the same ancient myths that had already been proved wrong ten years ago.

For instance, it used to be very hard to rid a typical Oracle DBA of the idea that using a stripe size identical to Oracle's block size (usually 8 KB) is a brilliant idea. It is indeed a terrible idea, as it creates a massive overload of extraneous I/O without improving any other area of the storage hierarchy. But do not be surprised if your DBA dispels this as a myth, citing some obscure Oracle document he received fifteen years ago.

So basically we are stuck with having to do the whole tuning ourselves: the application developers only talk to the guys with the antiquated ideas about storage systems, so we must make the best out of the situation.

How can we make the best out of the situation? We can, first of all, **not** stripe out volumes unless we expect highly localized high-speed traffic to our volumes. This is not normally the case. Striping will lead to extraneous I/O whenever a stripe column boundary is crossed. This will cause additional overhead in the storage array controllers. The only advantage we might get is increased load balancing across several physical disks. But this may not be an advantage at all. The disks that we are transferring load to are only partially used by our volume. the rest of their capacity is shared with other hosts using the same storage array. They have allocated their LUNs on the same physical disks as we have. If we stripe heavily, then we will split single large I/Os into more small I/Os, resulting in more load on the storage array's controllers. But apart from this we might hit this one unlucky disk that happens to be massively overloaded with scattered read requests (which can not be ameliorated by caching). This would drag down out performance to abysmal levels, even if the rest of the volume actually did gain something from striping (which it usually does not).

Up to five years ago we would have recommended you choose physical disk spindles inside the storage array for your application, create LUNs from them, and use them wisely. It was at that time possible to actually tune storage arrays to your application. Because of the extreme consequences of Moore's law in recent years it has become next to impossible to successfully implement this nowadays. The best you can do now is to restrain yourself to do only very limited striping (or preferably use concatenation instead) in order not to generate additional I/Os on the storage array controllers..

On the other hand we should not overdo it. If you choose to do all your I/O on a single LUN then the host resident request queue for this LUN (which looks to your host system like a physical disk) may grow very large, which hampers performance on the host side. It is not generally a clever idea to increase the maximum queue size on the host, by the way: The algorithms used for queues work very well on short queue length, but processing time may grow rapidly with increasing queue lengths. In this case it is better to distribute the I/O onto several LUNs.

Not striping, but still distributing load across several LUNs to keeps queue lengths

short may seem like an oxymoron, but actually it isn't: If you have control over the creation of the LUNs you can create several LUNs in such a way that they are on physically contiguous space. If you concatenate these LUNs (each of which may be many GBs in size), then you can have the best of both worlds: short queues on the host (at least for random I/O), and no additional load on the front and back end controllers in the storage array.

13.1.2 Understanding and Modifying VxVM Defaults

Defaults for various VxVM commands are stored (in Solaris) in the **/etc/defaults** directory, under the same name as the VxVM utility that the defaults pertain to. For instance, you could create a file **/etc/defaults/vxdg** to set the default type of Disk group (e.g. **cds** or not **cds**), or a file **/etc/default/vxassist** to set the defaults for volume and log characteristics.

Identifying vxassist Defaults

You can check which defaults vxassist will use by issuing the command vxassist help showattrs (show attributes) On a system with an empty defaults file you will get an output that looks similar to this:

```
# vxassist help showattrs
#Attributes:
layout=nomirror,nostripe,nomirror-stripe,nostripe-mirror,
 nostripe-mirror-col,nostripe-mirror-sd,noconcat-mirror,nomirror-concat,
 span,nocontig,raid5log,noregionlog,diskalign,nostorage
 mirrors=2 columns=0 regionlogs=1 raid5logs=1 dcmlogs=0 dcologs 0
 autogrow=no destroy=no sync=no
 min_columns=2 max_columns=8
 regionloglen=0 regionlogmaplen=0 raid5loglen=0 dcmloglen=0 logtype=region
 stripe_stripeunitsize=128 raid5_stripeunitsize=32
 stripe-mirror-col-trigger-pt=2097152 stripe-mirror-col-split-trigger-pt=2097152
 usetype=fsgen diskgroup= comment="" fstype=
 sal_user=
 user=0 group=0 mode=0600
 probe_granularity=2048
mirrorgroups (in the end)
 alloc=
 wantalloc=vendor:confine
 mirror=
 wantmirror=
 mirrorconfine=
 wantmirrorconfine=protection
 stripe=
 wantstripe=
 tmpalloc=
```

What you see in the layout line is that **vxassist** will use no mirroring (**nomirror**), no striping (**nostripe**), and no combination of any mirroring or striping, and that it will allow the spanning of disks (**span**), will align subdisks to cylinder boundaries (**diskalign**), etc.

The next line shows that if you specify a mirrored layout on the command line (i.e. if you create a mirrored volume) then this volume will have by default 2 mirrors (**mirrors=2**).

Two lines further down you see

```
min_columns=2 max_columns=8
```

This refers to the boundaries of the stripe column default. Let's say you specify a striped layout (although you should think twice about doing it - remember what we said about reducing scattered reads!) but you do not specify a number of columns. In that case VxVM picks the default as follows:

It divides the number of disks in the Disk group by two to allow for later mirroring the volume. Then, from the result it picks the highest number of columns possible that is within the limits given by **min_columns** and **max_columns**. If we were using a Disk group with six disks and we gave the following command:

```
vxassist make avol 1g layout=stripe
```

(i.e. no mirroring), it would create a three-column stripe. If our Disk group had fourteen disks, then using the same command it would create a seven-column stripe. And if there were many more disks in the Disk group, like thirty or forty, it would still never exceed eight, because that is the value of **max_columns**.

Changing vxassist Defaults

We can put our own default into a file and make vxassist use those defaults. The simplest way is to edit (or create) the file **/etc/default/vxassist**, which **vxassist** will use automatically unless you specify otherwise. If you like different sets of defaults for different tasks you could create several files and pass them to vxassist using the **-d $DEFAULTSFILE** parameter. In that case, vxassist will ignore **/etc/defaults/vxassist** and just read the default file that you passed it.

Format of the vxassist Defaults File

The defaults file for vxassist looks like a collection of command line parameters to vxassist. In principle, you could create a space-separated or newline-separated list of your favorite options. But the reality is more tricky, and there are important differences, which concern mirroring and striping: even if you set **nmirror=3** and **ncol=5** in the defaults file, these values will not be activated just because they are in the defaults file.

Only if the **vxassist** command actually receives the parameter appropriate for mirroring will the new default for **nmirror** be used. I.e. in the case given above, the volume will be a three-way mirror. And only if the parameter relevant for striping is used on the command line will the number of columns from the defaults file be applied.

So setting the **nmirror** attribute does not lead to mirroring, and setting the **ncol** attribute does not lead to striping. But there actually are ways to turn mirroring on by default.

One is to add "**mirror=yes**" to the defaults file. This will lead to all volumes, regardless of what is specified on the command line, being mirrored the default number of times (dependent on the rest of the file or the default that was compiled into **vxassist**).

While **mirror=yes** turns the default on for mirroring, there is no such parameter for striping. You cannot specify **stripe=yes** to coerce striping by default. You **can**, however, specify a default layout, using the "**layout=...**" parameter just like you used to one the command line. So with a defaults file like this:

```
# cat /etc/default/vxassist
columns=2
nmirror=3
```

You will get a concat volume unless you specify mirroring. If you do specify mirroring (**layout=mirror**) on the command line you will get a three-way mirror (because you changed the default value for the number of mirrors). But the volume will not be striped unless you also specify striping on the command line (**layout=stripe, layout=stripe-mirror** or similar). If you do specify striping, then the number of columns will default to two as given in the defaults file. If you add **mirror=yes** to the default file the volume will be mirrored. If you add layout=mirror instead, the volume will be mirrored, too. But if you add **layout=stripe-mirror**, it will be striped, but not mirrored. In that case you need an additional line adding **mirror=yes** to actually create striped and mirrored volumes be default. If this seems non-linear and counter-intuitive to you, then rest assured that you are not the only one who thinks so. But we cannot change the software, just explain it.

After you wrote your defaults file for **vxassist**, the output of the **vxassist showattrs** command will vary to reflect what you put into the defaults file. For instance, with a defaults file like this:

```
# cat /etc/default/vxassist
layout=stripe-mirror
mirror=yes
columns=2
nmirror=3
```

the output of **vxassist help showattrs** will change to resemble the new defaults:

```
# vxassist help showattrs
#Attributes:
 layout=mirror,nostripe,nomirror-stripe,stripe-mirror,
 nostripe-mirror-col,nostripe-mirror-sd, noconcat-mirror,nomirror-concat,
 span,nocontig,raid5log,noregionlog,diskalign,nostorage
 mirrors=3 columns=2 regionlogs=1 raid5logs=1 dcmlogs=0 dcologs 0
 autogrow=no destroy=no sync=no
 min_columns=2 max_columns=8
 regionloglen=0 regionlogmaplen=0 raid5loglen=0 dcmloglen=0 logtype=region
 stripe_stripeunitsize=128 raid5_stripeunitsize=32
 stripe-mirror-col-trigger-pt=2097152 stripe-mirror-col-split-trigger-pt=2097152
 usetype=fsgen diskgroup= comment="" fstype=
```

```
 sal_user=
 user=0 group=0 mode=0600
 probe_granularity=2048
mirrorgroups (in the end)
 alloc=
 wantalloc=vendor:confine
 mirror=
 wantmirror=
 mirrorconfine=
 wantmirrorconfine=protection
 stripe=
 wantstripe=
 tmpalloc=
```

13.1.3 Tuning VxFS

Tuning Extent Allocation

Tuning a VxFS file system can be done on two levels: One is making sure the extents that are allocated for the files are as contiguous as possible, i.e. files do not consist of hundreds of little, non-sequential snippets, but rather of a single, large block. The VxFS file system is very good at allocating contiguously when a file is written, as has been discussed in the appropriate section on page 436 of the file system chapter. But during the lifetime of a file system extents are constantly being rewritten, new extents allocated, old extents freed and so on. The result is that files end up consisting of little snippets after all. This happens to both UFS and VxFS file systems. It is worse if the file system is nearly full, because then the system is less free to find appropriate extents (or blocks) for the files and must resort to allocating e.g. several small extents far away from the existing file rather than a single large extent close by.

The standard UFS offers no way of handling this slow but certain deterioration of file system contiguousness (and therefore deterioration of performance) except copying the files away, creating a new file system, and copying the files back. This is usually unacceptable due to the downtime involved. The VxFS does have utilities that do extent (and directory) reallocation on the fly, on a active file system. In fact it is highly recommended to perform a reallocation run at regular intervals, like daily, weekly, or at least monthly. The total cost in I/O load is not very high, but application performance will not degrade, as it would otherwise. In one real example from 2006 a file system for Oracle table spaces had an average(!) number of several hundred extents allocated per file. That was reduced to an average of one(!) extent per file during three successive runs of optimization, each of which took only a few minutes. File system performance for sequential I/O was increased by a factor of ten. This is an extreme case, but because so few people know the tools for VxFS optimization we suspect that there several Petabyte of storage out there which have become extremely scattered and would indeed profit a lot from a regular optimization. Refer to the file system chapter for more information about the **fsadm** command and how

it can be used to optimize your file systems.

Tuning File System Parameters

VxFS defaults are dynamically created when the file system is started by reading (if applicable) the layout of the underlying volume and adapting several internal values to it (e.g. the maximum I/O size). But these values can be influenced by creating an entry in the file **/etc/vxfstunetab**. In this file, each line defines the VxFS tunables for one volume. You can read the initial value with **vxtunefs $RAWDEVICE**, then change their format to match the **vxfstunetab** file format (which unfortunately does not match at all), and then changing the values of the individual tunables for that file system. The next time the file system is mounted, these values will be applied.

Let's look at the tuning parameters for a newly created VxFS file system:

```
# vxassist -g adg make avol 1g
# mkfs -F vxfs /dev/vx/rdsk/adg/avol
    version 7 layout
    2097152 sectors, 1048576 blocks of size 1024, log size 16384 blocks
    largefiles supported
# mount -Fvxfs /dev/vx/dsk/adg/avol /mnt
# vxtunefs /mnt
Filesystem i/o parameters for /mnt
read_pref_io = 65536
read_nstream = 1
read_unit_io = 65536
write_pref_io = 65536
write_nstream = 1
write_unit_io = 65536
pref_strength = 10
buf_breakup_size = 1048576
discovered_direct_iosz = 262144
max_direct_iosz = 1048576
default_indir_size = 8192
qio_cache_enable = 0
write_throttle = 0
max_diskq = 1048576
initial_extent_size = 8
max_seqio_extent_size = 2048
max_buf_data_size = 8192
hsm_write_prealloc = 0
read_ahead = 1
inode_aging_size = 0
inode_aging_count = 0
fcl_maxalloc = 32537600
fcl_keeptime = 0
fcl_winterval = 3600
```

```
fcl_ointerval = 0
oltp_load = 0
```

The parameters reported by **vxtunefs** are actually derived from the volume layout. The VxFS specific mount command probes the underlying volume parameters and uses them to set the tunable parameters to something reasonable. Therefore it is seldom necessary to tune a VxFS file system for optimum performance with the underlying VxVM volume. At least this used to be the case if you were using physical disks rather than LUNs. You may find that some tuning may still be in order to adapt for application-specific I/O behavior or other special cases, even if the VxFS file system resides in a VxVM volume. If it does not reside in a VxVM volume but in a plain partition or a volume created by some other volume management product, then tuning is definitely a reasonable option. This is also true if you are using LUNs as the basis for your VxVM (or other) volumes, because the physical properties of LUNs allow much greater I/O sizes as well as greater parallelity than plain disks do.. Let's look at some volume layouts, their respective **tunefs**-parameters and how they change dependent on the volume layout.

First, we create four volumes: a concat volume, a stripe with 5 columns and a stripesize of 2048 blocks (or 1024k or 1 MB), a three-way mirror, and a stripe-mirror with three columns and two mirrors. We make VxFS file systems on them and mount them right away into directories with names corresponding to the volume layouts:

```
# mkdir /concat /stripe5col /mirror3way /stripemirror
# vxassist make concatvol 1g layout=concat
# mkfs -F vxfs /dev/vx/rdsk/adg/concatvol
    version 7 layout
    2097152 sectors, 1048576 blocks of size 1024, log size 16384 blocks
    largefiles supported
# mount -F vxfs /dev/vx/dsk/adg/concatvol /concat
# vxassist make stripe5colvol 1g layout=stripe ncol=5 stwid=1024k
# mkfs -F vxfs /dev/vx/rdsk/adg/stripe5colvol
    version 7 layout
    2097152 sectors, 1048576 blocks of size 1024, log size 16384 blocks
    largefiles supported
# mount -F vxfs /dev/vx/dsk/adg/stripe5colvol
# vxassist make mirror3wayvol 1g layout=mirror nmirror=3 init=active
# mkfs -F vxfs /dev/vx/rdsk/adg/mirror3wayvol
    version 7 layout
    2097152 sectors, 1048576 blocks of size 1024, log size 16384 blocks
    largefiles supported
# mount -F vxfs /dev/vx/dsk/adg/mirror3wayvol /mirror3way
# vxassist make stripemirrorvol 1g layout=stripe-mirror init=active
# mkfs -F vxfs /dev/vx/rdsk/adg/stripemirrorvol
    version 7 layout
    2097152 sectors, 1048576 blocks of size 1024, log size 16384 blocks
    largefiles supported
# mount -F vxfs /dev/vx/dsk/adg/stripemirrorvol  /stripemirror
```

Next, we dump the **vxtunefs** output for each file system into a file in /var/tmp, and then **diff** it with respect to the plain concat volume. Finally, we run the result over **cat -n** in order to get line numbers for convenient referencing:

```
# for MOUNTPOINT in /concat /stripe5col /mirror3way /stripemirror; do
  vxtunefs $MOUNTPOINT >/var/tmp/$MOUNTPOINT.tuna
done
```

First, let's look at the differences between the concat volume and the five column stripe we just created:

```
# diff /var/tmp/concat.tuna /var/tmp/stripe5col.tuna | cat -n
     1  1,8c1,8
     2  < Filesystem i/o parameters for /concat
     3  < read_pref_io = 65536
     4  < read_nstream = 1
     5  < read_unit_io = 65536
     6  < write_pref_io = 65536
     7  < write_nstream = 1
     8  < write_unit_io = 65536
     9  < pref_strength = 10
    10  ---
    11  > Filesystem i/o parameters for /stripe5col
    12  > read_pref_io = 1048576
    13  > read_nstream = 5
    14  > read_unit_io = 1048576
    15  > write_pref_io = 1048576
    16  > write_nstream = 5
    17  > write_unit_io = 1048576
    18  > pref_strength = 20
    19  15c15
    20  < max_diskq = 1048576
    21  ---
    22  > max_diskq = 83886080
```

What we see is that several parameters have changed. In particular, the preferred read-I/O and write-I/O size (lines 3, 6, 12, and 15) have adapted to reflect the stripe unit size of the striped volume. In addition, the number of read and write I/O streams (lines 4, 7, 13, and 16) has increased from one (which it is for the concat volume) to five (for the five column stripe). This reflects the fact that VxFS sees many more disks in the underlying storage and assumes it can put an I/O on each of the disks in parallel. Remember this assumption for the later part of the chapter. It will be the basis for some optimization for SAN storage.

Apart from the I/O parallelity parameters, the size of the disk queue generated per file has increased enormously (from 1MB to 80 MB). This result is arrived at by granting sixteen I/Os of the preferred I/O size (**read_pref_io** or **write_pref_io**) to every write stream (**write_nstream**). The concat volume has a preferred I/O size of 65536 (64KB) and only one

write stream. sixteen times 64K is 1MB, so this is the maximum disk queue value for VxFS (the maximum number of bytes per file that reside in pages which are eligible for flushing to disk).

The five-way stripe, due to its large stripe unit size of 1 MB, gets the better end of it: five write streams (due to the five columns) multiplied by 1 MB multiplied by 16 yields the comparatively whopping 80 MB of queue size. Only after this size is exceeded does VxFS throttle write I/O to this file system; much later than in the case of the concat volume.

Now, let's check the differences between the concat volume and the three-way mirrored volume:

```
# diff /var/tmp/concat.tuna /var/tmp/mirror3way.tuna | cat -n
     1  1c1
     2  < Filesystem i/o parameters for /concat
     3  ---
     4  > Filesystem i/o parameters for /mirror3way
     5  3c3
     6  < read_nstream = 1
     7  ---
     8  > read_nstream = 3
```

As you see, not much has changed compared to the concat volume. Only the number of read streams has increased (lines 6 and 8) to reflect the number of plexes that can be read (three), so maximum read parallelization has increased. The stripe was better than that in this respect. It used five parallel read streams. So let's see what we get when we compare the concat volume with the stripe-mirror:

```
# diff /var/tmp/concat.tuna /var/tmp/stripemirror.tuna  | cat -n
     1  1c1
     2  < Filesystem i/o parameters for /concat
     3  ---
     4  > Filesystem i/o parameters for /stripemirror
     5  3c3
     6  < read_nstream = 1
     7  ---
     8  > read_nstream = 3
     9  6c6
    10  < write_nstream = 1
    11  ---
    12  > write_nstream = 3
    13  8c8
    14  < pref_strength = 10
    15  ---
    16  > pref_strength = 20
    17  15c15
    18  < max_diskq = 1048576
    19  ---
    20  > max_diskq = 3145728
```

Again, the number of read streams has increased (lines 8 and 12) because this volume has three plexes instead of one. But because we stuck to the default stripe unit size of 64 KB (we did not specify "**stwid=...**" this time) the maximum disk queue has not increased dramatically; it is merely three times the value of the concat, which comes from the fact that there are now three columns that can be written to independently.

VxFS Tunable Parameters and How to Set Them

So, looking at the automatically generated VxFS tunables you might get the impression that striping a volume is not too bad after all. On the other hand, this book has been telling you over and over that striping in current data center setups at least tends to be counterproductive for performance. Who is right? Well, VxFS would be right and striping would be good if we are using physical disks. But if we are using SAN storage then VxFS is not right because then the parameters that the VxFS-specific mount command calculates are based on virtual objects (LUNs) rather than physical objects (disks). Virtual objects do not have the same limits as physical objects have (they were, after all, created specifically to overcome the limitations of their physical counterparts).

Using vxtunefs to Optimize Your SAN Storage

So the way to get the best of both world, to combine the enhanced features of virtual disks aka LUNs with the best volume layout (concat) without limiting the file system to ridiculously low values of disk queuing of parallelity is to tune your file systems to your SAN storage instead of some assumed physical disks.

You can do so by creating a file named **/etc/vx/tunefstab** that contains the tunable parameters for every file system of your host system. This file contains all the deviations from the default that you wish to impose on your file systems. If, like most data centers today, you are using basically one kind of storage array, and especially of you are using only one kind of LUN, then you can tweak the tunefstab file pretty easily: create a volume that resembles, on the VxVM level, the specifications of your LUNs.

E.g. if your LUNs are 6-way striped with a 512 KB stripe unit size, and you are mirroring all volumes (two plexes per volume) using VxVM, then you could create a sample volume that exposes all the underlying physical features to VxFS: just create a volume with two plexes, each of which is a 6-way stripe with 512 KB stripe unit size. Then, put a VxFS on it, mount it and run vxtunefs on the mount point.

The **vxtunefs** command will diligently create the necessary defaults to make the best out of a two-way mirror with underlying 6-column stripes with 512 KB stripe unit size, and output them to your terminal. Your job now is to catch that output, put it into **/etc/vx/tunefstab** for each volume that uses this particular SAN storage type and re-mount the volumes so the new values take effect. You will have to tweak the output format quite a bit to make a valid **/etc/vx/tunefstab** entry out of the **vxtunefs** output, but we will show you how to do it shortly.

Organization of the /etc/vx/tunefstab File

The file **/etc/vx/tunefstab** contains tuning information for all file systems, so in cluster systems it is a very good idea to replicate the file across all nodes in order to make sure that the relevant tuning parameters are accessible wherever a service group my go online.

The file is organized line by line, with each line containing either nothing, a comment, or the description of a single file system.

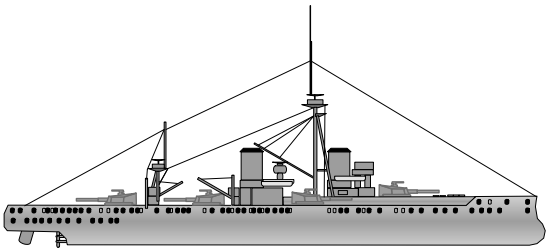
The following is a sample tunefstab for three of the four volumes we used. It also sets the system-wide default parameters for all **vxfs** file systems that have no specific entry in the tunefstab:

```
# cat /etc/vx/tunefstab
# Set some reasonable read/write defaults that match our SAN box
# These take precedence over the values derived from the volume
# layout which are found by mount_vxfs. Note that this does not
# seem to work any more as of Storage Foundation 5.0.
system_default  read_pref_io=1048576,write_pref_io=1048576,read_nstream=8

# Now fine-tune individual file systems
# This takes precedence over the values defined in system_default
/dev/vx/dsk/adg/stripemirror     max_diskq=83886080,write_nstream=8
/dev/vx/dsk/adg/stripe5col       max_diskq=3145728,write_pref_io=2097152

# The following entries will all be used, even if they are on
# different lines
/dev/vx/dsk/adg/concatvol        max_diskq=3145728,write_nstream=16
/dev/vx/dsk/adg/concatvol        read_nstream=16
```

The manual page for the **/etc/vx/tunefstab** file mentions an entry called **system_default**, which supposedly sets the values for all **vxfs** file systems. We have used this entry in the example above. But it seems like this entry does not work any more in Storage Foundation 5.0, as all tests performed on our hardware failed to produce any results using this setting.



The Full Battleship

13.2 Tools for Performance Tuning VxVM on SAN Storage

VxWork

A swiss company called **in&twork AG** has created a tool to ease administration and to improve performance on SAN storage by balancing the load as much as possible across all controllers in both the host and the storage array. It also recognizes and optimizes the paths to physical disks inside the array. Unless there are too many hosts attached to a single storage array this appears to be a very reasonably approach to optimization, because in many cases volumes under perform dramatically simply because of unfavorable mapping from volume blocks to hardware disk blocks. This problem can be fully remedied by using VxWork. In cases where there are too many hosts attached to a storage array it is not sure that there is a noticeable advantage, but it may be worth a try. Their web address is **`http://www.inwork.ch`**. Note that this is an independent third-party product and is not in any way related to Symantec.

Tracing and Benchmarking VxVM Volumes

In addition to the commercial tool mentioned above there are several tools delivered with Storage Foundation that can help you find bottlenecks and possibly remove them. These tools come with the support package (**`VRTSspt`**). Two of them – **`vxdmpadm`** and **`vxtrace`** – are installed in **`/usr/sbin`**, but the benchmark program – **`vxbench`** – is installed in **`/opt/VRTSspt/FS/VxBench/vxbench_$RELEASE`**, where **`$RELEASE`** corresponds to the release number of the Solaris operating system, e.g. **`/opt/VRTSspt/FS/VxBench/vxbench_10`**. So how do these tools work, and what are the relevant command usages for performance tuning?

vxdmpadm

The **`vxdmpadm`** command controls the dynamic multipathing layer of VxVM. It is often used for inquiring path status, and in preparation of scheduled maintenance on paths. For instance, if there was a firmware upgrade due on the storage array then this is done on

one storage array controller after the other, in order to keep the service online. In such a case you would first disable the corresponding path for the first controller, upgrade the controller, re-enable the path, then disable the second path and so on.

It is less widely known that **vxdmpadm** can also be used to find bottlenecks, especially in the form of overloaded controllers or LUNs. If you use the right parameters **vxdmpadm** will display the amount of usage per DMP path. Note that if you are using **mpxio** or another multipathing product that hides the individual paths from DMP's view, the sub-paths of that multipathing product cannot be shown; you will need to resort to the utilities that the vendor of that multipathing software delivers.

If you are using DMP then you can control gathering performance statistics and subsequently read out the statistics and display them by using simple **vxdmpadm** commands. The procedure – in short – is this:

```
# vxdmpadm iostat start # Automatically done at boot time
# vxdmpadm iostat show all      # Display all statistics
# vxdmpadm iostat reset # Set counters to zero
# vxdmpadm iostat show all interval=5 count=10   # more options
# vxdmpadm iostat stop  # Stop gathering statistics
```

It is good to know that gathering performance statistics is not CPU-intensive. You can leave it on without worrying about the overhead.

Here is a sample output of the command that shows the statistics:

```
# vxdmpadm iostat show all
                     cpu usage = 52748us    per cpu memory = 32768b
                       OPERATIONS              KBYTES           AVG TIME(ms)
PATHNAME             READS     WRITES      READS     WRITES     READS   WRITES
c1t1d1s2              2614        255        291         31 81.501718 42.064516
c2t17d1s2             2172       4983        271        587 82.704797 25.589438
c1t1d6s2              7109       2736        711        272 71.101266 18.393382
c2t17d6s2             5370       7162        663        810 70.126697 23.766667
c1t1d2s2              8452       2977       2257        235 19.428002 16.476596
c2t17d2s2             3824       5218        391       2443 62.506394 4.887843
c1t1d3s2              7767       7502        806        824 61.972705 21.253641
c2t17d3s2             6344       2308        765        249 58.785621 18.859438
c1t1d4s2              7116       7178        802        818 62.201995 21.117359
c2t17d4s2             6122       2028        765        253 59.546405 18.169960
c1t1d7s2              3990       4272       2149       3086 17.217310 3.563837
c1t1d9s2              2662        229        296         28 55.266892 43.071429
```

VXTRACE

A nice utility was developed for VxVM to follow I/Os from the Volume to the LUN and see what actually gets sent to the storage array. This tool is called vxtrace and will become your good friend especially when trying to find out exactly what kind of I/O pattern your application creates. This is often the entry point for optimization efforts. It goes without

saying that in order to optimize, you first need to know what to optimize for. Does the application create mostly read or mostly write I/O? Are the I/Os large or small, consecutive or random? Do they address the whole volume or are they mostly limited to hot spots? You can find all that out using vxtrace on a mounted file system.

The following are a few walkthroughs of **vxtrace** on a volume while using vxbench (see below) to put database-like I/O onto a file in the file system residing on that volume.

```
# vxtrace -o dev,disk -g adg mirrorvol
1091 START read vdev mirrorvol block 3904352 len 16 concurrency 1 pid 24980
1092 START read disk c1t1d7s2 op 1091 block 3970144 len 16
1092 END read disk c1t1d7s2 op 1091 block 3970144 len 16 time 0
1091 END read vdev mirrorvol op 1091 block 3904352 len 16 time 1
1093 START read vdev mirrorvol block 34640160 len 16 concurrency 1 pid 24980
1094 START read disk c1t1d7s2 op 1093 block 34705952 len 16
1094 END read disk c1t1d7s2 op 1093 block 34705952 len 16 time 0
1093 END read vdev mirrorvol op 1093 block 34640160 len 16 time 0
1095 START read vdev mirrorvol block 23138688 len 16 concurrency 1 pid 24980
1096 START read disk c1t1d7s2 op 1095 block 23204480 len 16
1096 END read disk c1t1d7s2 op 1095 block 23204480 len 16 time 0
1095 END read vdev mirrorvol op 1095 block 23138688 len 16 time 1
1097 START read vdev mirrorvol block 61058832 len 16 concurrency 1 pid 24980
1098 START read disk c1t1d7s2 op 1097 block 61124624 len 16
1098 END read disk c1t1d7s2 op 1097 block 61124624 len 16 time 0
1097 END read vdev mirrorvol op 1097 block 61058832 len 16 time 0
1099 START read vdev mirrorvol block 11335712 len 16 concurrency 1 pid 24980
1100 START read disk c1t1d2s2 op 1099 block 44955936 len 16
1100 END read disk c1t1d2s2 op 1099 block 44955936 len 16 time 1
1099 END read vdev mirrorvol op 1099 block 11335712 len 16 time 1
1101 START read vdev mirrorvol block 71050496 len 16 concurrency 1 pid 24980
1102 START read disk c1t1d8s2 op 1101 block 25451776 len 16
1102 END read disk c1t1d8s2 op 1101 block 25451776 len 16 time 1
1101 END read vdev mirrorvol op 1101 block 71050496 len 16 time 1
```

And so on; you get the point: Every I/O is tagged (in the left most column) by an identifier that makes it easy to correlate the I/O START to the I/O END. Each I/O to or from a "**vdev**" (virtual device) is converted to an I/O to or from a "**disk**" (physical device, or what the computer thinks is a physical device). The length field always says "**len 16**", which means all I/Os in the (admittedly tiny) time frame we observed were 8KB in size (16 blocks). As you can see from the block numbers they are not sequential at all. In that particular case, nothing can be gained from tuning the I/O size to some large value (all I/Os are small anyway) or from making the files more contiguous (the I/Os access the file in a random fashion anyway, so that would not help).

Here is another example from a mirror-stripe that shows how striping can have a massively negative effect on volume I/O behavior if it is not used wisely:

```
# vxtrace -o dev,disk -g adg mirror-stripevol
1147 START read vdev mirror-stripevol block 43008 len 2048 concurrency 1 pid
25646
```

```
1148 START read disk c1t1d4s2 op 1147 block 89552896 len 64
1149 START read disk c1t1d5s2 op 1147 block 89552896 len 64
1150 START read disk c1t1d6s2 op 1147 block 89552896 len 64
1151 START read disk c1t1d2s2 op 1147 block 79293376 len 64
1152 START read disk c1t1d3s2 op 1147 block 89552960 len 64
1153 START read disk c1t1d4s2 op 1147 block 89552960 len 64
1154 START read disk c1t1d5s2 op 1147 block 89552960 len 64
1155 START read disk c1t1d6s2 op 1147 block 89552960 len 64
1156 START read disk c1t1d2s2 op 1147 block 79293440 len 64
1157 START read disk c1t1d3s2 op 1147 block 89553024 len 64
1158 START read disk c1t1d4s2 op 1147 block 89553024 len 64
1159 START read disk c1t1d5s2 op 1147 block 89553024 len 64
1160 START read disk c1t1d6s2 op 1147 block 89553024 len 64
1148 END read disk c1t1d4s2 op 1147 block 89552896 len 64 time 0
1161 START read disk c1t1d2s2 op 1147 block 79293504 len 64
1149 END read disk c1t1d5s2 op 1147 block 89552896 len 64 time 0
1162 START read disk c1t1d3s2 op 1147 block 89553088 len 64
1163 START read disk c1t1d4s2 op 1147 block 89553088 len 64
1164 START read disk c1t1d5s2 op 1147 block 89553088 len 64
1165 START read disk c1t1d6s2 op 1147 block 89553088 len 64
1166 START read disk c1t1d2s2 op 1147 block 79293568 len 64
1167 START read disk c1t1d3s2 op 1147 block 89553152 len 64
1150 END read disk c1t1d6s2 op 1147 block 89552896 len 64 time 0
1168 START read disk c1t1d4s2 op 1147 block 89553152 len 64
1169 START read disk c1t1d5s2 op 1147 block 89553152 len 64
1151 END read disk c1t1d2s2 op 1147 block 79293376 len 64 time 0
1170 START read disk c1t1d6s2 op 1147 block 89553152 len 64
1171 START read disk c1t1d2s2 op 1147 block 79293632 len 64
1172 START read disk c1t1d3s2 op 1147 block 89553216 len 64
1173 START read disk c1t1d4s2 op 1147 block 89553216 len 64
1174 START read disk c1t1d5s2 op 1147 block 89553216 len 64
1175 START read disk c1t1d6s2 op 1147 block 89553216 len 64
1152 END read disk c1t1d3s2 op 1147 block 89552960 len 64 time 0
1176 START read disk c1t1d2s2 op 1147 block 79293696 len 64
1177 START read disk c1t1d3s2 op 1147 block 89553280 len 64
1178 START read disk c1t1d4s2 op 1147 block 89553280 len 64
1179 START read disk c1t1d5s2 op 1147 block 89553280 len 64
1157 END read disk c1t1d3s2 op 1147 block 89553024 len 64 time 0
1156 END read disk c1t1d2s2 op 1147 block 79293440 len 64 time 0
1159 END read disk c1t1d5s2 op 1147 block 89553024 len 64 time 0
1158 END read disk c1t1d4s2 op 1147 block 89553024 len 64 time 0
1154 END read disk c1t1d5s2 op 1147 block 89552960 len 64 time 0
1153 END read disk c1t1d4s2 op 1147 block 89552960 len 64 time 0
1160 END read disk c1t1d6s2 op 1147 block 89553024 len 64 time 0
1155 END read disk c1t1d6s2 op 1147 block 89552960 len 64 time 0
1166 END read disk c1t1d2s2 op 1147 block 79293568 len 64 time 0
1167 END read disk c1t1d3s2 op 1147 block 89553152 len 64 time 0
```

```
1168 END read disk c1t1d4s2 op 1147 block 89553152 len 64 time 0
1170 END read disk c1t1d6s2 op 1147 block 89553152 len 64 time 0
1162 END read disk c1t1d3s2 op 1147 block 89553088 len 64 time 0
1161 END read disk c1t1d2s2 op 1147 block 79293504 len 64 time 0
1169 END read disk c1t1d5s2 op 1147 block 89553152 len 64 time 0
1174 END read disk c1t1d5s2 op 1147 block 89553216 len 64 time 0
1163 END read disk c1t1d4s2 op 1147 block 89553088 len 64 time 0
1175 END read disk c1t1d6s2 op 1147 block 89553216 len 64 time 0
1176 END read disk c1t1d2s2 op 1147 block 79293696 len 64 time 0
1177 END read disk c1t1d3s2 op 1147 block 89553280 len 64 time 0
1178 END read disk c1t1d4s2 op 1147 block 89553280 len 64 time 0
1179 END read disk c1t1d5s2 op 1147 block 89553280 len 64 time 0
1165 END read disk c1t1d6s2 op 1147 block 89553088 len 64 time 1
1172 END read disk c1t1d3s2 op 1147 block 89553216 len 64 time 6
1164 END read disk c1t1d5s2 op 1147 block 89553088 len 64 time 6
1171 END read disk c1t1d2s2 op 1147 block 79293632 len 64 time 7
1173 END read disk c1t1d4s2 op 1147 block 89553216 len 64 time 7
1147 END read vdev mirror-stripevol op 1147 block 43008 len 2048 time 7
```

The first and the last line of output are the START and END lines for the volume I/O number 1147, which is 1MB (2048 blocks). As you can see this single I/O is split up into lots of 32K (64 blocks) I/Os because the underlying volume is striped with a stripe unit size which is much smaller than the application's I/O size. It is obvious that this has a negative effect on the overall performance of the storage array. The storage array has to handle many more individual I/Os, cannot efficiently use its read-ahead cache etc. There might be an advantage if the storage array was dedicated to our host, because we are reading from several spindles in parallel. But first of all, storage arrays nowadays are usually shared by a massive number of hosts. And second, it is not at all certain that there actually would be any noticeable performance increase due to striping due to the increased read latency and the high speed of sequential access. Keep all the relevant parameters in mind when optimizing your storage!

That said, a realistic cycle of performance tuning a live application volume in a data center is the following:

1. Make sure any reasonable optimization can be done at all. This is usually **not** the case if several hundred servers are attached to the same storage array, as any optimization that works to the benefit of your server will work against the other servers.
2. Thoroughly optimize the file system first using (possibly several runs of) **`fsadm -de $MOUNTPOINT`**. Doing the following steps on a file system that is not optimized leads to bogus results, as I/Os that would normally be large and sequential will get split into smaller random I/Os.
3. Run **`vxprint`** to find the current volume layout
4. Run a cycle of **`vxdmpadm iostat reset / vxdmpadm iostat show all`** to find any obviously overloaded controllers or LUNs
5. Run the command **`vxtrace -o dev,disk -g $DG $VOLNAME`** on the volume or **`vxtrace -o dev,disk -g $DG $DISKNAME`** on an overloaded target several times during various operating cycles of the application, like during normal operation, backup,

database export, etc.

6. Identify the I/O pattern of the application.
7. Relayout the volume to match the I/O pattern of the application and ideally the I/O requirements of the storage array. For this, you need to know the internal organization of your storage array's LUNs. For example, a LUN may consist of a slice out of a RAID-5 group. The RAID-5 group may consist of eight disks and use a stripe size of 64 KB or 512 KB.

If performance is still lacking, there is probably not much you can do, at least from a Volume Manager perspective. Most likely your SAN or storage array is overloaded (this is all too common nowadays) or the data access code in the application is not optimal (also very common).

So we are not expecting you to experience any multi-digit performance gains by optimizing VxVM volumes and VxFS file systems. It happens, but is not very common. You still need to do it, though, because in many cases the persons responsible for the SAN infrastructure, the SAN and volume storage, the database, and the application that uses the database will quarrel about whose fault the perceived performance problem is. If you have optimized your volumes and file systems (and maybe gained 5 percent), not only can you seriously claim that from your side the optimum is reached, but you might even claim that you have been keeping the volumes in good shape all the time: only little could be gained. Now the issue lies with the other departments and you are outside of the firing line.

VXBENCH

If you have analyzed the application's I/O pattern then you can simulate this I/O pattern in a reliable, repeatable way using **vxbench**. This little program resides – as mentioned above – inside **/opt/VRTSspt/FS/VxBench** and there are several versions of it: one for each version of the Solaris OS. The one for Solaris 10, for instance, has the path **/opt/VRTSspt/FS/VxBench/vxbench_10**. It takes a number of parameters, most notably a workload parameter **-w**, which determines the type of I/O: read or write, sequential or random, memory-mapped or normal, asynchronous or synchronous etc. The parameter **-i** specifies the sub-options to the workload, like the size of each I/O, how many threads to use, how many I/Os to make etc. There is a great lot of options and it is probably best if you call **vxbench -h** yourself to get some help. We will tell you just what you need to know in addition to what you can see from the help text because there are some things in **vxbench** which are not obvious.

1. The most important thing is that vxbench does not create files. Files must exist or the program fails. But if you use existing files be careful not to specify any workload that does write I/O or you will lose data! If you want to check just the volume speed you can specify the raw device to skip the file system code path, but again, be careful if you specify writing workloads, as this may corrupt an existing file system on that device.
2. If your files are small then **vxbench** will likely read beyond the end of the file and terminate with an error. It sounds like a stupid oversight, but that is the way it has been for years , so there is probably a good (if not particularly obvious) reason to it. You can (and sometimes must) limit the maximum offset in a file that is accessed by vxbench to something significantly smaller than the file size. In particular, if you

are doing hundreds of one MB I/Os in a sequential workload, then it is better to stay hundreds of MBs away from the end of the file.

3. To simulate database I/O, you need to specify a synchronous open-type. That causes **vxbench** to open the file in synchronous mode, like databases usually do. The flag is **vxbench -o dsync** or **vxbench -o sync**. You can somewhat emulate the I/O behavior of Veritas Storage Foundation Database Edition for Oracle on database files by additionally mounting the volume in the following way:

```
# mount -F vxfs -o convosync=direct,mincache=direct,nodatainlog,delaylog \
        /dev/vx/dsk/$DG/$VOLNAME /$MOUNTPOINT
```

This will convert all synchronous I/O to direct I/O, which is unbuffered and therefore does not taint the operating system's buffer cache. It will also turn down the use of the file system's intent log in favor of the database's own transaction log.

One thing that should be obvious but may need to be mentioned: In order to exclude side effects due to caching on the side of the operating system, you must run all benchmarks on a freshly mounted file system (unless, of course, you are benchmarking a raw device). Unmounting the file system every time and remounting it before every test can be a tedious task, so we suggest a smarter way: Before you run the benchmark, change your working directory to the mount point that you want to test. If your current working directory is inside the mount point, then that file system cannot be unmounted. But the system does not know that. So if you now enter the umount command, the system will flush all the buffers that relate to the file system you are trying to unmount. Then, after all the buffers have been flushed and invalidated, the umount system call will fail. You are left with a file system that has nothing buffered in the system's cache without having to actually remount it.

And if you redirect **stderr** to **/dev/null**, you don't even get the error message. Here's an example of the command:

```
# cd $MOUNTPOINT# Make umount fail if tried on $MOUNTPOINT
# umount $MOUNTPOINT 2>/dev/null # Flush all buffers: no caching side effects
# vxbench -w rand_read -i iocount=50,iosize=8,maxfilesize=35g /stripe/DATAFILE
```

13.3 Performance Tuning

13.3.1 Overview and Disclaimer

One of the most prominent marketing features for all kinds of computer equipment and software is a vast array of mutually interdependent figures collectively called "performance". This ominous feature is usually measured by some software suite in what is called a "benchmark". Both words, performance as well as benchmark, are not well defined at all, mostly because performance and everything related to it is such a multi-faceted issue. This chapter will highlight performance from various perspectives. It will help you define what kind of performance you actually need, how to measure it and possible ways to increase it or, more likely, to prevent doing something that stands in the way of good performance. Be warned that it is not reliable possible to guide anyone towards achieving high performance in their particular setup. Too many variables influence each other. But anybody looking for ways to optimize their system's performance will find lots of useful information in this chapter, and with a little bit of luck, it may hint you towards the bottleneck that has so far throttled your system's throughput.

13.3.2 Identifying Performance and Performance Requirements

Since tuning reminds many of cars and motorbikes let's imagine that we are not running a data center but a car racing team. We do so not for the sake of a demoralizing the reader (running race cars is more fun than running computers) but also, as you will see, because it turns out the two fields are very comparable in surprisingly many fields.

In the following pages we will help you identify your exact needs. This question is harder to answer than you may think. Once you have found out what the goal is, you can use your technical, managerial, and social skills reach that goal. So here's the analogy:

A car racing team and a data center operation are similar in several ways because both

1. Use leading edge hardware and personnel.

   It is very hard to win a car race with an old vehicle and an untrained or unmotivated team, just as it is hard to get decent and reliable performance from old computer systems and an untrained or unmotivated team.

2. Employ extremely complex setups with multiple interdependencies. Only if all

of the parameters are set correctly will the computer – or the car – be competitive. Unfortunately, in race car setup as well as in computer setup, it is not at all easy to find the reason for poor performance, as there are so many possibilities and they are so intimately intertwined. **There is never a magic bullet to make any complex setup fast! But there are dozens of areas that can make it slow!**

Some parameters that are interdependent in race cars: Tire pressure, tire rubber mixture and construction, spring rate, damper rate, strut, toe-in, down force, weight, grip, engine power, track layout, tarmac type, temperature, average cornering speed etc.

Some parameters that are interdependent in data center applications: Memory, CPU cores, degree of parallelization, latency, queue lengths, IOPS, I/O sizes, throughput, number of users, response times, etc.

3. Must cope with certain budget restrictions, which may be alleviated by over performing

A race team gets money from the sponsors and from winning races. A successful team gets more prize money and attracts more sponsors and high-end staff. A data center has a limited budget, but if the operations run particularly well, the company can expand and pass more money towards the data center. Unfortunately, in the current global economic climate this often is not the case. Many managers take the short-sighted approach of demanding ever increasing shareholder value, thereby disgusting their staff and leading to eventual meltdown. We hope and trust it is a matter of time until this is fixed.

4. Are multi-layered, with each layer not necessarily pursuing the same goal

While the race car driver may just want to arrive first and have fun doing so, the team management usually pursues longer-term goals: to ultimately win the championship in order to attract more sponsors; the sponsors, in turn, would prefer the team to display excellent craftsmanship and fairness in order to better convey their message. And the team owner may not be looking at just this year's championship, but may have much longer-term goals. In a data center, the typical administrator wants to shut up the machine as much as possible so he can do more interesting things or handle more machines. The application owners want their application to run at optimum speed and do not care about the others. The SAN admin wants everybody to use the same kind and size of LUN to reduce the complexity. The network and security people would

like to install fingerprint and retina scanners in every network node and forward packets only after thorough inspection concerning all conceivable security holes discovered since 1975. The data center manager wants the whole thing as cheaply as possible and often couldn't care less about current technological and physical limits, especially when these get in the way of reducing cost.

5. Are confronted with sudden unexpected developments so they must be able to react to quickly and with great flexibility.

   A race can be turned upside-down by a yellow-phase, a sudden onset of rain, a technical defect, driver errors and so on. A data center must cope with unexpectedly high user demand, technical defects and outages, administrator errors etc.

Even with all the similarities, there is also one big difference between a race car team and data center staff: races are only held at certain times, while data center applications are usually 24/365 due to the ubiquity of internet applications.

So here is the question: What is performance? What kind of performance do we wish to attain? This is usually connected to another question: Who are we optimizing for? The administrator? The manager? The SAN group? The network and security group? (Hint: the application owner is a good, but often forgotten, candidate in the real world, although the other candidates are just as valid). This is the first question that we need to clarify, lest we optimize for something impressive yet irrelevant. It sounds trivial, but in fact it is very often overlooked.

So think about the following things that you might be trying to reach. Each of them is a completely different, yet valid, aspect of performance:

Shortest time to execute a given task

That's obvious. It's like winning the race. But it may be prohibitively costly or very hard to administer.

Lightest system load to execute a given task

Sounds good, too. But it does not help if the machine is idling anyway, or it may cause too long a response time.

Largest number of users served

This goes into the cost-saving direction, with a bittersweet note that response time may be underrepresented

Lowest overall cost solution

The full emphasis on cost-saving

Shortest response time to the user

What the users want (but they don't want to pay a lot for it).

Now think about the following factors that might contribute to improving overall "performance":

CPU clock

Costs money

Number of CPU cores

Costs money, especially license fees

Parallelity of the problem set

Can't help it much...

Parallelity of the implementation (software)

Requires really smart developers

I/O operations per second of the machine

Costs money

Size of the I/Os

Requires at least some degree of data locality in the application, ideally sequential access

Latency of I/O

Requires money or smart solution to reduce

On a more hardware-oriented technical level, the following aspects:

Protocols used

Can't usually help it much

Long-range latencies

Requires smart solution or additional bunker site close by that holds just the logs and replicates asynchronously from there to the remote site

Error rates and recovery mechanisms

Can't usually help it much; they are defined by the transport medium's physical properties and the protocol used...

Timeout values

Can tweak them somewhat, but usually not much

And on a more global level, these:

Number of systems connected to a storage array

Costs money to reduce

Frame rate for SAN switches

Costs money to increase

Distance of the remote disaster recovery site

Can't usually help it much...

As you see, there are many parameters and optimization goals possible, so there can not be one performance tuning guide that does it all right. One could argue that in the context of this book at least, it should be clear that what is needed is a performance tuning guide to accelerate I/Os, right? It turns out that even this apparently simple goal is actually manifold: The first problem that springs to mind is the use of shared SAN resources. If user A makes his volumes run faster, then user B typically gets less performance. Storage arrays are usually saturated with I/O requests, and the more some egotistic application owner or admin optimizes his "own" LUNs, the more I/Os per second are loaded onto the shared disk spindles. But that is only part of the problem: User A will also tend to increase the volume's latency, because in most cases administrators will try to increase the striping factor in order to gain performance. Increasing the striping factor distributes the load across more LUNs, but also increases read latency and CPU and channel overhead. It will also degrade overall performance of the storage array instead of improving it because disk heads have to move more (because load is distributed across more targets) and the storage array's read-ahead cache is not used as efficiently. So in short: one cannot even say that "speeding up the volumes" is a good thing.

Alternatively, you may want to think about using storage checkpoints to speed up your backups. Storage checkpoints have been extensively covered in the chapter about point-in-time copies. They can speed up backups by allowing you, in combination with Oracle's RMAN (Recovery MANager) utility, to feed just those blocks into RMAN that have been updated since the last backup. But this feature is not part of the base license, so it may cost a lot of money to do implement a solution based on storage checkpoints. Finally, even if you manage to increase I/O throughput without degrading everybody else's you may not have an I/O-bound problem, but a memory-bound problem. In that case, it would be so much more worthwhile to just add memory to the system.

But the real problem comes when you look at how complex the I/O subsystem alone has become. It is next to impossible to know all the limiting factors in the chain from the application down to the disk hardware, but here's at least a list of some of the more important ones:

- OS queue length and buffer sizes
- OS I/O parallelity
- HBA frame buffers
- Buffer credits HBA <–> intermediate nodes <–> storage array
- Fabric speed and error rates
- Storage array (SA) front end controller HBA buffers
- SA back end controller queue length
- SA disk queue length and on-disk controller hardware
- SA disk rotational speed
- SA disk type (S-ATA tends to be slow in mixed read-write operation)

The bottom line is that optimization is best done as high up as possible: the people who write the software have control over the amount, frequency, and locality of data they read. If they fail to deliver a smart approach to a problem, the next layer is usually the database staff. More often than not, database staff think differently from systems staff, so it may be hard to talk to them. (Systems people – at least the good ones – tend to rely on strict logic and prefer to understand exactly what is going on. Database people are usually very happy that their database gives them some kind of table or view for everything they want to know, and they often do not question the origin of the values in these tables, not their exact meaning). Operating systems and their staff is stuck at the low end of the food chain, and is often accused, but rarely guilty, of delivering under performing I/O systems. So what we can at least do here is provide you with a set of argumentative guidelines so that you can redirect the tuning efforts to where they belong: higher up the food chain, towards the database and application.

13.3.3 Comparative Benchmarks of Various Volume Layouts

Just to give you a rough idea of how much or how little performance can be gained by varying the volume layout versus varying the access pattern we have done a series of benchmarks using **vxbench** on the test bed graciously provided by the fine people of Sun microsystems in Langen, Germany. The machine was a 16-core SPARC LDOM which was redundantly connected to a Hitachi 9980 storage array. The benchmarks were done on varying volumes, each created to the same size (100g) and with a **vxfs** file system on it. Only default parameters were used, the number of stripe columns (**ncol**) is 5. The volume layout is shown in the left most column. The kind of access is given in the next column: read and write (which are sequential), and rand_read and rand_write (random). The next column is the size of the individual I/Os in KBytes. It is not the blocksize of the file system or volume! The columns labeled **PARLL** contains the parallelity, i.e. the number of threads concurrently accessing the volume. The speed is output in the next column, and is given in MB/sec. The columns for **time**, **sys**, and **user** stand are given in seconds and are equivalent to the values output by the UNIX **time** command.

Two runs were executed, with different sort orders. This first run is ordered by layout, then I/O size, then access type. The second one is ordered by layout, then access type, then I/O size.

We urge you to look at how excruciatingly low the throughput in small random reads is. It is often below one MB/sec. Don't forget that this is the way that most data base accesses are one! Retrieving data from any table that is significantly larger than the physical machine memory almost always requires at least one, but usually several, random reads (the database needs to read several nodes of the index tables first before it knows which block or blocks the data resides on).

Concat

#	Layout	Access	BKSZ	PARLL	Speed	Time	Sys	User
	concat	write	8	32	23.91	0.082	0.02	0.00

concat	rand_write	8	32	17.86	0.109	0.02	0.00
concat	read	8	32	25.94	0.075	0.02	0.00
concat	rand_read	8	32	1.58	1.240	0.02	0.00
concat	write	64	32	96.16	0.162	0.02	0.00
concat	rand_write	64	32	78.81	0.198	0.02	0.00
concat	read	64	32	101.05	0.155	0.02	0.00
concat	rand_read	64	32	8.88	1.760	0.02	0.00
concat	write	512	32	142.06	0.880	0.02	0.00
concat	rand_write	512	32	138.93	0.900	0.02	0.00
concat	read	512	32	139.06	0.899	0.02	0.00
concat	rand_read	512	32	23.22	5.384	0.02	0.00
concat	write	1024	32	147.19	1.698	0.03	0.00
concat	rand_write	1024	32	146.47	1.707	0.03	0.00
concat	read	1024	32	140.43	1.780	0.02	0.00
concat	rand_read	1024	32	26.90	9.294	0.03	0.00

Stripe

#	Layout	Access	BKSZ	PARLL	Speed	Time	Sys	User
	stripe	write	8	32	23.19	0.084	0.02	0.00
	stripe	rand_write	8	32	17.56	0.111	0.02	0.00
	stripe	read	8	32	26.46	0.074	0.02	0.00
	stripe	rand_read	8	32	1.08	1.817	0.02	0.00
	stripe	write	64	32	95.42	0.164	0.02	0.00
	stripe	rand_write	64	32	80.59	0.194	0.02	0.00
	stripe	read	64	32	95.01	0.164	0.02	0.00
	stripe	rand_read	64	32	7.11	2.197	0.02	0.00
	stripe	write	512	32	198.63	0.629	0.09	0.00
	stripe	rand_write	512	32	193.25	0.647	0.08	0.00
	stripe	read	512	32	229.04	0.546	0.09	0.00
	stripe	rand_read	512	32	28.22	4.430	0.09	0.00
	stripe	write	1024	32	220.81	1.132	0.16	0.00
	stripe	rand_write	1024	32	207.11	1.207	0.17	0.00
	stripe	read	1024	32	243.37	1.027	0.15	0.00
	stripe	rand_read	1024	32	29.16	8.574	0.14	0.00

Mirror

#	Layout	Access	BKSZ	PARLL	Speed	Time	Sys	User
	mirror	write	8	32	21.58	0.091	0.03	0.00
	mirror	rand_write	8	32	14.76	0.132	0.04	0.00
	mirror	read	8	32	24.61	0.079	0.02	0.00
	mirror	rand_read	8	32	1.09	1.792	0.02	0.00
	mirror	write	64	32	83.64	0.187	0.04	0.00
	mirror	rand_write	64	32	72.51	0.215	0.04	0.00
	mirror	read	64	32	98.67	0.158	0.02	0.00
	mirror	rand_read	64	32	8.28	1.888	0.02	0.00

```
        mirror      write       512       32    138.48   0.903    0.04    0.00
        mirror rand_write       512       32    133.18   0.939    0.05    0.00
        mirror       read       512       32    138.95   0.900    0.02    0.00
        mirror  rand_read       512       32     20.69   6.042    0.03    0.00
        mirror      write      1024       32    144.94   1.725    0.04    0.00
        mirror rand_write      1024       32    142.19   1.758    0.05    0.00
        mirror       read      1024       32    143.12   1.747    0.03    0.00
        mirror  rand_read      1024       32     27.01   9.254    0.03    0.00
```

Stripe-mirror

```
#         Layout      Access      BKSZ     PARLL      Speed     Time      Sys     User
  stripe-mirror      write         8        32      16.68   0.117    0.04    0.00
  stripe-mirror rand_write         8        32      10.69   0.183    0.04    0.00
  stripe-mirror       read         8        32      22.63   0.086    0.02    0.00
  stripe-mirror  rand_read         8        32       1.03   1.900    0.02    0.00
  stripe-mirror      write        64        32      57.56   0.271    0.04    0.00
  stripe-mirror rand_write        64        32      72.70   0.215    0.04    0.00
  stripe-mirror       read        64        32      94.23   0.166    0.02    0.00
  stripe-mirror  rand_read        64        32       7.54   2.071    0.02    0.00
  stripe-mirror      write       512        32      95.19   1.313    0.17    0.00
  stripe-mirror rand_write       512        32      92.51   1.351    0.18    0.00
  stripe-mirror       read       512        32     221.67   0.564    0.09    0.00
  stripe-mirror  rand_read       512        32      26.50   4.717    0.10    0.00
  stripe-mirror      write      1024        32     111.04   2.251    0.34    0.00
  stripe-mirror rand_write      1024        32     112.10   2.230    0.34    0.00
  stripe-mirror       read      1024        32     244.69   1.022    0.15    0.00
  stripe-mirror  rand_read      1024        32      26.73   9.354    0.18    0.00
```

This is the output of the second run, this time ordered by access type rather than I/O size to make it more easy to compare the influence of the relative I/O sizes.

Concat

```
#         Layout      Access      BKSZ     PARLL      Speed     Time      Sys     User
          concat      write         8        32      24.68   0.079    0.02    0.00
          concat      write        64        32      98.58   0.159    0.02    0.00
          concat      write       512        32     144.78   0.863    0.02    0.00
          concat      write      1024        32     146.85   1.702    0.03    0.00
          concat rand_write         8        32      17.88   0.109    0.02    0.00
          concat rand_write        64        32      79.23   0.197    0.02    0.00
          concat rand_write       512        32     137.94   0.906    0.03    0.00
          concat rand_write      1024        32     145.59   1.717    0.03    0.00
          concat       read         8        32      25.89   0.075    0.02    0.00
          concat       read        64        32      93.67   0.167    0.02    0.00
          concat       read       512        32     135.17   0.925    0.02    0.00
          concat       read      1024        32     140.03   1.785    0.02    0.00
```

concat	rand_read	8	32	0.44	4.467	0.02	0.00
concat	rand_read	64	32	2.79	5.595	0.02	0.00
concat	rand_read	512	32	14.34	8.714	0.02	0.00
concat	rand_read	1024	32	34.88	7.168	0.03	0.00

Stripe

#	Layout	Access	BKSZ	PARLL	Speed	Time	Sys	User
	stripe	write	8	32	22.66	0.086	0.02	0.00
	stripe	write	64	32	94.09	0.166	0.02	0.00
	stripe	write	512	32	209.06	0.598	0.08	0.00
	stripe	write	1024	32	221.14	1.131	0.15	0.00
	stripe	rand_write	8	32	17.71	0.110	0.02	0.00
	stripe	rand_write	64	32	81.39	0.192	0.02	0.00
	stripe	rand_write	512	32	189.98	0.658	0.09	0.00
	stripe	rand_write	1024	32	202.33	1.236	0.17	0.00
	stripe	read	8	32	26.00	0.075	0.02	0.00
	stripe	read	64	32	97.31	0.161	0.01	0.00
	stripe	read	512	32	223.65	0.559	0.08	0.00
	stripe	read	1024	32	234.69	1.065	0.16	0.00
	stripe	rand_read	8	32	0.62	3.147	0.02	0.00
	stripe	rand_read	64	32	4.51	3.465	0.02	0.00
	stripe	rand_read	512	32	23.95	5.220	0.09	0.00
	stripe	rand_read	1024	32	44.35	5.637	0.16	0.00

Mirror

#	Layout	Access	BKSZ	PARLL	Speed	Time	Sys	User
	mirror	write	8	32	10.92	0.179	0.04	0.00
	mirror	write	64	32	79.78	0.196	0.03	0.00
	mirror	write	512	32	130.31	0.959	0.04	0.00
	mirror	write	1024	32	138.40	1.806	0.05	0.00
	mirror	rand_write	8	32	14.81	0.132	0.04	0.00
	mirror	rand_write	64	32	60.45	0.258	0.04	0.00
	mirror	rand_write	512	32	132.12	0.946	0.05	0.00
	mirror	rand_write	1024	32	141.07	1.772	0.05	0.00
	mirror	read	8	32	24.12	0.081	0.02	0.00
	mirror	read	64	32	91.07	0.172	0.02	0.00
	mirror	read	512	32	136.33	0.917	0.02	0.00
	mirror	read	1024	32	140.23	1.783	0.03	0.00
	mirror	rand_read	8	32	0.49	3.976	0.02	0.00
	mirror	rand_read	64	32	4.27	3.660	0.02	0.00
	mirror	rand_read	512	32	20.06	6.231	0.03	0.00
	mirror	rand_read	1024	32	29.22	8.556	0.03	0.00

STRIPE-MIRROR

# Layout	Access	BKSZ	PARLL	Speed	Time	Sys	User
stripe-mirror	write	8	32	16.07	0.122	0.04	0.00
stripe-mirror	write	64	32	78.11	0.200	0.04	0.00
stripe-mirror	write	512	32	93.56	1.336	0.17	0.00
stripe-mirror	write	1024	32	112.78	2.217	0.33	0.00
stripe-mirror	rand_write	8	32	8.86	0.220	0.04	0.00
stripe-mirror	rand_write	64	32	68.97	0.227	0.04	0.00
stripe-mirror	rand_write	512	32	97.46	1.283	0.18	0.00
stripe-mirror	rand_write	1024	32	102.66	2.435	0.34	0.00
stripe-mirror	read	8	32	22.21	0.088	0.02	0.00
stripe-mirror	read	64	32	90.14	0.173	0.02	0.00
stripe-mirror	read	512	32	207.29	0.603	0.10	0.00
stripe-mirror	read	1024	32	203.92	1.226	0.18	0.00
stripe-mirror	rand_read	8	32	0.89	2.200	0.03	0.00
stripe-mirror	rand_read	64	32	4.79	3.263	0.02	0.00
stripe-mirror	rand_read	512	32	20.92	5.975	0.10	0.00
stripe-mirror	rand_read	1024	32	31.98	7.818	0.18	0.00

13.3.4 Summary

We hope to have given a reasonable overview over the various tuning possibilities that come with Storage Foundation. We have to admit that there were times when it was much more fun to optimize volumes: Raw disks have such a charming amount of intricacies: you could run into any limit: queue size on disk and in the controller, cache size on disk and in the controller and in the OS, cache entry size on disk versus I/O size on the SCSI bus, hot spots and so on. Tuning a data center machine was like the fine art of preparing a race car: If the setup was completely optimal you had a chance of winning the race. But if any of a (large) number of parameters was wrong – let alone several – there was no way you could have gotten decent performance.

With SAN storage that fine art was lost. Everyone just allocated storage from the box, and the box did all the thinking. Physics suddenly seemed irrelevant.

But now the discrepancy between disk head speed, disk transfer rate, and disk size has grown so much out of proportion (and is still continuing to do so), that physics has come back big time. So let's all be a little reasonable and keep in mind that no storage array can speed up random reads as long as it uses rotating disks for backing store.

The number of disks times the number of transactions per second is the upper limit to all I/O activity on your storage array, and that number hasn't kept up remotely with Moore's law. In addition, we are probably not the only ones using the storage array, so if we optimize the volumes to maximize our own performance, it is very likely that performance for everybody else is deteriorating. The others will then try to optimize their volumes to gain performance, and eventually everybody's performance becomes lousy.

If you have performance problems, it is most likely that you are simply expecting too much from your storage array. To modify an old proverb: "It's the disk drives, stupid!"