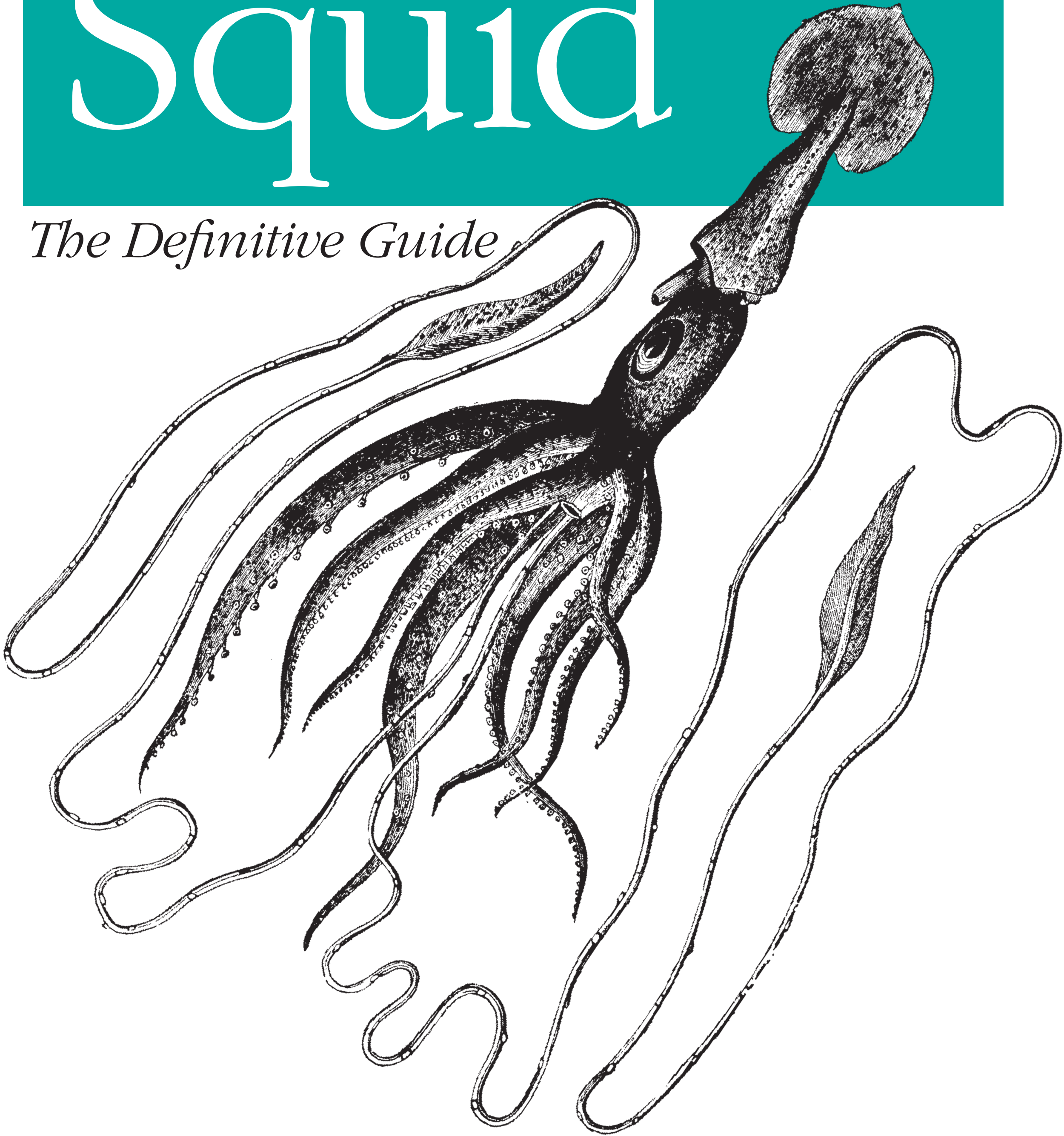
Making the Most of Your Internet Connection

Squid

The Definitive Guide



O'REILLY®

Duane Wessels

CHAPTER 8

Advanced Disk Cache Topics

Performance is one of the biggest concerns for Squid administrators. As the load placed on Squid increases, disk I/O is typically the primary bottleneck. The reason for this performance limitation is due to the importance that Unix filesystems place on consistency after a system crash.

By default, Squid uses a relatively simple storage scheme (*ufs*). All disk I/O is performed by the main Squid process. With traditional Unix filesystems, certain operations always block the calling process. For example, calling `open( )` on the Unix Fast Filesystem (UFS) causes the operating system to allocate and initialize certain on-disk data structures. The system call doesn't return until these I/O operations complete, which may take longer than you'd like if the disks are already busy with other tasks.

Under heavy load, these filesystem operations can block the Squid process for small, but significant, amounts of time. The point at which the filesystem becomes a bottleneck depends on many different factors, including:

- The number of disk drives
- The rotational speed and seek time of your hard drives
- The type of disk drive interface (ATA, SCSI)
- Filesystem tuning options
- The number of files and percentage of free space

Do I Have a Disk I/O Bottleneck?

Web caches such as Squid don't usually come right out and tell you when disk I/O is becoming a bottleneck. Instead, response time and/or hit ratio degrade as load increases. The tricky thing is that response time and hit ratio may be changing for other reasons, such as increased network latency and changes in client request patterns.

Perhaps the best way to explore the performance limits of your cache is with a benchmark, such as Web Polygraph. The good thing about a benchmark is that you can fully control the environment and eliminate many unknowns. You can also repeat the same experiment with different cache configurations. Unfortunately, benchmarking often takes a lot of time and requires spare systems that aren't already being used.

If you have the resources to benchmark Squid, begin with a standard caching workload. As you increase the load, at some point you should see a significant increase in response time and/or a decrease in hit ratio. Once you observe this performance degradation, run the experiment again but with disk caching disabled. You can configure Squid never to cache any response (with the *null* storage scheme, see the later section "The null Storage Scheme"). Alternatively, you can configure the workload to have 100% uncachable responses. If the average response time is significantly better without caching, you can be relatively certain that disk I/O is a bottleneck at that level of throughput.

If you're like most people, you have neither the time nor resources to benchmark Squid. In this case, you can examine Squid's runtime statistics to look for disk I/O bottlenecks. The cache manager General Runtime Information page (see Chapter 14) gives you median response times for both cache hits and misses:

```
Median Service Times (seconds)  5 min    60 min:
        HTTP Requests (All):   0.39928  0.35832
        Cache Misses:          0.42149  0.39928
        Cache Hits:            0.12783  0.11465
        Near Hits:             0.37825  0.39928
        Not-Modified Replies:  0.07825  0.07409
```

For a healthy Squid cache, hits are significantly faster than misses. Your median hit response time should usually be 0.5 seconds or less. I strongly recommend that you use SNMP or another network monitoring tool to collect periodic measurements from your Squid caches (see Chapter 14). A significant (factor of two) increase in median hit response time is a good indication that you have a disk I/O bottleneck.

If you believe your production cache is suffering in this manner, you can test your theory with the same technique mentioned previously. Configure Squid not to cache any responses, thus avoiding all disk I/O. Then closely observe the *cache miss* response time. If it goes down, your theory is probably correct.

Once you've convinced yourself that disk throughput is limiting Squid's performance, you can try a number of things to improve it. Some of these require recompiling Squid, while others are relatively simple steps you can take to tune the Unix filesystems.

Filesystem Tuning Options

First of all, you should never use RAID for Squid cache directories. In my experience, RAID always degrades filesystem performance for Squid. It is much better to have a number of separate filesystems, each dedicated to a single disk drive.

I have found four simple ways to improve UFS performance for Squid. Some of these are specific to certain operating systems, such as BSD and Linux, and may not be available on your platform:

- Some UFS implementations support a *noatime* mount option. Filesystems mounted with *noatime* don't update the inode access time value for reads. The easiest way to use this option is to add it to the */etc/fstab* like this:

```
# Device            Mountpoint    FStype  Options       Dump    Pass#
/dev/ad1s1c         /cache0       ufs     rw,noatime    0       0
```

- Check your `mount(8)` manpage for the *async* option. With this option set, certain I/O operations (such as directory updates) may be performed asynchronously. The documentation for some systems notes that it is a dangerous flag. Should your system crash, you may lose the entire filesystem. For many installations, the performance improvement is worth the risk. You should use this option only if you don't mind losing the contents of your entire cache. If the cached data is very valuable, the *async* option is probably not for you.

- BSD has a feature called *soft updates*. Soft updates are BSD's alternative to journaling filesystems.[*] On FreeBSD, you can enable this option on an unmounted filesystem with the `tunefs` command:

```
# umount /cache0
# tunefs -n enable /cache0
# mount /cache0
```

  You only have to run the `tunefs` once for each filesystem. Soft updates are automatically enabled on the filesystem again when your system reboots.

  On OpenBSD and NetBSD, you can use the *softdep* mount option:

```
# Device            Mountpoint    FStype  Options       Dump    Pass#
/dev/sd0f           /usr          ffs     rw,softdep    1       2
```

If you're like me, you're probably wondering what the difference is between the async option and soft updates. One important difference is that soft update code has been designed to maintain filesystem consistency in the event of a system crash, while the async option has not. This might lead you to conclude that async performs better than soft updates. However, as I show in Appendix D, the opposite is true.

[*] For further information, please see "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast File System" by Marshall Kirk McKusik and Gregory R. Ganger. *Proceedings of the 1999 USENIX Annual Technical Conference,* June 6–11, 1999, Monterey, California.

Previously, I mentioned that UFS performance, especially writing, depends on the amount of free space. Disk writes for empty filesystems are much faster than for full ones. This is one reason behind UFS's *minfree* parameter and space/time optimization tradeoffs. If your cache disks are full and Squid's performance seems bad, try reducing the *cache_dir* capacity values so that more free space is available. Of course, this reduction in cache size also decreases your hit ratio, but the response time improvement may be worth it. If you're buying the components for a new Squid cache, consider getting much larger disks than you need and using only half the space.

Alternative Filesystems

Some operating systems support filesystems other than UFS (or ext2fs). Journaling filesystems are a common alternative. The primary difference between UFS and journaling filesystems is in the way that they handle updates. With UFS, updates are made in-place. For example, when you change a file and save it to disk, the new data replaces the old data. When you remove a file, UFS updates the directory directly.

A journaling filesystem, on the other hand, writes updates to a separate journal, or log file. You can typically select whether to journal file changes, metadata changes, or both. A background process reads the journal during idle moments and applies the actual changes. Journaling filesystems typically recover much faster from crashes than UFS. After a crash, the filesystem simply reads the journal and commits all the outstanding changes.

The primary drawback of journaling filesystems is that they require additional disk writes. Changes are first written to the log and later to the actual files and/or directories. This is particularly relevant for web caches because they tend to have more disk writes than reads in the first place.

Journaling filesystems are available for a number of operating systems. On Linux, you can choose from ext3fs, reiserfs, XFS, and others. XFS is also available for SGI/IRIX, where it was originally developed. Solaris users can use the Veritas filesystem product. The TRU64 (formerly Digital Unix) Advanced Filesystem (advfs) supports journaling.

You can use a journaling filesystem without making any changes to Squid's configuration. Simply create and mount the filesystem as described in your operating system documentation. You don't need to change the *cache_dir* line in *squid.conf*. Use a command like this to make a reiserfs filesystem on Linux:

```
# /sbin/mkreiserfs /dev/sda2
```

For XFS, use:

```
# mkfs -t xfs -f /dev/sda2
```

Note that ext3fs is simply ext2fs with journaling enabled. Use the `-j` option to `mke2fs` when creating the filesystem:

```
# /sbin/mke2fs -j /dev/sda2
```

Refer to your documentation (e.g., manpages) for other operating systems.

The aufs Storage Scheme

The *aufs* storage scheme has evolved out of the very first attempt to improve Squid's disk I/O response time. The "a" stands for asynchronous I/O. The only difference between the default *ufs* scheme and *aufs* is that I/Os aren't executed by the main Squid process. The data layout and format is the same, so you can easily switch between the two schemes without losing any cache data.

aufs uses a number of thread processes for disk I/O operations. Each time Squid needs to read, write, open, close, or remove a cache file, the I/O request is dispatched to one of the thread processes. When the thread completes the I/O, it signals the main Squid process and returns a status code. Actually, in Squid 2.5, certain file operations aren't executed asynchronously by default. Most notably, disk writes are always performed synchronously. You can change this by setting `ASYNC_WRITE` to `1` in *src/fs/aufs/store_asyncufs.h* and recompiling.

The *aufs* code requires a *pthreads* library. This is the standard threads interface, defined by POSIX. Even though *pthreads* is available on many Unix systems, I often encounter compatibility problems and differences. The *aufs* storage system seems to run well only on Linux and Solaris. Even though the code compiles, you may encounter serious problem on other operating systems.

To use *aufs*, you must add a special `./configure` option:

```
% ./configure --enable-storeio=aufs,ufs
```

Strictly speaking, you don't really need to specify *ufs* in the list of `storeio` modules. However, you might as well because if you try *aufs* and don't like it, you'll be able to fall back to the plain *ufs* storage scheme.

You can also use the `--with-aio-threads=N` option if you like. If you omit it, Squid automatically calculates the number of threads to use based on the number of *aufs* *cache_dir*s. Table 8-1 shows the default number of threads for up to six cache directories.

Table 8-1. Default number of threads for up to six cache directories

cache_dirs	Threads
1	16
2	26
3	32

Table 8-1. Default number of threads for up to six cache directories (continued)

cache_dirs	Threads
4	36
5	40
6	44

After you compile *aufs* support into Squid, you can specify it on a *cache_dir* line in *squid.conf*:

```
cache_dir aufs /cache0 4096 16 256
```

After starting Squid with *aufs* enabled, make sure everything still works correctly. You may want to run `tail -f store.log` for a while to make sure that objects are being swapped out to disk. You should also run `tail -f cache.log` and look for any new errors or warnings.

How aufs Works

Squid creates a number of thread processes by calling `pthread_create()`. All threads are created upon the first disk activity. Thus, you'll see all the thread processes even if Squid is idle.

Whenever Squid wants to perform some disk I/O operation (e.g., to open a file for reading), it allocates a couple of data structures and places the I/O request into a queue. The thread processes have a loop that take I/O requests from the queue and executes them. Because the request queue is shared by all threads, Squid uses mutex locks to ensure that only one thread updates the queue at a given time.

The I/O operations block the thread process until they are complete. Then, the status of the operation is placed on a *done queue*. The main Squid process periodically checks the done queue for completed operations. The module that requested the disk I/O is notified that the operation is complete, and the request or response processing proceeds.

As you may have guessed, *aufs* can take advantage of systems with multiple CPUs. The only locking that occurs is on the request and result queues. Otherwise, all other functions execute independently. While the main process executes on one CPU, another CPU handles the actual I/O system calls.

aufs Issues

An interesting property of threads is that all processes share the same resources, including memory and file descriptors. For example, when a thread process opens a file as descriptor 27, all other threads can then access that file with the same descriptor number. As you probably know, file-descriptor shortage is a common problem with first-time Squid administrators. Unix kernels typically have two file-descriptor

limits: per process and systemwide. While you might think that 256 file descriptors per process is plenty (because of all the thread processes), it doesn't work that way. In this case, all threads share that small number of descriptors. Be sure to increase your system's per-process file descriptor limit to 4096 or higher, especially when using *aufs*.

Tuning the number of threads can be tricky. In some cases, you might see this warning in *cache.log*:

```
2003/09/29 13:42:47| squidaio_queue_request: WARNING - Disk I/O overloading
```

It means that Squid has a large number of I/O operations queued up, waiting for an available thread. Your first instinct may be to increase the number of threads. I would suggest, however, that you decrease the number instead.

Increasing the number of threads also increases the queue size. Past a certain point, it doesn't increase *aufs*'s load capacity. It only means that more operations become queued. Longer queues result in higher response times, which is probably something you'd like to avoid.

Decreasing the number of threads, and the queue size, means that Squid can detect the overload condition faster. When a *cache_dir* is overloaded, it is removed from the selection algorithm (see Chapter 7). Then, Squid either chooses a different *cache_dir* or simply doesn't store the response on disk. This may be a better situation for your users. Even though the hit ratio goes down, response time remains relatively low.

Monitoring aufs Operation

The Async IO Counters option in the cache manager menu displays a few statistics relating to *aufs*. It shows counters for the number of open, close, read, write, stat, and unlink requests received. For example:

```
% squidclient mgr:squidaio_counts
...
ASYNC IO Counters:
Operation       # Requests
open              15318822
close             15318813
cancel            15318813
write                    0
read              19237139
stat                     0
unlink             2484325
check_callback   311678364
queue                    0
```

The cancel counter is normally equal to the close counter. This is because the close function always calls the cancel function to ensure that any pending I/O operations are ignored.

The write counter is zero because this version of Squid performs writes synchronously, even for *aufs*.

The check_callback counter shows how many times the main Squid process has checked the done queue for completed operations.

The queue value indicates the current length of the request queue. Normally, the queue length should be less than the number of threads × 5. If you repeatedly observe a queue length larger than this, you may be pushing Squid too hard. Adding more threads may help but only to a certain point.

The diskd Storage Scheme

diskd (short for disk daemons) is similar to *aufs* in that disk I/Os are executed by external processes. Unlike *aufs*, however, *diskd* doesn't use threads. Instead, interprocess communication occurs via message queues and shared memory.

Message queues are a standard feature of modern Unix operating systems. They were invented many years ago in AT&T's Unix System V, Release 1. The messages passed between processes on these queues are relatively small: 32–40 bytes. Each *diskd* process uses one queue for receiving requests from Squid and another queue for transmitting results back.

How diskd Works

Squid creates one *diskd* process for each *cache_dir*. This is different from *aufs*, which uses a large pool of threads for all *cache_dir*s. Squid sends a message to the corresponding *diskd* process for each I/O operation. When that operation is complete, the *diskd* process sends a status message back to Squid. Squid and the *diskd* processes preserve the order of messages in the queues. Thus, there is no concern that I/Os might be executed out of sequence.

For reads and writes, Squid and the *diskd* processes use a shared memory area. Both processes can read from, and write to, this area of memory. For example, when Squid issues a read request, it tells the *diskd* process where to place the data in memory. *diskd* passes this memory location to the `read()` system call and notifies Squid that the read is complete by sending a message on the return queue. Squid then accesses the recently read data from the shared memory area.

diskd (as with *aufs*) essentially gives Squid nonblocking disk I/Os. While the *diskd* processes are blocked on I/O operations, Squid is free to work on other tasks. This works really well as long as the *diskd* processes can keep up with the load. Because the main Squid process is now able to do more work, it's possible that it may overload the *diskd* helpers. The *diskd* implementation has two features to help out in this situation.

First, Squid waits for the *diskd* processes to catch up if one of the queues exceeds a certain threshold. The default value is 64 outstanding messages. If a *diskd* process gets this far behind, Squid "sleeps" a small amount of time and waits for it to complete some of the pending operations. This essentially puts Squid into a blocking I/O mode. It also makes more CPU time available to the *diskd* processes. You can configure this threshold by specifying a value for the `Q2` parameter on a *cache_dir* line:

```
cache_dir diskd /cache0 7000 16 256 Q2=50
```

Second, Squid stops asking the *diskd* process to open files if the number of outstanding operations reaches another threshold. Here, the default value is 72 messages. If Squid would like to open a disk file for reading or writing, but the selected *cache_dir* has too many pending operations, the open request fails internally. When trying to open a file for reading, this causes a cache miss instead of a cache hit. When opening files for writing, it prevents Squid from storing a cachable response. In both cases the user still receives a valid response. The only real effect is that Squid's hit ratio decreases. This threshold is configurable with the Q1 parameter:

```
cache_dir diskd /cache0 7000 16 256 Q1=60 Q2=50
```

Note that in some versions of Squid, the Q1 and Q2 parameters are mixed-up in the default configuration file. For optimal performance, Q1 should be greater than Q2.

Compiling and Configuring diskd

To use *diskd*, you must add it to the `--enable-storeio` list when running `./configure`:

```
% ./configure --enable-storeio=ufs,diskd
```

diskd seems to be portable since shared memory and message queues are widely supported on modern Unix systems. However, you'll probably need to adjust a few kernel limits relating to both. Kernels typically have the following variables or parameters:

MSGMNB
: This is the maximum characters (octets) per message queue. With *diskd*, the practical limit is about 100 outstanding messages per queue. The messages that Squid passes are 32–40 octets, depending on your CPU architecture. Thus, MSGMNB should be 4000 or more. To be safe, I recommend setting this to 8192.

MSGMNI
: This is the maximum number of message queues for the whole system. Squid uses two queues for each *diskd cache_dir*. If you have 10 disks, that's 20 queues. You should probably add even more in case other applications also use message queues. I recommend a value of 40.

MSGSSZ
: This is the size of a message segment, in octets. Messages larger than this size are split into multiple segments. I usually set this to 64 so that the *diskd* message isn't split into multiple segments.

MSGSEG
: This is the maximum number of message segments that can exist in a single queue. Squid normally limits the queues to 100 outstanding messages. Remember that if you don't increase MSGSSZ to 64 on 64-bit architectures, each message requires more than one segment. To be safe, I recommend setting this to 512.

MSGTQL
: This is the maximum number of messages that can exist in the whole system. It should be at least 100 multiplied by the number of *cache_dir*s. I recommend setting it to 2048, which should be more than enough for as many as 10 cache directories.

MSGMAX
: This is the maximum size of a single message. For Squid, 64 bytes should be sufficient. However, your system may have other applications that use larger messages. On some operating systems such as BSD, you don't need to set this. BSD automatically sets it to MSGSSZ × MSGSEG. On other systems you may need to increase the value from its default. In this case, you can set it to the same as MSGMNB.

SHMSEG
: This is the maximum number of shared memory segments allowed per process. Squid uses one shared memory identifier for each *cache_dir*. I recommend a setting of 16 or higher.

SHMMNI
: This is the systemwide limit on the number of shared memory segments. A value of 40 is probably enough in most cases.

SHMMAX
: This is the maximum size of a single shared memory segment. By default, Squid uses about 409,600 bytes for each segment. Just to be safe, I recommend setting this to 2 MB, or 2,097,152.

SHMALL
: This is the systemwide limit on the amount of shared memory that can be allocated. On some systems, SHMALL may be expressed as a number of pages, rather than bytes. Setting this to 16 MB (4096 pages) is enough for 10 *cache_dir*s with plenty remaining for other applications.

To configure message queues on BSD, add these options to your kernel configuration file:*

```
# System V message queues and tunable parameters
options         SYSVMSG         # include support for message queues
options         MSGMNB=8192     # max characters per message queue
options         MSGMNI=40       # max number of message queue identifiers
options         MSGSEG=512      # max number of message segments per queue
options         MSGSSZ=64       # size of a message segment MUST be power of 2
options         MSGTQL=2048     # max number of messages in the system
options         SYSVSHM
options         SHMSEG=16       # max shared mem segments per process
options         SHMMNI=32       # max shared mem segments in the system
options         SHMMAX=2097152  # max size of a shared mem segment
options         SHMALL=4096     # max size of all shared memory (pages)
```

To configure message queues on Linux, add these lines to */etc/sysctl.conf*:

```
kernel.msgmnb=8192
kernel.msgmni=40
kernel.msgmax=8192
kernel.shmall=2097152
kernel.shmmni=32
kernel.shmmax=16777216
```

Alternatively, or if you find that you need more control, you can manually edit *include/linux/msg.h* and *include/linux/shm.h* in your kernel sources.

For Solaris, add these lines to */etc/system* and then reboot:

```
set msgsys:msginfo_msgmax=8192
set msgsys:msginfo_msgmnb=8192
set msgsys:msginfo_msgmni=40
set msgsys:msginfo_msgssz=64
set msgsys:msginfo_msgtql=2048
set shmsys:shminfo_shmmax=2097152
set shmsys:shminfo_shmmni=32
set shmsys:shminfo_shmseg=16
```

For Digital Unix (TRU64), you can probably add lines to the kernel configuration in the style of BSD, seen previously. Alternatively, you can use the `sysconfig` command. First, create a file called *ipc.stanza* like this:

```
ipc:
          msg-max = 2048
          msg-mni = 40
          msg-tql = 2048
          msg-mnb = 8192
          shm-seg = 16
          shm-mni = 32
          shm-max = 2097152
          shm-max = 4096
```

* OpenBSD is a little different. Use `option` instead of `options`, and specify the SHMMAX value in pages, rather than bytes.

Now, run this command and reboot:

```
# sysconfigdb -a -f ipc.stanza
```

After you have message queues and shared memory configured in your operating system, you can add the *cache_dir* lines to *squid.conf*:

```
cache_dir diskd /cache0 7000 16 256 Q1=72 Q2=64
cache_dir diskd /cache1 7000 16 256 Q1=72 Q2=64
...
```

If you forget to increase the message queue limits, or if you don't set them high enough, you'll see messages like this in *cache.log*:

```
2003/09/29 01:30:11| storeDiskdSend: msgsnd: (35) Resource temporarily unavailable
```

Monitoring diskd

The best way to monitor *diskd* performance is with the cache manager. Request the *diskd* page; for example:

```
% squidclient mgr:diskd
...
sent_count: 755627
recv_count: 755627
max_away: 14
max_shmuse: 14
open_fail_queue_len: 0
block_queue_len: 0

             OPS SUCCESS    FAIL
   open   51534   51530       4
 create   67232   67232       0
  close  118762  118762       0
 unlink   56527   56526       1
   read   98157   98153       0
  write  363415  363415       0
```

See Chapter 14 for a description of this output.

The coss Storage Scheme

The Cyclic Object Storage Scheme (*coss*) is an attempt to develop a custom filesystem for Squid. With the *ufs*-based schemes, the primary performance bottleneck comes from the need to execute so many `open()` and `unlink()` system calls. Because each cached response is stored in a separate disk file, Squid is always opening, closing, and removing files.

coss, on the other hand, uses one big file to store all responses. In this sense, it is a small, custom filesystem specifically for Squid. *coss* implements many of the functions normally handled by the underlying filesystem, such as allocating space for new data and remembering where there is free space.

Unfortunately, *coss* is still a little rough around the edges. Development of *coss* has been proceeding slowly over the last couple of years. Nonetheless, I'll describe it here in case you feel adventurous.

How coss Works

On the disk, each *coss cache_dir* is just one big file. The file grows in size until it reaches its maximum size. At this point, Squid starts over at the beginning of the file, overwriting any data already stored there. Thus, new objects are always stored at the "end" of this cyclic file.*

Squid actually doesn't write new object data to disk immediately. Instead, the data is copied into a 1-MB memory buffer, called a *stripe*. A stripe is written to disk when it becomes full. *coss* uses asynchronous writes so that the main Squid process doesn't become blocked on disk I/O.

As with other filesystems, *coss* also uses the *blocksize* concept. Back in Chapter 7, I talked about file numbers. Each cached object has a file number that Squid uses to locate the data on disk. For *coss*, the file number is the same as the block number. For example, a cached object with a swap file number equal to 112 starts at the 112th block in a *coss* filesystem. File numbers aren't allocated sequentially with *coss*. Some file numbers are unavailable because cached objects generally occupy more than one block in the *coss* file.

The *coss* block size is configurable with a *cache_dir* option. Because Squid's file numbers are only 24 bits, the block size determines the maximum size of a *coss* cache directory: size = block_size $\times 2^{24}$. For example, with a 512-byte block size, you can store up to 8 GB in a *coss cache_dir*.

coss doesn't implement any of Squid's normal cache replacement algorithms (see Chapter 7). Instead, cache hits are "moved" to the end of the cyclic file. This is, essentially, the LRU algorithm. It does, unfortunately, mean that cache hits cause disk writes, albeit indirectly.

With *coss*, there is no need to unlink or remove cached objects. Squid simply forgets about the space allocated to objects that are removed. The space will be reused eventually when the end of the cyclic file reaches that place again.

Compiling and Configuring coss

To use *coss*, you must add it to the `--enable-storeio` list when running `./configure`:

```
% ./configure --enable-storeio=ufs,coss ...
```

* The beginning is the location where data was first written; the end is the location where data was most recently written.

coss cache directories require a `max-size` option. Its value must be less than the stripe size (1 MB by default, but configurable with the `--enable-coss-membuf-size` option). Also note that you must omit the L1 and L2 values that are normally present for *ufs*-based schemes. Here is an example:

```
cache_dir coss /cache0/coss 7000 max-size=1000000
cache_dir coss /cache1/coss 7000 max-size=1000000
cache_dir coss /cache2/coss 7000 max-size=1000000
cache_dir coss /cache3/coss 7000 max-size=1000000
cache_dir coss /cache4/coss 7000 max-size=1000000
```

Furthermore, you can change the default *coss* block size with the `block-size` option:

```
cache_dir coss /cache0/coss 30000 max-size=1000000 block-size=2048
```

One tricky thing about *coss* is that the *cache_dir* directory argument (e.g., */cache0/coss*) isn't actually a directory. Instead, it is a regular file that Squid opens, and creates if necessary. This is so you can use raw partitions as *coss* files. If you mistakenly create the *coss* file as a directory, you'll see an error like this when starting Squid:

```
2003/09/29 18:51:42|  /usr/local/squid/var/cache: (21) Is a directory
FATAL: storeCossDirInit: Failed to open a coss file.
```

Because the *cache_dir* argument isn't a directory, you must use the *cache_swap_log* directive (see Chapter 13). Otherwise Squid attempts to create a *swap.state* file in the *cache_dir* directory. In that case, you'll see an error like this:

```
2003/09/29 18:53:38| /usr/local/squid/var/cache/coss/swap.state:
        (2) No such file or directory
FATAL: storeCossDirOpenSwapLog: Failed to open swap log.
```

coss uses asynchronous I/Os for better performance. In particular, it uses the `aio_read()` and `aio_write()` system calls. These may not be available on all operating systems. At this time, they are available on FreeBSD, Solaris, and Linux. If the *coss* code seems to compile okay, but you get a "Function not implemented" error message, you need to enable these system calls in your kernel. On FreeBSD, your kernel must have this option:

```
options         VFS_AIO
```

coss Issues

coss is still an experimental feature. The code has not yet proven stable enough for everyday use. If you want to play with and help improve it, be prepared to lose any data stored in a *coss cache_dir*. On the plus side, *coss*'s preliminary performance tests are very good. For an example, see Appendix D.

coss doesn't support rebuilding cached data from disk very well. When you restart Squid, you might find that it fails to read the *coss swap.state* files, thus losing any cached data. Furthermore, Squid doesn't remember its place in the cyclic file after a restart. It always starts back at the beginning.

coss takes a nonstandard approach to object replacement. This may cause a lower hit ratio than you might get with one of the other storage schemes.

Some operating systems have problems with files larger than 2 GB. If this happens to you, you can always create more, smaller *coss* areas. For example:

```
cache_dir coss /cache0/coss0 1900 max-size=1000000 block-size=128
cache_dir coss /cache0/coss1 1900 max-size=1000000 block-size=128
cache_dir coss /cache0/coss2 1900 max-size=1000000 block-size=128
cache_dir coss /cache0/coss3 1900 max-size=1000000 block-size=128
```

Using a raw disk device (e.g., */dev/da0s1c*) doesn't work very well yet. One reason is that disk devices usually require that I/Os take place on 512-byte block boundaries. Another concern is that direct disk access bypasses the systems buffer cache and may degrade performance. Many disk drives, however, have built-in caches these days.

The null Storage Scheme

Squid has a fifth storage scheme called *null*. As the name implies, this is more of a nonstorage scheme. Files that are "written" to a *null cache_dir* aren't actually written to disk.

Most people won't have any reason to use the *null* storage system. It's primarily useful if you want to entirely disable Squid's disk cache.* You can't simply remove all *cache_dir* lines from *squid.conf* because then Squid adds a default *ufs cache_dir*. The *null* storage system is also sometimes useful for testing and benchmarking Squid. Since the filesystem is typically the performance bottleneck, using the *null* storage scheme gives you an upper limit of Squid's performance on your hardware.

To use this scheme you must first specify it on the `--enable-storeio` list when running `./configure`:

```
% ./configure --enable-storeio=ufs,null ...
```

You can then create a *cache_dir* of type `null` in *squid.conf*:

```
cache_dir /tmp null
```

It may seem odd that you need to specify a directory for the *null* storage scheme. However, Squid uses the directory name as a *cache_dir* identifier. For example, you'll see it in the cache manager output (see Chapter 14).

Which Is Best for Me?

Squid's storage scheme choices may seem a little overwhelming and confusing. Is *aufs* better than *diskd*? Does my system support *aufs* or *coss*? Will I lose my data if I use one of these fancy schemes? Is it okay to mix-and-match storage schemes?

* Some responses may still be cached in memory, however.

First of all, if your Squid is lightly used (say, less than five requests per second), the default *ufs* storage scheme should be sufficient. You probably won't see a noticeable performance improvement from the other schemes at this low request rate.

If you are trying to decide which scheme to try, your operating system may be a determining factor. For example, *aufs* runs well on Linux and Solaris but seems to have problems on other systems. The *coss* code uses functions that aren't available on certain operating systems (e.g., NetBSD) at this time.

It seems to me that higher-performing storage schemes are also more susceptible to data loss in the event of a system crash. This is the tradeoff for better performance. For many people, however, cached data is of relatively low value. If Squid's cache becomes corrupted due to a crash, you may find it easier to simply `newfs` the disk partition and let the cache fill back up from scratch. If you find it difficult or expensive to replace the contents of Squid's cache, you probably want to use one of the slow, but reliable, filesystems and storage schemes.

Squid certainly allows you to use different filesystems and storage schemes for each *cache_dir*. In practice, however, this is uncommon. You'll probably have fewer hassles if all cache directories are approximately the same size and use the same storage scheme.

Exercises

- Try to compile all possible storage schemes on your system.
- Run Squid with a separate *cache_dir* for each storage scheme you can get to compile.
- Run Squid with one or more *diskd cache_dir*s. Then run the **`ipcs -o`** command.