5

File Systems

File systems are typically observed as a layer between an application and the I/O services providing the underlying storage. When you look at file system performance, you should focus on the latencies observed at the application level. Historically, however, we have focused on techniques that look at the latency and throughput characteristics of the underlying storage and have been flying in the dark about the real latencies seen at the application level.

With the advent of DTrace, we now have end-to-end observability, from the application all the way through to the underlying storage. This makes it possible to do the following:

- Observe the latency and performance impact of file-level requests at the application level.
- Attribute physical I/O by applications and/or files.
- Identify performance characteristics contributed by the file system layer, in between the application and the I/O services.

5.1 Layers of File System and I/O

We can observe file system activity at three key layers:

- **I/O layer.** At the bottom of a file system is the I/O subsystem providing the backend storage for the file system. For a disk-based file system, this is typically

the block I/O layer. Other file systems (for example, NFS) might use networks or other services to provide backend storage.

- **POSIX libraries and system calls.** Applications typically perform I/O through POSIX library interfaces. For example, an application needing to open and read a file would call `open(2)` followed by `read(2)`.

  Most POSIX interfaces map directly to system calls, the exceptions being the asynchronous I/O interfaces. These are emulated by user-level thread libraries on top of POSIX `pread`/`pwrite`.

  You can trace at this layer with a variety of tools—`truss` and DTrace can trace the system calls on behalf of the application. `truss` has significant overhead when used at this level since it starts and stops the application at every system call. In contrast, DTrace typically only adds a few microseconds to each call.

- **VOP layer.** Solaris provides a layer of common entry points between the upper-level system calls and the file system—the file system vnode operations (VOP) interface layer. We can instrument these layers easily with DTrace. We've historically made special one-off tools to monitor at this layer by using kernel VOP-level interposer modules, a practice that adds significant instability risk and performance overhead.

Figure 5.1 shows the end-to-end layers for an application performing I/O through a file system.

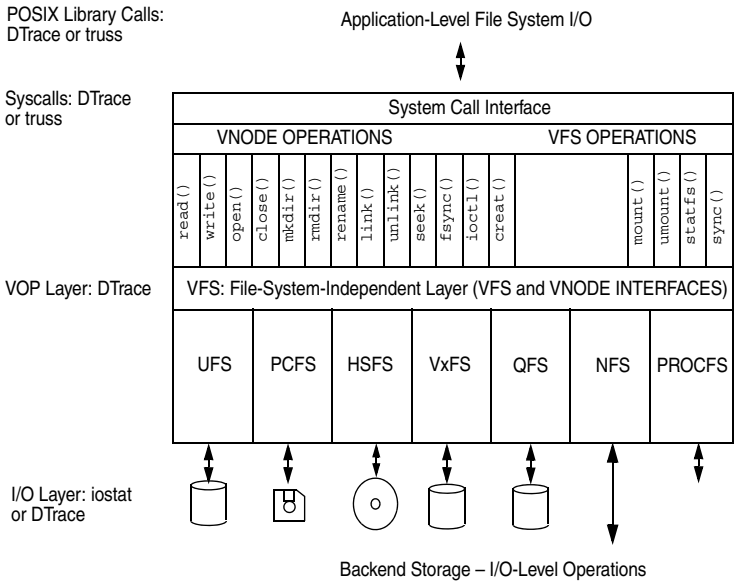

Figure 5.1 Layers for Observing File System I/O

5.2 Observing Physical I/O

The traditional method of observing file system activity is to induce information from the bottom end of the file system, for example, physical I/O. This can be done easily with `iostat` or DTrace, as shown in the following `iostat` example and further in Chapter 4.

```
$ iostat -xnczpm 3
     cpu
 us sy wt id
  7  2  8 83
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
    0.6    3.8    8.0   30.3  0.1  0.2   20.4   37.7   0   3 c0t0d0
    0.6    3.8    8.0   30.3  0.1  0.2   20.4   37.7   0   3 c0t0d0s0 (/)
    0.0    0.0    0.0    0.0  0.0  0.0    0.0   48.7   0   0 c0t0d0s1
    0.0    0.0    0.0    0.0  0.0  0.0    0.0    0.0   0   0 c0t0d0s2
    0.0    0.0    0.0    0.0  0.0  0.0  405.2 1328.5   0   0 c0t1d0
    0.0    0.0    0.0    0.0  0.0  0.0  405.9 1330.8   0   0 c0t1d0s1
    0.0    0.0    0.0    0.0  0.0  0.0    0.0    0.0   0   0 c0t1d0s2
   14.7    4.8  330.8    6.8  0.0  0.3    0.0   13.9   0   8 c4t16d1
   14.7    4.8  330.8    6.8  0.0  0.3    0.0   13.9   0   8 c4t16d1s7 (/export/home)
    1.4    0.4   70.4    4.3  0.0  0.0    0.0   21.8   0   2 c4t16d2
    1.4    0.4   70.4    4.3  0.0  0.0    0.0   21.8   0   2 c4t16d2s7 (/export/home2)
   12.8   12.4   73.5    7.4  0.0  0.1    0.0    2.5   0   3 c4t17d0
   10.8   10.8    0.4    0.4  0.0  0.0    0.0    0.0   0   0 c4t17d0s2
    2.0    1.6   73.1    7.0  0.0  0.1    0.0   17.8   0   3 c4t17d0s7 (/www)
    0.0    2.9    0.0  370.4  0.0  0.1    0.0   19.1   0   6 rmt/1
```

Using `iostat`, we can observe I/O counts, bandwidth, and latency at the device level, and optionally per-mount, by using the `-m` option (note that this only works for file systems like UFS that mount only one device). In the above example, we can see that `/export/home` is mounted on `c4t16d1s7`. It is generating 14.7 reads per second and 4.8 writes per second, with a response time of 13.9 milliseconds. But that's all we know—far too often we deduce too much by simply looking at the physical I/O characteristics. For example, in this case we could easily assume that the upper-level application is experiencing good response times, when in fact substantial latency is being added in the file system layer, which is masked by these statistics. We talk more about common scenarios in which latency is added in the file system layer in Section 5.4.

By using the DTrace I/O provider, we can easily connect physical I/O events with some file-system-level information; for example, *file names*. The script from Section 5.4.3 shows a simple example of how DTrace can display per-operation information with combined file-system-level and physical I/O information.

```
# ./iotrace.d

    DEVICE                                                  FILE RW     SIZE
     cmdk0                                /export/home/rmc/.sh_history  W     4096
     cmdk0                                  /opt/Acrobat4/bin/acroread  R     8192
     cmdk0                                  /opt/Acrobat4/bin/acroread  R     1024
     cmdk0                                  /var/tmp/wscon-:0.0-gLaW9a  W     3072
     cmdk0                             /opt/Acrobat4/Reader/AcroVersion  R     1024
     cmdk0          /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R     8192
     cmdk0          /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R     8192
     cmdk0          /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R     4096
     cmdk0          /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R     8192
     cmdk0          /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R     8192
```

5.3 File System Latency

When analyzing performance, consider the file system as a black box. Look at the latency as it impacts the application and then identify the causes of the latency. For example, if an application is making `read()` calls at the POSIX layer, your first interest should be in how long each `read()` takes as a percentage of the overall application thread-response time. Only when you want to dig deeper should you consider the I/O latency behind the `read()`, such as disk service times—which ironically is where the performance investigation has historically begun. Figure 5.2 shows an example of how you can estimate performance. You can evaluate the percentage of time in the file system (*Tfilesys*) against the total elapsed time (*Ttotal*).

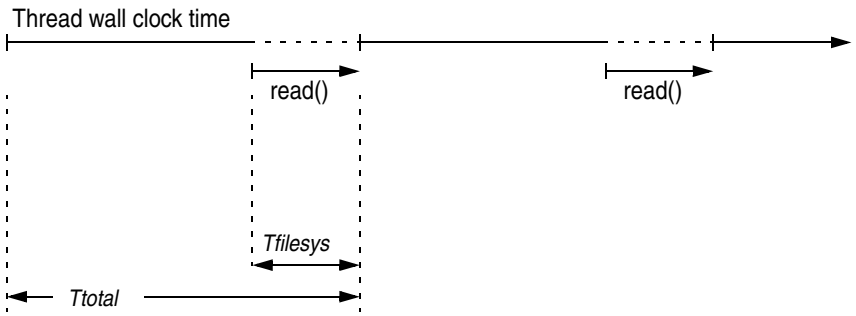

Figure 5.2 Estimating File System Performance Impact

Using `truss`, you can examine the POSIX-level I/O calls. You can observe the file descriptor and the size and duration for each logical I/O. In the following example, you can see `read()` and `write()` calls during a `dd` between two files.

```
# dd if=filea of=fileb bs=1024k&

# truss -D -p 13092
13092:   0.0326 read(3, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576)  = 1048576
13092:   0.0186 write(4, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576) = 1048576
13092:   0.0293 read(3, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576)  = 1048576
13092:   0.0259 write(4, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576) = 1048576
13092:   0.0305 read(3, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576)  = 1048576
13092:   0.0267 write(4, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576) = 1048576
13092:   0.0242 read(3, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576)  = 1048576
13092:   0.0184 write(4, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576) = 1048576
13092:   0.0368 read(3, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576)  = 1048576
13092:   0.0333 write(4, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576) = 1048576
13092:   0.0297 read(3, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576)  = 1048576
13092:   0.0175 write(4, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576) = 1048576
13092:   0.0315 read(3, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576)  = 1048576
13092:   0.0231 write(4, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576) = 1048576
13092:   0.0338 read(3, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576)  = 1048576
13092:   0.0181 write(4, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576) = 1048576
13092:   0.0381 read(3, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576)  = 1048576
13092:   0.0177 write(4, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576) = 1048576
13092:   0.0323 read(3, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576)  = 1048576
13092:   0.0199 write(4, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576) = 1048576
13092:   0.0364 read(3, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576)  = 1048576
13092:   0.0189 write(4, "\0\0\0\0\0\0\0\0\0\0\0\0".., 1048576) = 1048576
...
```

The `truss` example shows that `read()` occurs on file descriptor 3 with an average response time of 30 ms and `write()` occurs on file descriptor 4 with an average response time of 25 ms. This gives some insight into the high-level activity but no other process statistics with which to formulate any baselines.

By using DTrace, you could gather a little more information about the proportion of the time taken to perform I/O in relation to the total execution time. The following excerpt from the `pfilestat` DTrace command shows how to sample the time within each system call. By tracing the entry and return from a file system system call, you can observe the total latency as experienced by the application. You could then use probes within the file system to discover where the latency is being incurred.

```
/* sample reads */
 syscall::read:entry,
 syscall::pread*:entry
 /pid == PID && OPT_read/
 {
        runstate = READ;
        @logical["running", (uint64_t)0, ""] = sum(timestamp - last);
        totaltime += timestamp - last;
        last = timestamp;

        self->fd = arg0 + 1;
        self->bytes = arg2;
        totalbytes += arg2;
 }
```

continues

```
fbt::fop_read:entry,
fbt::fop_write:entry
/self->fd/
{
       self->vp = (vnode_t *)arg0;
       self->path = self->vp->v_path == 0 ? "<none>" :
           cleanpath(self->vp->v_path);
}

syscall::read:return,
syscall::pread*:return
/pid == PID && OPT_read/
{
       runstate = OTHER;
       @logical["read", self->fd - 1, self->path] = sum(timestamp - last);
       @bytes["read", self->fd - 1, self->path] = sum(self->bytes);
       totaltime += timestamp - last;
       last = timestamp;
}
```

Using an example target process (`tar`) with `pfilestat`, you can observe that `tar` spends 10% of the time during `read()` calls of `/var/crash/rmcferrari/vmcore.0` and 14% during `write()` calls to `test.tar` out of the total elapsed sample time, and a total of 75% of its time waiting for file system read-level I/O.

```
# ./pfilestat 13092

     STATE   FDNUM      Time Filename
   waitcpu       0        4%
   running       0        9%
      read      11       10% /var/crash/rmcferrari/vmcore.0
     write       3       14% /export/home/rmc/book/examples/test.tar
   sleep-r       0       75%

     STATE   FDNUM      KB/s Filename
      read      11     53776 /var/crash/rmcferrari/vmcore.0
     write       3     53781 /export/home/rmc/book/examples/test.tar

Total event time (ms): 1840   Total Mbytes/sec: 89
```

5.4 Causes of Read/Write File System Latency

There are several causes of latency in the file system read/write data path. The simplest is that of latency incurred by waiting for physical I/O at the backend of the file system. File systems, however, rarely simply pass logical requests straight through to the backend, so latency can be incurred in several other ways. For example, one logical I/O event can be fractured into two physical I/O events, resulting in the latency penalty of two disk operations. Figure 5.3 shows the layers that could contribute latency.

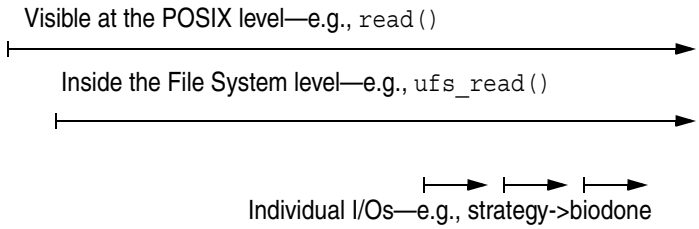

Figure 5.3 Layers for Observing File System I/O

Common sources of latency in the file system stack include:

- Disk I/O wait (or network/filer latency for NFS)
- Block or metadata cache misses
- I/O breakup (logical I/Os being fractured into multiple physical I/Os)
- Locking in the file system
- Metadata updates

5.4.1 Disk I/O Wait

Disk I/O wait is the most commonly assumed type of latency problem. If the underlying storage is in the synchronous path of a file system operation, then it affects file-system-level latency. For each logical operation, there could be zero (a hit in a the block cache), one, or even multiple physical operations.

This `iowait.d` script uses the file name and device arguments in the I/O provider to show us the total latency accumulation for physical I/O operations and the breakdown for each file that initiated the I/O. See Chapter 4 for further information on the I/O provider and Section 10.6.1 for information on its arguments.

```
# ./iowait.d 639
^C
Time breakdown (milliseconds):
 <on cpu>                                                2478
 <I/O wait>                                              6326

I/O wait breakdown (milliseconds):
 file1                                                    236
 file2                                                    241
 file4                                                    244
 file3                                                    264
 file5                                                    277
 file7                                                    330
 ...
```

5.4.2 Block or Metadata Cache Misses

Have you ever heard the saying "the best I/O is the one you avoid"? Basically, the file system tries to cache as much as possible in RAM, to avoid going to disk for repetitive accesses. As discussed in Section 5.6, there are multiple caches in the file system—the most obvious is the data block cache, and others include metadata, inode, and file name caches.

5.4.3 I/O Breakup

I/O breakup occurs when logical I/Os are fractured into multiple physical I/Os. A common file-system-level issue arises when multiple physical I/Os result from a single logical I/O, thereby compounding latency.

Output from running the following DTrace script shows VOP level and physical I/Os for a file system. In this example, we show the output from a single `read()`. Note the many page-sized 8-Kbyte I/Os for the single 1-Mbyte POSIX-level `read()`. In this example, we can see that a single 1-MByte read is broken into several 4-Kbyte, 8-Kbyte, and 56-Kbyte physical I/Os. This is likely due to the file system maximum cluster size (`maxcontig`).

```
# ./fsrw.d
Event            Device RW     Size Offset Path
sc-read               .  R  1048576      0 /var/sadm/install/contents
 fop_read             .  R  1048576      0 /var/sadm/install/contents
   disk_ra       cmdk0  R     4096     72 /var/sadm/install/contents
   disk_ra       cmdk0  R     8192     96 <none>
   disk_ra       cmdk0  R    57344     96 /var/sadm/install/contents
   disk_ra       cmdk0  R    57344    152 /var/sadm/install/contents
   disk_ra       cmdk0  R    57344    208 /var/sadm/install/contents
   disk_ra       cmdk0  R    49152    264 /var/sadm/install/contents
   disk_ra       cmdk0  R    57344    312 /var/sadm/install/contents
   disk_ra       cmdk0  R    57344    368 /var/sadm/install/contents
   disk_ra       cmdk0  R    57344    424 /var/sadm/install/contents
   disk_ra       cmdk0  R    57344    480 /var/sadm/install/contents
   disk_ra       cmdk0  R    57344    536 /var/sadm/install/contents
   disk_ra       cmdk0  R    57344    592 /var/sadm/install/contents
   disk_ra       cmdk0  R    57344    648 /var/sadm/install/contents
   disk_ra       cmdk0  R    57344    704 /var/sadm/install/contents
   disk_ra       cmdk0  R    57344    760 /var/sadm/install/contents
   disk_ra       cmdk0  R    57344    816 /var/sadm/install/contents
   disk_ra       cmdk0  R    57344    872 /var/sadm/install/contents
   disk_ra       cmdk0  R    57344    928 /var/sadm/install/contents
   disk_ra       cmdk0  R    57344    984 /var/sadm/install/contents
   disk_ra       cmdk0  R    57344   1040 /var/sadm/install/contents
```

5.4.4 Locking in the File System

File systems use locks to serialize access within a file (we call these explicit locks) or within critical internal file system structures (implicit locks).

Explicit locks are often used to implement POSIX-level read/write ordering within a file. POSIX requires that writes must be committed to a file in the order in which they are written and that reads must be consistent with the data within the order of any writes. As a simple and cheap solution, many files systems simply implement a per-file reader-writer lock to provide this level of synchronization. Unfortunately, this solution has the unwanted side effect of serializing all accesses within a file, even if they are to non-overlapping regions. The reader-writer lock typically becomes a significant performance overhead when the writes are synchronous (issued with `O_DSYNC` or `O_SYNC`) since the writer-lock is held for the entire duration of the physical I/O (typically, in the order of 10 or more milliseconds), blocking all other reads and writes to the same file.

The POSIX lock is the most significant file system performance issue for databases because they typically use a few large files with hundreds of threads accessing them. If the POSIX lock is in effect, then I/O is serialized, effectively limiting the I/O throughput to that of a single disk. For example, if we assume a file system with 10 disks backing it and a database attempting to write, each I/O will lock a file for 10 ms; the maximum I/O rate is around 100 I/Os per second, even though there are 10 disks capable of 1000 I/Os per second (each disk is capable of 100 I/Os per second).

Most file systems using the standard file system page cache (see Section 14.7 in *Solaris™ Internals*) have this limitation. UFS when used with Direct I/O (see Section 5.6.2) relaxes the per-file reader-writer lock and can be used as a high-performance, uncached file system, suitable for applications such as databases that do their own caching.

5.4.5 Metadata Updates

File system metadata updates are a significant source of latency because many implementations synchronously update the on-disk structures to maintain integrity of the on-disk structures. There are *logical metadata* updates (file creates, deletes, etc.) and *physical metadata* updates (updating a block map, for example).

Many file systems perform several synchronous I/Os per metadata update, which limits metadata performance. Operations such as creating, renaming, and deleting files often exhibit higher latency than reads or writes as a result. Another area affected by metadata updates is file-extends, which can require a physical metadata update.

5.5 Observing File System "Top End" Activity

Applications typically access their data from a file system through the POSIX I/O library and system calls. These accesses are passed into the kernel and into the underlying file system through the VOP layer (see Section 5.1).

Using DTrace function boundary probes, we can trace the VOP layer and monitor file system activity. Probes fired at the entry and exit of each VOP method can record event counts, latency, and physical I/O counts. We can obtain information about the methods by casting the arguments of the VOP methods to the appropriate structures; for example, we can harvest the file name, file system name, I/O size, and the like from these entry points.

The DTrace `vopstat` command instruments and reports on the VOP layer activity. By default, it summarizes each VOP in the system and reports a physical I/O count, a VOP method count, and the total latency incurred for each VOP during the sample period. This utility provides a useful first-pass method of understanding where and to what degree latency is occurring in the file system layer.

The following example shows `vopstat` output for a system running ZFS. In this example, the majority of the latency is being incurred in the `VOP_FSYNC` method (see Table 14.3 in *Solaris™ Internals*).

```
# ./vopstat

VOP Physical IO                                                 Count
fop_fsync                                                         236

VOP Count                                                       Count
fop_create                                                          1
fop_fid                                                             1
fop_lookup                                                          2
fop_access                                                          3
fop_read                                                            3
fop_poll                                                           11
fop_fsync                                                          31
fop_putpage                                                        32
fop_ioctl                                                         115
fop_write                                                         517
fop_rwlock                                                        520
fop_rwunlock                                                      520
fop_inactive                                                      529
fop_getattr                                                      1057

VOP Wall Time                                                 mSeconds
fop_fid                                                             0
fop_access                                                          0
fop_read                                                            0
fop_poll                                                            0
fop_lookup                                                          0
fop_create                                                          0
fop_ioctl                                                           0
fop_putpage                                                         1
fop_rwunlock                                                        1
```

continues

```
fop_rwlock                                                    1
fop_inactive                                                  1
fop_getattr                                                   2
fop_write                                                    22
fop_fsync                                                   504
```

5.6 File System Caches

File systems make extensive use of caches to eliminate physical I/Os where possible. A file system typically uses several different types of cache, including logical metadata caches, physical metadata caches, and block caches. Each file system implementation has its unique set of caches, which are, however, often logically arranged, as shown in Figure 5.4.

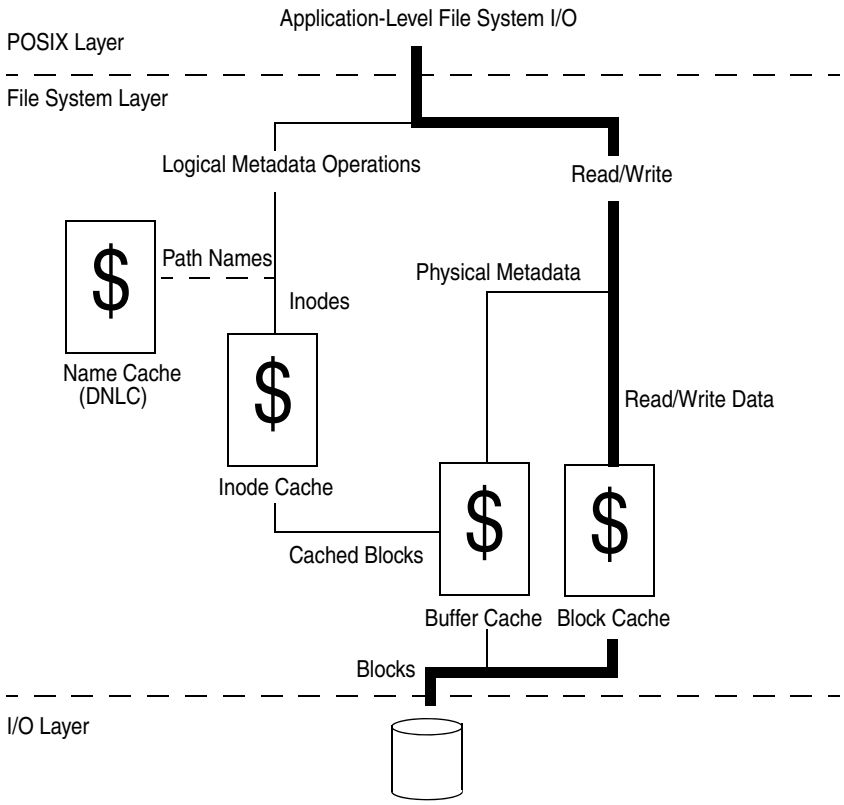

Figure 5.4 File System Caches

The arrangement of caches for various file systems is shown below:

- **UFS.** The file data is cached in a block cache, implemented with the VM system page cache (see Section 14.7 in *Solaris™ Internals*). The physical metadata (information about block placement in the file system structure) is cached in the buffer cache in 512-byte blocks. Logical metadata is cached in the UFS inode cache, which is private to UFS. Vnode-to-path translations are cached in the central directory name lookup cache (DNLC).
- **NFS.** The file data is cached in a block cache, implemented with the VM system *page cache* (see Section 14.7 in *Solaris™ Internals*). The physical metadata (information about block placement in the file system structure) is cached in the *buffer cache* in 512-byte blocks. Logical metadata is cached in the *NFS attribute cache*, and NFS nodes are cached in the *NFS rnode cache*, which are private to NFS. File name-to-path translations are cached in the central DNLC.
- **ZFS.** The file data is cached in ZFS's *adaptive replacement cache* (ARC), rather than in the page cache as is the case for almost all other file systems.

5.6.1 Page Cache

File and directory data for traditional Solaris file systems, including UFS, NFS, and others, are cached in the page cache. The virtual memory system implements a page cache, and the file system uses this facility to cache files. This means that to understand file system caching behavior, we need to look at how the virtual memory system implements the page cache.

The virtual memory system divides physical memory into chunks known as pages; on UltraSPARC systems, a page is 8 kilobytes. To read data from a file into memory, the virtual memory system reads in one page at a time, or "pages in" a file. The page-in operation is initiated in the virtual memory system, which requests the file's file system to page in a page from storage to memory. Every time we read in data from disk to memory, we cause paging to occur. We see the tally when we look at the virtual memory statistics. For example, reading a file will be reflected in `vmstat` as page-ins.

In our example, we can see that by starting a program that does random reads of a file, we cause a number of page-ins to occur, as indicated by the numbers in the `pi` column of `vmstat`.

There is no parameter equivalent to `bufhwm` to limit or control the size of the page cache. The page cache simply grows to consume available free memory. See Section 14.8 in *Solaris™ Internals* for a complete description of how the page cache is managed in Solaris.

```
# ./rreadtest testfile&

# vmstat
  procs     memory            page            disk          faults      cpu
 r b w   swap  free  re  mf  pi  po fr de sr s0 -- -- --   in   sy   cs us sy id
 0 0 0 50436  2064   5   0  81   0  0  0  0 15  0  0  0  168  361   69  1 25 74
 0 0 0 50508  1336  14   0 222   0  0  0  0 35  0  0  0  210  902  130  2 51 47
 0 0 0 50508   648  10   0 177   0  0  0  0 27  0  0  0  168  850  121  1 60 39
 0 0 0 50508   584  29  57  88 109  0  0  6 14  0  0  0  108 5284  120  7 72 20
 0 0 0 50508   484   0  50 249  96  0  0 18 33  0  0  0  199  542  124  0 50 50
 0 0 0 50508   492   0  41 260  70  0  0 56 34  0  0  0  209  649  128  1 49 50
 0 0 0 50508   472   0  58 253 116  0  0 45 33  0  0  0  198  566  122  1 46 53
```

You can use an MDB command to view the size of the page cache. The macro is included with Solaris 9 and later.

```
sol9# mdb -k
Loading modules: [ unix krtld genunix ip ufs_log logindmux ptm cpc sppp ipc random nfs ]
> ::memstat

Page Summary                Pages                MB  %Tot
------------     ----------------  ----------------  ----
Kernel                      53444               208   10%
Anon                       119088               465   23%
Exec and libs                2299                 8    0%
Page cache                  29185               114    6%
Free (cachelist)              347                 1    0%
Free (freelist)            317909              1241   61%

Total                      522272              2040
Physical                   512136              2000
```

The page-cache-related categories are described as follows:

- **Exec and libs.** The amount of memory used for mapped files interpreted as binaries or libraries. This is typically the sum of memory used for user binaries and shared libraries. Technically, this memory is part of the page cache, but it is page-cache-tagged as "executable" when a file is mapped with `PROT_EXEC` and file permissions include execute permission.
- **Page cache.** The amount of unmapped page cache, that is, page cache not on the cache list. This category includes the `segmap` portion of the page cache and any memory mapped files. If the applications on the system are solely using a read/write path, then we would expect the size of this bucket not to exceed `segmap_percent` (defaults to 12% of physical memory size). Files in `/tmp` are also included in this category.
- **Free (cache list).** The amount of page cache on the free list. The free list contains unmapped file pages and is typically where the majority of the file system cache resides. Expect to see a large cache list on a system that has

large file sets and sufficient memory for file caching. Beginning with Solaris 8, the file system cycles its pages through the cache list, preventing it from stealing memory from other applications unless a true memory shortage occurs.

The complete list of categories is described in Section 6.4.3 and further in Section 14.8 in *Solaris™ Internals*.

With DTrace, we now have a method of collecting one of the most significant performance statistics for a file system in Solaris—the *cache hit ratio* in the file system page cache. By using DTrace with probes at the entry and exit to the file system, we can collect the logical I/O events into the file system and physical I/O events from the file system into the device I/O subsystem.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

::fop_read:entry
/self->trace == 0 && (((vnode_t *)arg0)->v_vfsp)->vfs_vnodecovered/
{
        vp = (vnode_t*)arg0;
        vfs = (vfs_t *)vp->v_vfsp;
        mountvp = vfs->vfs_vnodecovered;
        uio = (uio_t*)arg1;
        self->path=stringof(mountvp->v_path);
        @rio[stringof(mountvp->v_path), "logical"] = count();
        @rbytes[stringof(mountvp->v_path), "logical"] = sum(uio->uio_resid);
        self->trace = 1;
}

::fop_read:entry
/self->trace == 0 && (((vnode_t *)arg0)->v_vfsp == `rootvfs)/
{
        vp = (vnode_t*)arg0;
        vfs = (vfs_t *)vp->v_vfsp;
        mountvp = vfs->vfs_vnodecovered;
        uio = (uio_t*)arg1;
        self->path="/";
        @rio[stringof("/"), "logical"] = count();
        @rbytes[stringof("/"), "logical"] = sum(uio->uio_resid);
        self->trace = 1;
}

::fop_read:return
/self->trace == 1/
{
        self->trace = 0;
}

io::bdev_strategy:start
/self->trace/
{
        @rio[self->path, "physical"] = count();
        @rbytes[self->path, "physical"] = sum(args[0]->b_bcount);
}
```

continues

```
tick-5s
{
        trunc (@rio, 20);
        trunc (@rbytes, 20);
        printf("\033[H\033[2J");
        printf ("\nRead IOPS\n");
        printa ("%-60s %10s %10@d\n", @rio);
        printf ("\nRead Bandwidth\n");
        printa ("%-60s %10s %10@d\n", @rbytes);
        trunc (@rbytes);
        trunc (@rio);
}
```

These two statistics give us insight into how effective the file system cache is, and whether adding physical memory could increase the amount of file-system-level caching.

Using this script, we can probe for the number of logical bytes in the file system through the new Solaris 10 file system fop layer. We count the physical bytes by using the `io` provider. Running the script, we can see the number of logical and physical bytes for a file system, and we can use these numbers to calculate the hit ratio.

```
Read IOPS
/data1                                                    physical        287
/data1                                                     logical       2401

Read Bandwidth
/data1                                                    physical    2351104
/data1                                                     logical    5101240
```

The `/data1` file system on this server is doing 2401 logical IOPS and 287 physical—that is, a hit ratio of 2401 ÷ (2401 + 287) = 89%. It is also doing 5.1 Mbytes/sec logical and 2.3 Mbytes/sec physical.

We can also do this at the file level.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

::fop_read:entry
/self->trace == 0 && (((vnode_t *)arg0)->v_path)/
{
        vp = (vnode_t*)arg0;
        uio = (uio_t*)arg1;
        self->path=stringof(vp->v_path);
        self->trace = 1;
        @rio[stringof(vp->v_path), "logical"] = count();
        @rbytes[stringof(vp->v_path), "logical"] = sum(uio->uio_resid);
}
```

continues

```
::fop_read:return
/self->trace == 1/
{
        self->trace = 0;
}

io::bdev_strategy:start
/self->trace/
{
        @rio[self->path, "physical"] = count();
        @rbytes[self->path, "physical"] = sum(args[0]->b_bcount);
}

tick-5s
{
        trunc (@rio, 20);
        trunc (@rbytes, 20);
        printf("\033[H\033[2J");
        printf ("\nRead IOPS\n");
        printa ("%-60s %10s %10@d\n", @rio);
        printf ("\nRead Bandwidth\n");
        printa ("%-60s %10s %10@d\n", @rbytes);
        trunc (@rbytes);
        trunc (@rio);
}
```

5.6.2 Bypassing the Page Cache with Direct I/O

In some cases we may want to do completely unbuffered I/O to a file. A *direct I/O* facility in most file systems allows a direct file read or write to completely bypass the file system page cache. Direct I/O is supported on the following file systems:

- **UFS.** Support for direct I/O was added to UFS starting with Solaris 2.6. Direct I/O allows reads and writes to files in a regular file system to bypass the page cache and access the file at near raw disk performance. Direct I/O can be advantageous when you are accessing a file in a manner where caching is of no benefit. For example, if you are copying a very large file from one disk to another, then it is likely that the file will not fit in memory and you will just cause the system to page heavily. By using direct I/O, you can copy the file through the file system without reading through the page cache and thereby eliminate both the memory pressure caused by the file system and the additional CPU cost of the layers of cache.

  Direct I/O also eliminates the double copy that is performed when the `read` and `write` system calls are used. When we read a file through normal buffered I/O, the file system takes two steps: (1) It uses a DMA transfer from the disk controller into the kernel's address space and (2) it copies the data into the buffer supplied by the user in the `read` system call. Direct I/O eliminates the second step by arranging for the DMA transfer to occur directly into the user's address space.

Direct I/O bypasses the buffer cache only if all the following are true:

- The file is not memory mapped.
- The file does not have holes.
- The read/write is sector aligned (512 byte).

- **QFS.** Support for direct I/O is the same as with UFS.
- **NFS.** NFS also supports direct I/O. With direct I/O enabled, NFS bypasses client-side caching and passes all requests directly to the NFS server. Both reads and writes are uncached and become synchronous (they need to wait for the server to complete). Unlike disk-based direct I/O support, NFS's support imposes no restrictions on I/O size or alignment; all requests are made directly to the server.

You enable direct I/O by mounting an entire file system with the `forcedirectio` `mount` option, as shown below.

```
# mount -o forcedirectio /dev/dsk/c0t0d0s6 /u1
```

You can also enable direct I/O for any file with the `directio` system call. Note that the change is file based, and every reader and writer of the file will be forced to use `directio` once it's enabled.

```
int directio(int fildes, DIRECTIO_ON | DIRECTIO_OFF);
                                                    See sys/fcntl.h
```

Direct I/O can provide extremely fast transfers when moving data with big block sizes (>64 kilobytes), but it can be a significant performance limitation for smaller sizes. If an application reads and writes in small sizes, then its performance may suffer since there is no read-ahead or write clustering and no caching.

Databases are a good candidate for direct I/O since they cache their own blocks in a shared global buffer and can cluster their own reads and writes into larger operations.

A set of direct I/O statistics is provided with the `ufs` implementation by means of the `kstat` interface. The structure exported by `ufs_directio_kstats` is shown below. Note that this structure may change, and performance tools should not rely on the format of the direct I/O statistics.

```
struct ufs_directio_kstats {
        uint_t  logical_reads;  /* Number of fs read operations */
        uint_t  phys_reads;     /* Number of physical reads */
        uint_t  hole_reads;     /* Number of reads from holes */
        uint_t  nread;          /* Physical bytes read */
        uint_t  logical_writes; /* Number of fs write operations */
        uint_t  phys_writes;    /* Number of physical writes */
        uint_t  nwritten;       /* Physical bytes written */
        uint_t  nflushes;       /* Number of times cache was cleared */
} ufs_directio_kstats;
```

You can inspect the direct I/O statistics with a utility from our Web site at `http://www.solarisinternals.com`.

```
# directiostat 3
  lreads lwrites  preads pwrites     Krd     Kwr holdrds  nflush
       0       0       0       0       0       0       0       0
       0       0       0       0       0       0       0       0
       0       0       0       0       0       0       0       0
```

5.6.3 The Directory Name Lookup Cache

The directory name cache caches path names for `vnodes`, so when we open a file that has been opened recently, we don't need to rescan the directory to find the file name. Each time we find the path name for a `vnode`, we store it in the directory name cache. (See Section 14.10 in *Solaris™ Internals* for further information on the DNLC operation.) The number of entries in the DNLC is set by the system-tuneable parameter, `ncsize`, which is set at boot time by the calculations shown in Table 5.1. The `ncsize` parameter is calculated in proportion to the `maxusers` parameter, which is equal to the number of megabytes of memory installed in the system, capped by a maximum of 1024. The `maxusers` parameter can also be overridden in `/etc/system` to a maximum of 2048.

Table 5.1 DNLC Default Sizes

Solaris Version	Default ncsize Calculation
Solaris 2.4, 2.5, 2.5.1	`ncsize` = (17 * `maxusers`) + 90
Solaris 2.6 onwards	`ncsize` = (68 * `maxusers`) + 360

The size of the DNLC rarely needs to be adjusted, because the size scales with the amount of memory installed in the system. Earlier Solaris versions had a

default maximum of 17498 (34906 with `maxusers` set to 2048), and later Solaris versions have a maximum of 69992 (139624 with `maxusers` set to 2048).

Use MDB to determine the size of the DNLC.

```
# mdb -k
> ncsize/D
ncsize:
ncsize:         25520
```

The DNLC maintains housekeeping threads through a task queue. The `dnlc_reduce_cache()` activates the task queue when name cache entries reach `ncsize`, and it reduces the size to `dnlc_nentries_low_water`, which by default is one hundredth less than (or 99% of) `ncsize`. If `dnlc_nentries` reaches `dnlc_max_nentries` (twice `ncsize`), then we know that `dnlc_reduce_cache()` is failing to keep up. In this case, we refuse to add new entries to the `dnlc` until the task queue catches up. Below is an example of DNLC statistics obtained with the `kstat` command.

```
# vmstat -s
        0 swap ins
        0 swap outs
        0 pages swapped in
        0 pages swapped out
   405332 total address trans. faults taken
  1015894 page ins
      353 page outs
  4156331 pages paged in
     1579 pages paged out
  3600535 total reclaims
  3600510 reclaims from free list
        0 micro (hat) faults
   405332 minor (as) faults
   645073 major faults
    85298 copy-on-write faults
   117161 zero fill page faults
        0 pages examined by the clock daemon
        0 revolutions of the clock hand
  4492478 pages freed by the clock daemon
     3205 forks
       88 vforks
     3203 execs
 33830316 cpu context switches
 58808541 device interrupts
   928719 traps
214191600 system calls
 14408382 total name lookups (cache hits 90%)
   263756 user   cpu
   462843 system cpu
 14728521 idle   cpu
  2335699 wait   cpu
```

The hit ratio of the directory name cache shows the number of times a name was looked up and found in the name cache. A high hit ratio (>90%) typically shows that the DNLC is working well. A low hit ratio does not necessarily mean that the DNLC is undersized; it simply means that we are not always finding the names we want in the name cache. This situation can occur if we are creating a large number of files. The reason is that a `create` operation checks to see if a file exists before it creates the file, causing a large number of cache misses.

The DNLC statistics are also available with `kstat`.

```
$ kstat -n dnlcstats
module: unix                            instance: 0
name:   dnlcstats                       class:    misc
        crtime                          208.832373709
        dir_add_abort                   0
        dir_add_max                     0
        dir_add_no_memory               0
        dir_cached_current              1
        dir_cached_total                13
        dir_entries_cached_current      880
        dir_fini_purge                  0
        dir_hits                        463
        dir_misses                      11240
        dir_reclaim_any                 8
        dir_reclaim_last                3
        dir_remove_entry_fail           0
        dir_remove_space_fail           0
        dir_start_no_memory             0
        dir_update_fail                 0
        double_enters                   6
        enters                          11618
        hits                            1347693
        misses                          10787
        negative_cache_hits             76686
        pick_free                       0
        pick_heuristic                  0
        pick_last                       0
        purge_all                       1
        purge_fs1                       0
        purge_total_entries             3013
        purge_vfs                       158
        purge_vp                        31
        snaptime                        94467.490008162
```

5.6.4 Block Buffer Cache

The buffer cache used in Solaris for caching of inodes and file metadata is now also dynamically sized. In old versions of UNIX, the buffer cache was fixed in size by the `nbuf` kernel parameter, which specified the number of 512-byte buffers. We now allow the buffer cache to grow by `nbuf`, as needed, until it reaches a ceiling

specified by the bufhwm kernel parameter. By default, the buffer cache is allowed to grow until it uses 2% of physical memory. We can look at the upper limit for the buffer cache by using the sysdef command.

```
# sysdef
*
* Tunable Parameters
*
 7757824        maximum memory allowed in buffer cache (bufhwm)
    5930        maximum number of processes (v.v_proc)
      99        maximum global priority in sys class (MAXCLSYSPRI)
    5925        maximum processes per user id (v.v_maxup)
      30        auto update time limit in seconds (NAUTOUP)
      25        page stealing low water mark (GPGSLO)
       5        fsflush run rate (FSFLUSHR)
      25        minimum resident memory for avoiding deadlock (MINARMEM)
      25        minimum swapable memory for avoiding deadlock (MINASMEM)
```

Now that we only keep inode and metadata in the buffer cache, we don't need a very large buffer. In fact, we need only 300 bytes per inode and about 1 megabyte per 2 gigabytes of files that we expect to be accessed concurrently (note that this rule of thumb is for UFS file systems).

For example, if we have a database system with 100 files totaling 100 gigabytes of storage space and we estimate that we will access only 50 gigabytes of those files at the same time, then at most we would need 100 × 300 bytes = 30 kilobytes for the inodes and about 50 ÷ 2 × 1 megabyte = 25 megabytes for the metadata (direct and indirect blocks). On a system with 5 gigabytes of physical memory, the defaults for bufhwm would provide us with a bufhwm of 102 megabytes, which is more than sufficient for the buffer cache. If we are really memory misers, we could limit bufhwm to 30 megabytes (specified in kilobytes) by setting the bufhwm parameter in the /etc/system file. To set bufhwm smaller for this example, we would put the following line into the /etc/system file.

```
*
* Limit size of bufhwm
*
set bufhwm=30000
```

You can monitor the buffer cache hit statistics by using sar -b. The statistics for the buffer cache show the number of logical reads and writes into the buffer cache, the number of physical reads and writes out of the buffer cache, and the read/write hit ratios.

```
# sar -b 3 333
SunOS zangief 5.7 Generic sun4u    06/27/99

22:01:51 bread/s lread/s %rcache bwrit/s lwrit/s %wcache pread/s pwrit/s
22:01:54       0    7118     100       0       0     100       0       0
22:01:57       0    7863     100       0       0     100       0       0
22:02:00       0    7931     100       0       0     100       0       0
22:02:03       0    7736     100       0       0     100       0       0
22:02:06       0    7643     100       0       0     100       0       0
22:02:09       0    7165     100       0       0     100       0       0
22:02:12       0    6306     100       8      25      68       0       0
22:02:15       0    8152     100       0       0     100       0       0
22:02:18       0    7893     100       0       0     100       0       0
```

On this system we can see that the buffer cache is caching 100% of the reads and that the number of writes is small. This measurement was taken on a machine with 100 gigabytes of files that were being read in a random pattern. You should aim for a read cache hit ratio of 100% on systems with only a few, but very large, files (for example, database systems) and a hit ratio of 90% or better for systems with many files.

5.6.5 UFS Inode Cache

The UFS uses the `ufs_ninode` parameter to size the file system tables for the expected number of inodes. To understand how the `ufs_ninode` parameter affects the number of inodes in memory, we need to look at how the UFS maintains inodes. Inodes are created when a file is first referenced. They remain in memory much longer than when the file is last referenced because inodes can be in one of two states: either the inode is referenced or the inode is no longer referenced but is on an idle queue. Inodes are eventually destroyed when they are pushed off the end of the inode idle queue. Refer to Section 15.3.2 in *Solaris™ Internals* for a description of how `ufs` inodes are maintained on the idle queue.

The number of inodes in memory is dynamic. Inodes will continue to be allocated as new files are referenced. There is no upper bound to the number of inodes open at a time; if one million inodes are opened concurrently, then a little over one million inodes will be in memory at that point. A file is referenced when its reference count is non-zero, which means that either the file is open for a process or another subsystem such as the directory name lookup cache is referring to the file.

When inodes are no longer referenced (the file is closed and no other subsystem is referring to the file), the inode is placed on the idle queue and eventually freed. The size of the idle queue is controlled by the `ufs_ninode` parameter and is limited to one-fourth of `ufs_ninode`. The maximum number of inodes in memory at a given point is the number of active referenced inodes plus the size of the idle queue (typically, one-fourth of `ufs_ninode`). Figure 5.5 illustrates the `inode` cache.

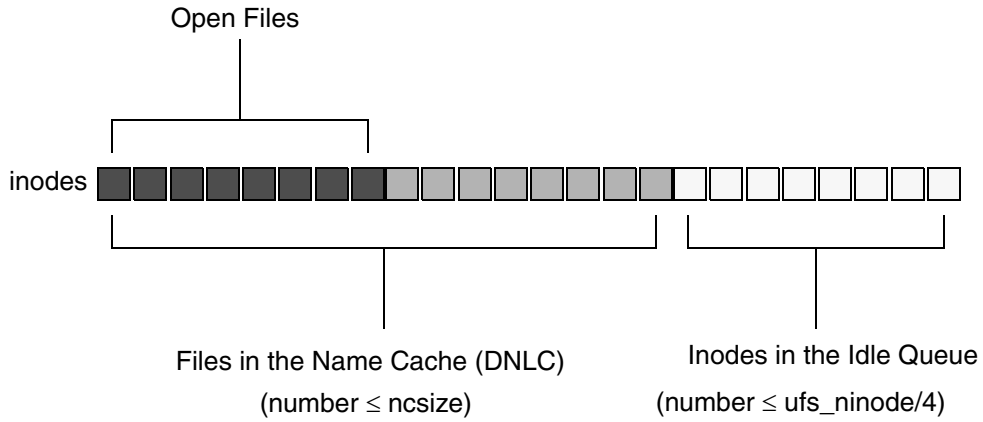

Figure 5.5 In-Memory Inodes (Referred to as the "Inode Cache")

We can use the `sar` command and inode kernel memory statistics to determine the number of inodes currently in memory. `sar` shows us the number of inodes currently in memory and the number of `inode` structures in the `inode` slab cache. We can find similar information by looking at the `buf_inuse` and `buf_total` parameters in the inode kernel memory statistics.

```
# sar -v 3 3

SunOS devhome 5.7 Generic sun4u    08/01/99

11:38:09  proc-sz    ov  inod-sz    ov  file-sz    ov   lock-sz
11:38:12  100/5930    0 37181/37181    0  603/603     0    0/0
11:38:15  100/5930    0 37181/37181    0  603/603     0    0/0
11:38:18  101/5930    0 37181/37181    0  607/607     0    0/0

# kstat -n ufs_inode_cache
ufs_inode_cache:
buf_size 440 align 8 chunk_size 440 slab_size 8192 alloc 1221573 alloc_fail 0
free 1188468 depot_alloc 19957 depot_free 21230 depot_contention 18 global_alloc 48330
global_free 7823 buf_constructed 3325 buf_avail 3678 buf_inuse 37182
buf_total 40860 buf_max 40860 slab_create 2270 slab_destroy 0 memory_class 0
hash_size 0 hash_lookup_depth 0 hash_rescale 0 full_magazines 219
empty_magazines 332 magazine_size 15 alloc_from_cpu0 579706 free_to_cpu0 588106
buf_avail_cpu0 15 alloc_from_cpu1 573580 free_to_cpu1 571309 buf_avail_cpu1 25
```

The inode memory statistics show us how many inodes are allocated by the `buf_inuse` field. We can also see from the `ufs` inode memory statistics that the size of each inode is 440 bytes on this system See below to find out the size of an inode on different architectures.

```
# mdb -k
Loading modules: [ unix krtld genunix specfs dtrace ...]
> a$d
radix = 10 base ten
> ::sizeof inode_t
sizeof (inode_t) = 0t276
> $q

$ kstat unix::ufs_inode_cache:chunk_size
module: unix                            instance: 0
name:   ufs_inode_cache                 class:    kmem_cache
        chunk_size                      280
```

We can use this value to calculate the amount of kernel memory required for desired number of inodes when setting `ufs_ninode` and the directory name cache size.

The `ufs_ninode` parameter controls the size of the hash table that is used for inode lookup and indirectly sizes the inode idle queue (`ufs_ninode ÷ 4`). The inode hash table is ideally sized to match the total number of inodes expected to be in memory—a number that is influenced by the size of the directory name cache. By default, `ufs_ninode` is set to the size of the directory name cache, which is approximately the correct size for the inode hash table. In an ideal world, we could set `ufs_ninode` to four-thirds the size of the DNLC, to take into account the size of the idle queue, but practice has shown this to be unnecessary.

We typically set `ufs_ninode` indirectly by setting the directory name cache size (`ncsize`) to the expected number of files accessed concurrently, but it is possible to set `ufs_ninode` separately in `/etc/system`.

```
* Set number of inodes stored in UFS inode cache
*
set ufs_ninode = new_value
```

5.6.6 Monitoring UFS Caches with `fcachestat`

We can monitor all four key UFS caches by using a single Perl tool: `fcachestat`. This tool measures the DNLC, inode, UFS buffer cache (metadata), and page cache by means of `segmap`.

```
$ ./fcachestat 5
  --- dnlc ---     -- inode ---     -- ufsbuf --     -- segmap --
 %hit    total    %hit    total    %hit    total    %hit    total
99.64   693.4M   59.46     4.9M   99.80    94.0M   81.39   118.6M
66.84    15772   28.30     6371   98.44     3472   82.97     9529
63.72    27624   21.13    12482   98.37     7435   74.70    14699
10.79    14874    5.64    16980   98.45    12349   93.44    11984
11.96    13312   11.89    14881   98.37    10004   93.53    10478
 4.08    20139    5.71    25152   98.42    17917   97.47    16729
 8.25    17171    3.57    20737   98.38    15054   93.64    11154
15.40    12151    6.89    13393   98.37     9403   93.14    11941
 8.26     9047    4.51    10899   98.26     7861   94.70     7186
66.67        6    0.00        3   95.45       44   44.44       18
```

5.7 NFS Statistics

The NFS client and server are instrumented so that they can be observed with `iostat` and `nfsstat`. For client-side mounts, `iostat` reports the latency for read and write operations per mount, and instead of reporting disk response times, `iostat` reports NFS server response times (including over-the-write latency). The `-c` and `-s` options of the `nfsstat` command reports both client- and server-side statistics for each NFS operation as specified in the NFS protocol.

5.7.1 NFS Client Statistics: `nfsstat -c`

The client-side statistics show the number of calls for RPC transport, virtual metadata (also described as attributes), and read/write operations. The statistics are separated by NFS version number (currently 2, 3, and 4) and protocol options (TCP or UDP).

```
$ nfsstat -c

Client rpc:
Connection oriented:
calls      badcalls   badxids    timeouts   newcreds   badverfs   timers
202499     0          0          0          0          0          0
cantconn   nomem      interrupts
0          0          0
Connectionless:
calls      badcalls   retrans    badxids    timeouts   newcreds   badverfs
0          0          0          0          0          0          0
timers     nomem      cantsend
0          0          0

Client nfs:
calls     badcalls  clgets    cltoomany
200657    0         200657    7
Version 2: (0 calls)
```

continues

```
null     getattr  setattr  root     lookup   readlink read     wrcache
0 0%     0 0%     0 0%     0 0%     0 0%     0 0%     0 0%     0 0%
write    create   remove   rename   link     symlink  mkdir    rmdir
0 0%     0 0%     0 0%     0 0%     0 0%     0 0%     0 0%     0 0%
readdir  statfs
0 0%     0 0%
Version 3: (0 calls)
null         getattr      setattr      lookup       access       readlink
0 0%         0 0%         0 0%         0 0%         0 0%         0 0%
read         write        create       mkdir        symlink      mknod
0 0%         0 0%         0 0%         0 0%         0 0%         0 0%
remove       rmdir        rename       link         readdir      readdirplus
0 0%         0 0%         0 0%         0 0%         0 0%         0 0%
fsstat       fsinfo       pathconf     commit
0 0%         0 0%         0 0%         0 0%
```

5.7.2 NFS Server Statistics: `nfsstat -s`

The NFS server-side statistics show the NFS operations performed by the NFS server.

```
$ nfsstat -s

Server rpc:
Connection oriented:
calls      badcalls   nullrecv   badlen     xdrcall    dupchecks  dupreqs
5897288    0          0          0          0          372803     0
Connectionless:
calls      badcalls   nullrecv   badlen     xdrcall    dupchecks  dupreqs
87324      0          0          0          0          0          0

...

Version 4: (949163 calls)
null                compound
3175 0%             945988 99%
Version 4: (3284515 operations)
reserved            access              close               commit
0 0%                72954 2%            199208 6%           2948 0%
create              delegpurge          delegreturn         getattr
4 0%                0 0%                16451 0%            734376 22%
getfh               link                lock                lockt
345041 10%          6 0%                101 0%              0 0%
locku               lookup              lookupp             nverify
101 0%              145651 4%           5715 0%             171515 5%
open                openattr            open_confirm        open_downgrade
199410 6%           0 0%                271 0%              0 0%
putfh               putpubfh            putrootfh           read
914825 27%          0 0%                581 0%              130451 3%
readdir             readlink            remove              rename
5661 0%             11905 0%            15 0%               201 0%
renew               restorefh           savefh              secinfo
30765 0%            140543 4%           146336 4%           277 0%
setattr             setclientid         setclientid_confirm verify
23 0%               26 0%               26 0%               10 0%
write               release_lockowner   illegal
9118 0%             0 0%                0 0%
...
```