SunOS 4.1 Performance Tuning

Hal L. Stern
Sun Microsystems
Northeast Area Consulting Group
Lincoln, MA
halstern@sun.com

SunOS 4.1 includes a number of features designed to improve performance on both a per-application and system-wide basis. Features such as the *tmpfs* swap-based filesystem, NFS dynamic retransmission, *mmap*() and the associated *madvise*() hinting algorithm and dynamic allocation of kernel data structures may degrade performance if ignored or misconfigured. This paper presents performance analysis procedures and suggested policies for optimizing use of system facilities under normal conditions and avoiding severe performance degradation under peak work loads.

1. Introduction

Optimizing computing system performance requires efforts by both users and system administrators. Users expect system services to be well conditioned and to gracefully handle periods of peak demand; system managers rely on the user community to make efficient use of local and networked resources. Tuning the SunOS operating system does not require modifying the kernel source code, but instead entails establishing guidelines for system configuration and application development.

1.1. Performance Tuning in the Distributed Environment

Distributed computing systems add to the complexity of performance tuning. The user's perception of system response time is dependent upon the composition of each of the system's components: The code being executed, the local machine's configuration, the network, and the performance of central servers.

System managers adopt a *global* view: performance tuning is geared toward meeting the aggregate demands placed on networks and servers. The global goal is to ensure acceptable response time during peak usage periods. Procedures used to achieve this goal involve varying configurations on both server and client machines. Increasing the size of kernel data structures on a server may provide part of a solution, while changing the NFS mounting schemes on the client side complements this process.

Users have a more *local* view of the tuning process. From a desktop workstation or terminal, the network and servers are generally of little interest to the user, particularly if network services are provided transparently. In an environment where all user files reside on NFS servers, the physical location of any file is hidden behind the NFS mount point scheme. A user will only become concerned about the server and filesystem containing his or her files when its response time exceeds the user's expectations. From the local viewpoint, the goal of performance tuning is to improve execution of a particular mixture of applications, without specific regard for the system infrastructure on which they rely.

The two views of performance tuning are quite dependent: aggregate user demands establish the requirements for the global tuning process, and these very demands may be reduced through careful tuning of applications.

1.2. Measuring Success and Setting Expectations

The primary goal of the performance tuning process should be to ensure system behavior that meets user expectations and that is consistent over a wide variety of conditions. A secondary goal is to identify the bottlenecks in the current hardware to guide incremental purchases of new equipment. Assumptions about linear increases in performance generated by additional hardware are generally false; doubling the amount of memory on a workstation will not double its performance if the typical application is disk-bound.

When evaluating the performance of current hardware or the impact of performance-oriented changes, it is imperative to measure either the quantity of interest or an appropriate subset of the real environment. Gauging the routing capacity of a machine is not a job for `ftp`, since the disk bandwidth of the receiving host will limit the number of packets per second that are transmitted. Similarly, execution of an application with a 16 Mbyte data set cannot be properly evaluated using benchmarks with a 4 Mbyte data set, since the real application may be bound by the paging efficiency of the system in use. Scaled-down benchmarks may be colored by cache effects.

A final caveat about performance tuning is that it is subject to a law of diminishing returns. Correcting poor configurations will have an immediate effect on overall performance driven by a finite investment of system administration resources. Extracting the last few percent out of a disk subsystem, however, may entail substantial analysis and experimentation, and simply not justify the investment of time. Furthermore, a system that is brought close to perfection for the current environment may not perform equally well if a user's files are moved to another disk or if new diskless workstations are added to the network. Computing environments are rarely static, but a well-tuned system should easily incorporate minor changes in its configuration or usage.

1.3. Tuning Suns and SunOS 4.1

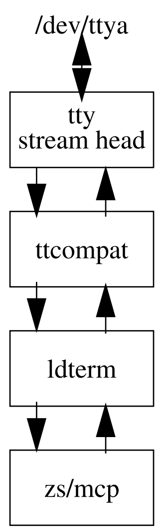
This paper presents procedures that can be applied to servers and to desktop machines. Areas of concern include STREAMS and network interfaces, IP routing, NFS strategies, memory management and efficient file I/O. Emphasis is placed on SunOS features as well as general Sun hardware-specific performance issues such as cache utilization. Its intent is to define part of a pro-active approach to system performance tuning. By anticipating common problems the system manager can reduce the amount of time spent resolving crises and listening to harsh words from users.

SunOS 4.1 introduced several performance-oriented features, including dynamic allocation of STREAMS structures, NFS dynamic retransmission, the *tmpfs* swap-based filesystem, asynchronous I/O and virtual memory management hinting through *madvise*(). Building a custom kernel now involves compiling several configuration files with user-tunable parameters. Ignoring or improperly configuring these services may degrade performance. In other cases, optimizations of kernel services hide flaws in applications that would have previously attracted the attention of the system administrator.

2. STREAMS System

The SunOS 4.1 *tty* and pseudo-tty (*ptty*) device drivers are STREAMS based, derived from the AT&T System V Release 3 data communications mechanism. STREAMS drivers comprise stacks of modules. Each module has a well-defined interface that receives data from lower-level modules, processes it, and passes it up to a higher-level module. Modules exchange data in messages, containing any amount of data from a single character input to a complete 8192 byte network packet. Data coming into the system is passed upstream from the external device driver, through the modules, to the stream head. Similarly, output goes through the stream head and down through the modules to some physical device.

This discussion will focus on physical terminal input, either through the CPU serial ports or an ALM-2 multiplexor. The SunOS terminal driver has are two modules: `ttcompat` module for Berkeley-style backward compatibility, and `ldterm` for **termio** input processing:



STREAMS input has a large impact on system performance because of the high priority at which it is handled and the amount of processing that is done on each input character. Asynchronous communications controllers have a relatively small hardware input buffer, requiring them to be serviced almost immediately. Serial input from the **zs** or **mcp** device drivers is handled at the highest device priority - higher than the hardware clock interrupt that drives the time-of-day counter. Overruns of serial inputs will degrade performance as SunOS must defer servicing of lower level interrupts. If ethernet driver interrupts are consistently missed, for example, ethernet packets will be dropped at the local network interface. When hardware clock interrupts are missed, the SunOS kernel incurs a penalty for time-of-day adjustment during succeeding hardclock interrupts.

Once characters are in the physical device driver, they are bundled into STREAMS messages and passed through all modules for input processing. Normal input processing such as handling newlines, character echo, erase and kill functions, and signal generation, is a non-trivial task performed a character at a time. Processing sustained bursts of character inputs can easily consume a large percentage of the system's CPU cycles. A pair of modems that attempt to communicate but cannot agree on parity and error correcting protocols, for example, will slow the system significantly each time **uucp** attempts to negotiate a connection but encounters invalid input filled with parity errors.

SunOS 4.1 dynamically allocates STREAMS data structures, leaving little room for tuning within the kernel. Optimizing performance of the STREAMS input system is essentially a user-level issue, and will impact the performance of interactive applications, terminal user servers, print servers and hosts with modem pools.

2.1. Streams Buffer Allocation

Each STREAMS message consists of a message header and one or more data blocks. The data blocks are the interesting part; message headers are used to chain messages together. Prior to SunOS 4.1, STREAMS data structures were allocated from a fixed-size pool of fixed-size blocks. The smallest block that could contain the message was allocated, and if no blocks of the best-fit size were available, the next largest, available size would be used. Under extreme conditions, it was possible to run out of STREAMS data blocks, deadlocking all terminal-based input devices.

SunOS 4.1 includes dynamic allocation of STREAMS data blocks, with small blocks cached in the message header itself. By eliminating the overhead of small block allocation and the memory waste due to fixed-size blocks, the STREAMS input system has become more efficient. However, it is also much more tolerant of ill-behaved input devices, and will allocate as many data blocks as possible to accommodate a noisy terminal line generating garbage, or a non-flow controlled input device. Symptoms of this problem include a marked increase in system CPU utilization, as shown by `vmstat`:

```
% vmstat 10
.... us sy id
..... 0 21 79
..... 1 18 82
```

Noisy terminal lines, or those that are not connected but still have a `getty` running on them, should be shielded or turned off.

2.2. STREAMS Buffer Statistics

`netstat -m` displays kernel memory allocation statistics for networking and STREAMS systems. The lower half of its output contains a histogram of data block sizes and allocation counts:

```
[sunne]% netstat -m
... mbuf and streams allocation statistics ...
streams buffers:
                 cur   max  total failed
external           0     0        0    0
within-dblk        1    55   721792    0
size <= 16         3    18    48731    0
size <= 32         5    21    73449    0
size <= 128       50   174   819435    0
size <= 512        0     4  1257404    0
size <= 1024       0    12   263686    0
size <= 2048       0     2   611607    0
size <= 8192       0     1    62864    0
size  > 8192       0     0        0    0
```

The *within-dblk* count shows the number of times the data block was able to be cached instead of allocated from the kernel's heap. The statistics are incremented for every allocation, and in active systems some benefit may be derived from increasing the granularity of the counters.

The kernel configuration file `/sys/conf.common/param.c` contains an array of the bucket sizes used to compile allocation statistics:

```
uint  strbufsizes[] = { 16, 32, 128, 512, 1024, 2048,
                        8192, 0 };
```

Each allocation compares the size of the block requested with each value in this array to find the appropriate counter. If there are fewer counters, the overhead of keeping the statistics is reduced. In the example above, statistics may be kept using only two buckets, choosing the bucket sizes to match most of the allocations with one comparison and all of them within two:

```
uint  strbufsizes[] = { 128, 8192, 0 };
```

2.3. Input Notification

There are several mechanisms for notification of terminal line input and they differ greatly in their relative efficiency and functionality. The most common requirements for character input notification are:

[1] Arrival of any character, with the application blocked during this time. This is the behavior of *read*() on a tty port; it blocks until a character is placed into the tty input queue.

[2] Arrival of any character, with the application performing other processing. Window-based applications frequently require this input notification so they may update other portions of the display while waiting for input. Because it is essentially a form of polled input, this approach can be implemented in a variety of bad ways. SunOS 4.1 has an asynchronous I/O facility to implement polled-type I/O without the overhead of continuous system calls.

[3] Arrival of at least *n* characters. The application may use a technique similar to [1] or [2] to get individual characters and bundle them into a larger buffer. Alternatively, the `FIONREAD` *ioctl*() call will return the number of characters waiting to be read, but it too must be called repeatedly until the notification condition is met.

An application that is polling for serial line input or building up input buffers from single character inputs can be rewritten to use the **termio** and asynchronous I/O facilities in SunOS 4.1, greatly reducing the number of system calls made by the application and the amount of system CPU time required for its execution.

Asynchronous I/O Notification

The asynchronous I/O mechanism in SunOS 4.1 is built on a lightweight process system in the kernel. Any open file descriptor that can be passed to *read*() can be handed to *aioread*() as well. When the asynchronous read completes, the user's process is sent a `SIGIO` signal and a user-supplied completion buffer is filled in with the *read*() return code. Asynchronous I/O should be used when an application must multiplex input events and other processing activity.

Using *select*(), an application can poll for input while performing other tasks:

```
tfd = open("/dev/ttyh0", O_RDONLY);
while (...) {
      FD_SET(tfd, &readmask);
      select(fdwidth, &readmask, NULL, NULL, NULL);
      if (FD_ISSET(tfd, &readmask)) {
            ..handle character input...
      }
      ...other processing...
}
```

Each trip through the loop generates a system call; the entire construct is overly CPU intensive. A non-NULL timeout may be passed to *select*(), although this will cause the remainder of the processing to occur sporadically. In an interactive application, the effect is disturbing to the user.

Using asynchronous I/O, the same code can be rewritten:

```
int got_chars();
tfd = open("/dev/ttyh0", O_RDONLY);
signal(SIGIO, got_chars);
aioread(tfd, charbuf, sizeof(charbuf), 0, 0, &aio_res);

while (...) {
      ...other processing...
}

int got_chars() {
      ...handle character input...
}
```

Asynchronous notification is useful when reading from a device other than the keyboard in a windows-based application since keyboard events are delivered asynchronously by the window system server.

Non-Canonical Mode Input Processing

Some applications will only process terminal input after *n* characters have been received. The application may issue a series of blocking *read*() calls, building up a buffer on its own, or it can use the `FIONREAD` *ioctl*() to determine how many characters are in the input queue before issuing a single *read*(). Polling for input using `FIONREAD` is more inefficient than using *select*() because the *ioctl*() must walk through all messages in the stream head, counting the number of characters in each message, while *select*() simply checks for the existence of input in the stream head.

The System V **termio** mechanism provides a robust input facility that replaces many *select*()-based processing loops and handles bursts of serial line input without significant kernel overhead. To disable the default character at a time behavior of the tty driver, canonical mode input processing is turned off by resetting the `ICANON` bit in the local mode `(c_lflag)` field of the **termio** structure. When canonical input processing is turned off, no erase or kill editing is performed. Characters are passed back through *read*() depending upon the settings of the `MIN` and `TIME` fields in the **termio** structure. Although there are four combinations of zero and non-zero values for `MIN` and `TIME`, only the case where `MIN` is non-zero and `TIME = 0` is applicable to this discussion.

When a value is specified for `MIN`, it becomes the input threshold for calls to *read*(). Any attempt to read from the terminal device will block until `MIN` characters have been received. Non-canonical input mode efficiently handles bursts of input characters because characters are assembled in the stream head, and there is little input processing performed on them. Applications should use this feature to replace polling loops; asynchronous reads can be used if the application cannot block waiting for the character input.

3. Networking

A common constraint in a distributed computing environment is network bandwidth. Ethernet utilization is normally measured on a logarithmic scale because its efficiency decreases exponentially as bandwidth is consumed. Tuning the SunOS network interfaces and planning effective routing policies will help to reduce demands placed on the network; when these techniques are coupled with NFS mounting strategies and optimized application file handling, significant capacity can be recovered from an overloaded network.

3.1. Protocol and Network Interface Configuration

Before modifying protocol and network interface buffers, an understanding of how ethernet packets are received is required.

- When the ethernet chip receives a packet, the data is copied into an ethernet receive buffer. There are a fixed number of receive buffers, and if the ethernet device driver runs out, it may drop input packets. When the packet has been copied into the buffer, the ethernet chip interrupts and initiates the operating system's processing of the packet.

- The received packet is wrapped in a message buffer, or **mbuf**, and passed to the IP layer where its destination IP address and higher-level protocol are examined.
- If the packet does not need to be forwarded, the IP layer queues it in a per-socket TCP or UDP protocol buffer. If the protocol's buffer is full, the packet is dropped.

The TCP and UDP protocols schedule a fast and slow handlers to process input packets and pass data to higher protocol layers. Fast handlers typically run every 200 usec and slow handlers every 500 usec. Waiting for the socket's input queue to be populated makes the UDP and TCP processing more efficient, as several IP packets can be concatenated into a larger datagram in a single effort.

Protocol Receive Spaces

SunOS 4.1 exposes several network interface and protocol configuration variables in user-accessible files that are compiled as part of a normal (object-only) kernel build. In the file `/sys/netinet/in_proto.c` are the default sizes for each protocol's send and receive buffers.

The per-socket UDP receive space is set by the value of `udp_recvspace`. Socket overflows experienced when this receive space is exhausted are reported by `netstat -s`:

```
[sunne]% netstat -s
udp:
      0 incomplete headers
      0 bad data length fields
      0 bad checksums
      75 socket overflows
```

Changing the default size of the UDP receive space may eliminate socket overflows reported by `netstat -s`, but it probably masks the real root of the problem. If the host is routinely running out of room in a UDP socket structure, then it is either being subjected to unusually high levels of UDP traffic (probably through **spray**) or user-level processes are not draining the UDP socket quickly enough. The NFS server daemon socket overflows most frequently, and running more nfsd daemons will usually prevent packet drop outs at the UDP layer by increasing the probability that at least one daemon will be sleeping on the socket when a datagram is delivered to it.

Ethernet Receive Buffers

A shortage of ethernet receive buffers will manifest itself as an excessive ethernet input error rate:

```
[sunne]% netstat -i
Name Mtu Net/Dest     Addres    Ipkts   Ierrs Opkts Oerrs Collis Queue
le0 1500 129.151.52.0 sunne     2242490 570 2814032 478   16386  0
lo0 1536 loopback     localhost 285548   0  285548 0         0   0
```

Dividing the *Ierr* count by the total number of input packets, *Ipkts*, should yield an error rate of less than 0.025%. An error rate higher than this threshold may be due to electrical problems with the transceiver or drop cable or to packets dropped due to lack of receive buffers. Sun-3s, particularly when serving SPARC clients, and SPARC machines with more than 2 network interfaces are most likely to run out of receive buffers.

The file `/sys/sunif/ie_conf.c` contains default values for the number of **ie** network interface buffers. `le_conf.c` configures the **le** interfaces. The values of interest are:

```
ie_rbufs    receive buffers
ie_rbds     receive buffer descriptors
ie_rfds     receive frame descriptors
```

A buffer and its two descriptors occupy about 1600 bytes. The interface configuration file defines both "high" and "low" versions of these parameters. The "high" values, e.g. `ie_high_rbufs`, are used when more than one network interface is present, otherwise the "low" values are used. The number of buffers and associated descriptors may be increased subject to the following constraint:

```
ie_rbufs > ie_rbds > ie_rfds
```

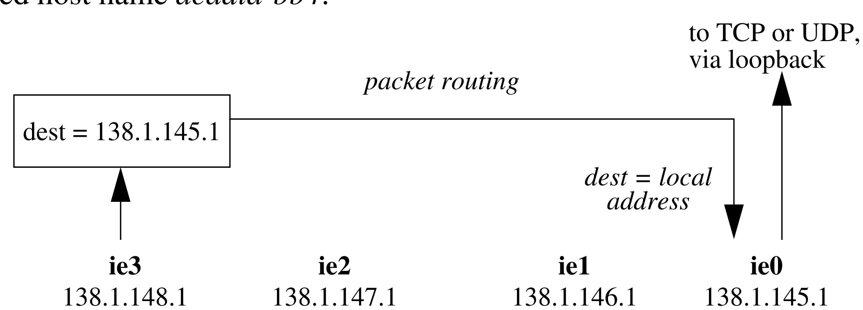
In addition, the total size of the ethernet receive buffers and descriptors must not exceed about 125 Kbytes on a Sun 4/400 machine so that buffers for off-board ethernet interfaces may be I/O cached.

3.2. Multi-Homed Hosts

A multi-homed host (one with more than one ethernet interface) has a different host name and IP address assigned to each interface. For example, a host on four backbones has four entries in the `hosts` file:

```
138.1.145.1 acadia-bb1 acadia # ie0
138.1.146.1 acadia-bb2        # ie1
138.1.147.1 acadia-bb3        # ie2
138.1.148.1 acadia-bb4        # ie3
```

On networks other than 138.1.145.0, any packet addressed to *acadia* will need to be forwarded to the 138.1.145.0 network. In the absence of other routers, *acadia* will do the forwarding itself. The sender transmits the packet to the named gateway host for network 138.1.145.0 - in this example, the packet is incident to *acadia* on interface **ie4**, which is assigned host name *acadia-bb4*.



acadia-bb4 needs to forward the packet to network 138.1.145.0, and the kernel's routing tables show that the route to this network is through interface **ie0**, which happens to be on that network. The packet never reaches the 138.1.145.0 network. because the outbound network interface realizes this packet has the local host as a destination. The packet gets turned around through the loopback interface. Even though the name *acadia* uniquely identifies the host, this name only exists on the 138.1.145.0 network; when *acadia* is referenced from other networks, the packets are sent to the appropriate *acadia-bb*n interface and forwarded internally.

Unnecessary packet forwarding drains CPU cycles and adversely affects performance. For `rlogin` and `telnet` the effects on user-perceived performance may be minimal, although the additional server loading should be considered. The SunOS NFS protocol, however, will enable the dynamic retransmission algorithms if the client and server are not on the same network; to avoid this the server name assigned to the client's network should be used in the client's `fstab`:

```
acadia-bb4:/home/acadia  /home/acadia   nfs  hard,bg 0 0
```

3.3. Improving Routing Strategies

SunOS 4.1 starts the IP routing daemon `in.routed` as part of the default boot process for all machines with local disks. `routed` listens for broadcasts of routing information and builds the kernel's IP routing table; the current state of this table may be examined using `netstat -r`. For most desktop machines, dynamic routing as provided by `routed` is inefficient for two reasons:

[1] The route daemon wakes up every time a routing information packet is received and host routers will send these packets every 30 seconds. In a heavily loaded system, scheduling the route daemon could require paging or swapping out another more interesting process.

[2] The number of routes defined directly affects the cost of sending a packet to a non-local network. Packet routing tends to be CPU intensive, and fewer CPU cycles are spent locating routers for non-local networks when only a few routes are defined.

With only one router on the local network, a single default route should be defined. The routing daemon is disabled in `/etc/rc.local` and replaced with a default route:

```
#if [ -f /usr/etc/in.routed ]; then
#     in.routed;echo -n ' routed'
#fi
route add default gatehost 1
```

The route daemon is not started on diskless clients, since they inherit a default route from their **bootparam** packet.

If multiple routes exist, it is possible to reduce the total volume of routing-related broadcasts by running the route daemon in broadcast mode only on hosts that have more than one network interface. On other hosts, the route daemon is run in quiet mode by invoking it with the `-q` flag:

```
in.routed -q
```

Machines with a single network interface will receive route broadcasts but will not forward this information to other hosts. This approach is ideal if some of the available routes are unreliable or if routes are made available only during specific day parts. Routing information reflecting the current state of the network will be maintained by the hosts actually doing the packet forwarding.

A default route can be used to define the router of choice with multiple routes out of each local network. This machine should be primarily responsible for routing packets and not be expected to provide other CPU loading services. The default router should run the route daemon, allowing it to determine which networks are best served by other routers on the local network.

If a router receives a packet for which it is not the optimal choice, it will send an ICMP redirect message to the sender and forward the packet on to the better router. Sub-optimal can be simply defined as any case in which a packet is forwarded on the same interface on which it was received; this implies that another host on the same network would have been a better first choice. When the sender receives the ICMP redirect, it updates its routing tables with information about the preferred route to the foreign network.

The usefulness and cost of ICMP redirects depends upon the number of networks connected through non-default routers. If the single-interface host can use the ICMP redirect to update its routing tables, then it will make a better routing choice for future transmissions to the target network. If packets are continually sent to the sub-optimal router, then the actual traffic level on the network is doubled: packets received by the router and resent on the same network to another router. However, some hosts cannot utilize the ICMP redirect messages, and their generation by the gateway merely adds to the traffic level on the network.

ICMP redirect messages can be suppressed by modifying the kernel configuration on the gateway host. The relevant flag is in `/sys/netinet/in_proto.c:`

```
#ifndef IPSENDREDIRECTS
#define IPSENDREDIRECTS 1
#endif
int   ip_sendredirects = IPSENDREDIRECTS
```

Rebulding the kernel with `IPSENDREDIRECTS` set to zero will turn off ICMP redirect message generation.

4. NFS Strategies

The Network File System (NFS) has several exposures to performance degradation:

- Congested networks that prevent packets from reaching the client or server hosts.
- Sluggish servers that fail to reply within the Remote Procedure Call (RPC) timeout

period.

- Client disk mounting strategies that create bottlenecks in the filesystem hierarchy.

SunOS measurement tools help to identify the source of each of these problems.

4.1. Lost Packets and Slow Servers

NFS clients cannot distinguish between a server that is sluggish and one that has crashed. Similarly, the client cannot determine whether packets composing a request were lost in transit from the client to the server or if the server just has not responded to the request.

The `nfsstat` utility will show request demographics and history at both the Remote Procedure Call (RPC) and NFS layers. `nfsstat` has five options to limit its output:

n	NFS only
r	RPC only
s	server-side only
c	client-side only
z	zero statistics after display

The NFS statistics provide a breakdown of the number of calls made to each NFS procedure and a count of the number of NFS calls rejected. The RPC statistics show how many calls were made successfully, and how many calls resulted in timeouts, retransmissions or duplicate responses.

On the client side, RPC statistics are used to differentiate between slow servers and congested networks:

```
[sunne]% nfsstat -rc

Client rpc:
calls badcalls retrans badxid timeout wait newcred timers
224978  487      64      265     549     0     0      696
```

The first column is the total number of NFS RPC calls made to all NFS servers. The *timeout* count reflects the number of RPC calls for which no reply was received before the timeout period expired; these calls may generate a retransmitted request or may cause an RPC timeout error. The number of RPC timeouts is included in the count of *badcalls*, and the number of calls retransmitted is shown under *retrans*. In general,

```
badcalls + retrans => timeout
```

Each NFS call has a unique transaction number known as an *xid*. The *xid* for a call is retained each time the request is retransmitted, and the *xid* is included as part of a server's acknowledgment that a request completed. The *badxid* column shows the number of times an *xid* was returned that matched that of a call already completed. It is a rough count of the number of retransmitted requests that were actually serviced, albeit after the default RPC timeout period had elapsed.

In SunOS 4.1, the NFS server code has a duplicate request cache. Any NFS request having an *xid* that matches the transaction number of a request currently in progress on the server is ignored. In addition to reducing server load by eliminating redundant operations, the duplicate request cache decreases the total number of *badxid* responses counted by the client.

The method for determining whether the NFS servers or the network is causing NFS retransmissions is to compare *badxid* and *timeout*. If they are of the same magnitude, then the NFS servers are receiving and servicing all of the RPC requests. The network is functioning properly, but the servers are too sluggish to meet client demands. If *badxid* is close to zero, then the servers are not seeing the first (or first few) request transmissions because the packets are being lost in transit.

4.2. Accommodating Slow Servers

The easiest way to accommodate slow servers is to increase the NFS RPC timeout period. The default timeout is 0.7 seconds, but the timeout may be set explicitly using the *timeo* NFS mount parameter:

```
sunne:/home/sunne  /home/sunne  nfs bg,timeo=150 0 0
```

The timeout period is specified in tenths of a second, so the example above more than doubles the default timeout to 1.5 seconds. The timeout value should not be increased to an arbitrarily large value, because a request that is lost due to network congestion rather than server sluggishness will also suffer from this long timeout before it can be retransmitted. In general, the timeout period should be as short as possible to accommodate the slow servers. The output of `nfsstat -m` (see *Dynamic Retransmission* below) may help in estimating timeout periods, but the best value is usually found empirically.

After an RPC request times out, the default timeout value is doubled and the request is tried again. The single-call RPC timeout is clamped to 60 seconds. This process continues until a major timeout occurs, after *retrans* attempts. The default value for *retrans* is 5, and it may also be specified as an NFS mount parameter.

```
sunne:/home/sunne /home/sunne nfs   bg,soft,retrans=7   0 0
```

The client's handling of a major timeout depends upon whether the filesystem was *hard* or *soft* mounted. If the filesystem is soft-mounted, then an error is returned by the system call that generated the NFS call. With a hard-mounted filesystem, the initial timeout is doubled and the timeout loop is re-entered. The initial timeout value is clamped to 30 seconds.

Modifying the default *retrans* mount parameter is most applicable when filesystems are soft-mounted. Increasing the number of retransmissions before a major timeout event will make clients more tolerant of slow servers or those that are prone to crashing. With hard-mounted filesystems, however, an increased *retrans* value may only send more packets to a host that has crashed, adding to the noise level on the network. In general, if a server has not responded within the first few retransmissions, it has experienced a failure.

Soft-mounted filesystems should only be used for read-only data files. In SunOS 4.1, executables are paged directly from the *vnode* into the process image, so the virtual memory system must be guaranteed that executable pages can always be fetched from a local or remote disk. If a filesystem containing executables is soft-mounted, it is possible for the paging system to fail on a major timeout. If it is necessary to interrupt an operation that is in a retransmission loop waiting for a hard mounted filesystem, the *intr* mount option specifices that interrupt signals should be processes at each timeout.

```
sunne:/home/sunne  /home/sunne  nfs hard,intr 0 0
```

Mounting a filesystem with the *hard* and *intr* options mimics the behavior of a soft mount when the user must force an RPC call to terminate before a reply is received.

4.3. Bypassing Network Bottlenecks

If `nfsstat` shows that packets are being lost between client and server, they may be encountering bottlenecks at bridges or slow ethernet interfaces on the server. Note that packets lost to collisions at the local network interface do not affect the RPC timeout count because the collision detection and recovery is handled below the RPC layer. Devices that cannot receive sustained bursts of IP packet packets, however, will affect NFS performance on the network.

NFS requests are buffered on the client side by the `biod` daemon. The default buffer size is 8 Kbytes, so that reads are always performed 8 Kbytes at a time and writes are batched until 8 Kbytes must be sent to the server. The *rsize* and *wsize* NFS mount options change the granularity of the buffered operations:

```
sunne:/home/sunne /home/sunne nfs bg,rsize=2048,wsize=2048 0 0
```

The client will generate fewer consecutive IP packets for each NFS read or write request with a smaller buffer size, increasing the chances that the NFS request makes it through a slow network interface on the first transmission.

Decreasing the NFS buffer size is not without cost: With a smaller buffer size, an NFS read or write operation requires more IP packets to transfer the data. With an 8 Kbyte NFS buffer size, 6 IP packets suffice to transmit all 8 Kbytes of data. If the buffer size is reduced to 2 Kbytes, then 4 RPCs, each with 2 IP packets, will be required. A total of 8 packets are sent to (or from) the server, and the server must service 4 times the number of RPC requests. The actual traffic volume on the network and the server load are increased, but the desired effect is to improve the probability that NFS requests reach their destination in their entirety.

When using NFS across a wide-area network (WAN), explicit resizing of the NFS buffers will reduce interference between NFS traffic and other packets sent by interactive users. On a limited bandwidth link, long trains of NFS packets will cause a noticeable delay in character echo. Reducing the NFS buffer size compensates for the smaller mean transmission unit (MTU) of the wide-area network link. Typically *rsize* and *wsize* are turned down to 1024 bytes, with each NFS request filling one IP packet.

4.4. Dynamic Retransmission

SunOS 4.1 has a dynamic retransmission algorithm used when an NFS client and server are on different logical networks. NFS-generated packet trains are frequently dropped by routers or network gateways, and the dynamic retransmission code attempts to detect these packet drops and scale down the NFS buffer sizes as shown above. The client and server must be separated by a router or a gateway host for dynamic retransmission to be enabled. A physical network partitioning enforced by a LAN bridge only provides physical separation; the two network segments are still a single logical IP network and do not meet the conditions for enabling the retransmission feedback algorithms.

The client sets a timer when sending an NFS request to a server based on the expected response time for that server. When dynamic retransmission is enabled, expiration of this timer is feedback for modifying the expected response time and retransmission timeout and for reducing the read and write buffer sizes. The output of `nfsstat -rc` includes a *timers* field, showing the number of dynamic retransmission timer expirations. This value may be larger than the number of RPC timeouts because the dynamic retransmission timer and the RPC timeout are independent. The transmission timer is based on projected response time while the RPC timeout limits the maximum server response time.

If the timeout is set explicitly in the NFS mount options, this values is taken to be an initial hint if dynamic retransmision is enabled. The current *rsize* and *wsize*, as well as the smoothed round-trip response times are displayed using `nfsstat -m`:

```
[sunne]% nfsstat -m
/home/thud from thud:/home/thud (Addr 129.151.52.13)
   Flags: hard read size=8192, write size=8192, count = 5
   Lookups: srtt=7 (17ms), dev=4 (20ms), cur=2 (40ms)
   Reads: srtt=14 (35ms), dev=3 (15ms), cur=3 (60ms)
   Writes: srtt=56 (140ms), dev=5 (25ms), cur=9 (180ms)
   All: srtt=7 (17ms), dev=4 (20ms), cur=2 (40ms)
```

The first summary line shows the mount options and the current read and write buffer sizes. The last four lines contain server response time and retransmission information. The first value, *srtt*, is the average round-trip response time. *dev* is the average deviation from the round-trip average time, and *cur* is the current dynamic retransmission counter. Response time and retransmission counters are maintained separately for path name look ups, read and write operations, because these operations vary greatly in terms of the server resources used to complete them. The final line of the summary gives an average over all operations.

If the read and write buffer sizes have been reduced to unusally small values, the client may be experiencing several RPC timeouts before a request can be successfully completed by the server. A large deviation in the round-trip time might indicate an intermittent gateway congestion problem, in which bursts of packets are dropped when other traffic is clogging the gateway. Small deviations with large round trip response times imply that the server is falling behind the RPC transmissions. The server's response is fairly predictable but may be outside of the time allowed by the RPC request timeout. Explicit setting of the NFS buffer sizes and timeout values will provide initial hints to the dynamic retransmission algorithm and may eliminate successive RPC timeouts.

4.5. Improving Mount Point Strategies

NFS looks up pathnames a component at a time because each component in the path might be a mount point for another filesystem. For example, if `/usr/local` is NFS-mounted, opening `/usr/local/lib/libY.a` requires the client to:

- Look up `usr` in / and then look up `local` in `/usr`.
- Get a file handle for the `/usr/local` mount point.
- Look up `lib` in `/usr/local` on the NFS server.
- Look up `libY.a` in `lib`.

There are two implications of this pathname resolution:

[1] The server must be capable of locking a sufficient number of *inodes* in core.

[2] Symbolic links should be avoided when they will be frequently resolved, due to the cost of reading the link and resolving the new pathname.

The size of the SunOS kernel directory name lookup cache (DNLC) is based on the `maxusers` kernel configuration parameter. `maxusers` should reflect the number of virtual users of the server, including NFS clients and users who remotely log in to the machine. If the DNLC is too small, the kernel will have to frequently flush entries and reload them when they are referenced again; in some cases it is possible to deadlock the system when all DNLC entries are locked and a process is waiting on a free table slot.

The NFS server-side statistics will also indicate an excessive number of symbolic link resolutions:

```
[sunne]% nfsstat -ns

Server nfs:
calls  badcalls
1563212  41
 null   getattr   setattr root     lookup     readlink    read
 32 0% 527178 33% 9288 0% 0 0%  449726 28% 189466 12% 188665 15%
wrcache  write     create  remove    rename  link   symlink
 0 0%   134797 8% 13799 0% 15826 1%  2726 0% 4388 0% 74 0%
 mkdir   rmdir  readdir  fsstat
1575 0% 1532 0% 23898 1% 242 0%
```

In this example, symbolic links were read with half the frequency of pathname component lookups. Replacing the link with a direct NFS mount point, or shuffling directory structures to eliminate the need for the link would reduce the number of RPC calls to this server by over 10%.

5. Swap Space Management

Swap space is used to save anonymous memory pages, consisting of a process's user area, stack and dynamically allocated storage. In SunOS 4.1, raw partitions, local files and NFS-mounted files may be used as swap devices; each is treated as an array of pages by the page daemon. SunOS 4.1 also supports the tmpfs swap-based filesystem, which can be used to cache short-lived files without the cost of going to disk.

5.1. Swapping on Multiple Partitions

Swapping may be done to multiple partitions or combinations of files and raw partitions. There is no explicit swap interleaving, so pages move to additional swap devices as the primary device fills up. Over time, this will tend to produce a random distribution of pages over swap partitions.

Multiple swap partitions (or files) will improve system performance if the total, simultaneous throughput from the multiple partitions is better than that from a single partition. If the total throughput from a single swap device is the same as that from multiple swap partitions, then swapping on multiple swap partitions may actually decrease performance due to queue imbalances.

Consider a disk subsystem with a throughput of 4 Mbtye/sec, where each disk can sustain 4 Mbyte/sec as well. However, overlapping disk seek operations may give the 2-disk configuration a higher throughput for random disk read and write operations than that of a single disk machine. If two partitions (or files) on the same disk are used, however, equal traffic to each swap device implies that each "swap queue" can use about 2 Mbyte/sec of the total available bandwidth. There are twice as many swap queues, but each is twice as slow as that for a single large partition. In general, minimizing the number of queues will minimize the average request waiting time, if throughput is the same in both cases.

5.2. tmpfs

The SunOS swap-based filesystem, *tmpfs*, is a memory-resident filesystem. *inode* information is kept in core, and writes to *tmpfs* files are cached in memory. The *tmpfs* filesystem uses the same resources as other processes: memory is used to cache file pages, and swap space is used to hold data that must be paged out of physical memory. File pages and *inode* data are not preserved across reboots. *tmpfs* is not a RAM disk as it does not have a fixed size and its contents can be paged out of memory if required.

tmpfs is best used on machines that create many short-lived files. `/tmp` is the most common directory replaced by a temporary filesystem, and other directories can be added as required. An application that creates scratch files in `/usr/local/data` would benefit from having a *tmpfs* filesystem mounted on this directory:

```
mount -t tmp swap /usr/local/data
```

The keyword `swap` is used in place of a physical device name because the backing store for this filesystem is the swap area.

There are several caveats for using tmpfs:

[1] Processes with large virtual memory spaces will reduce swap space available for the

temporary file system. If a large *tmpfs* is needed on such a system, more swap space should be added.

[2] Long-lived files should not be placed in a temporary file system; these files properly belong in `/var`. Persistent files will consume pages of memory, and may cause more paging activity than would occur if the files were written directly to disk.

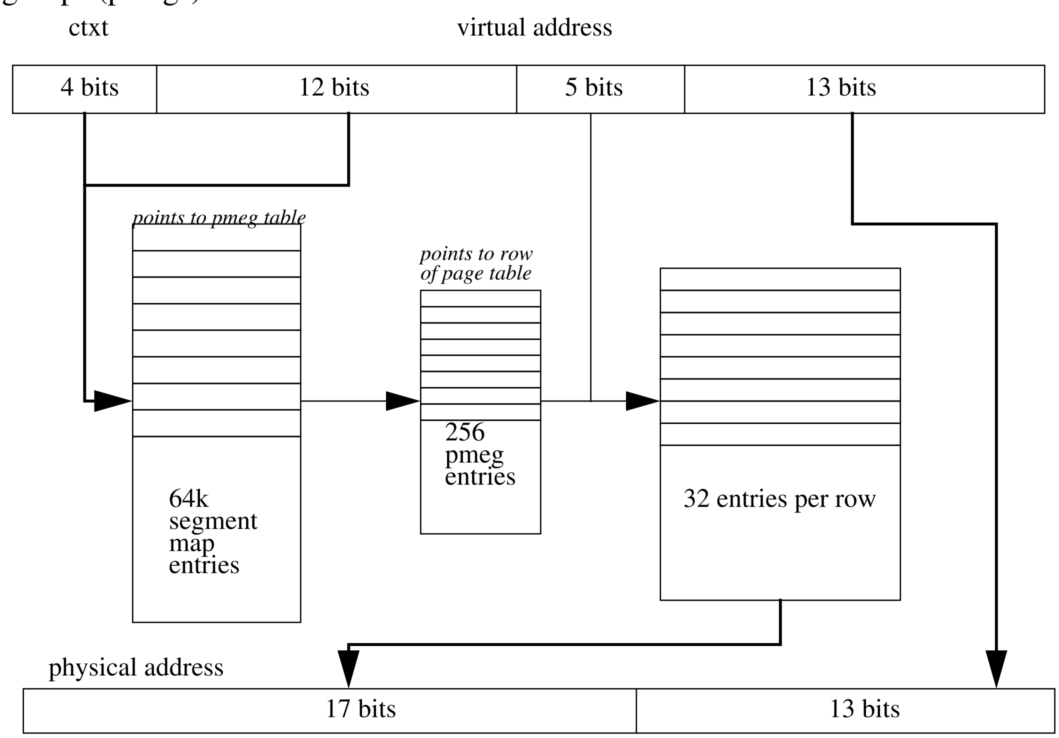
[3] *tmpfs* cannot be used for any directory in which crash recovery files are saved. `/var/tmp` is not a candidate because `vi` writes its recovery files there; replacing this directory with a temporary filesystem would break `vi -r`.

6. Memory Management

Applications that run poorly under SunOS 4.1 may suffer from virtual memory management problems. There are two caches that can significantly improve performance: the memory management unit (MMU) page translation tables and the virtual address memory cache. If the application cannot sustain a high hit rate to both of these caches, then its effective memory access time will be increased as it must wait for reloading of memory cache lines or page translations. Applications with good locality of reference will run poorly if the neighborhood of their references is spread across a sparse virtual address space or if it causes cache aliasing problems.

6.1. Sparse Address Spaces

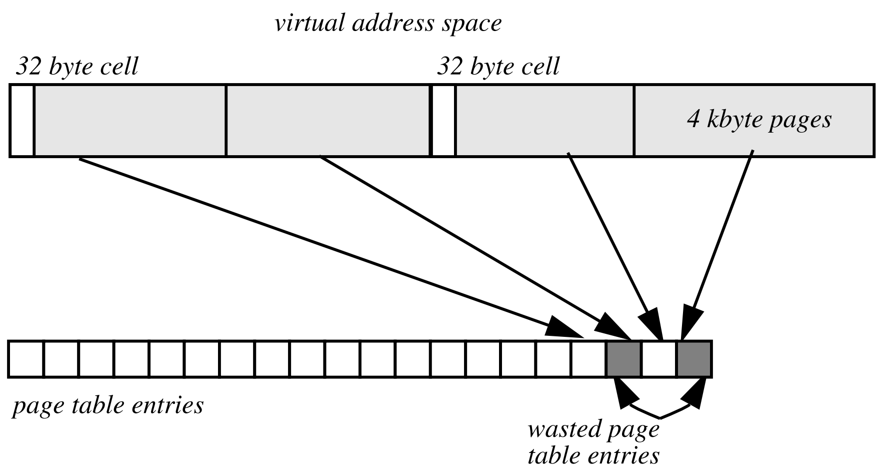
SPARC system MMUs are two-level lookup tables of segment maps and page map entry groups (pmegs):



The process context register and 12 most signifcant bits of the virtual address select an entry from the segment map. The segment map points to a pmeg containing 32 page table entries. The pmeg table is indexed by the lowest-order bits in the virtual address above the page offset, so the pages in a pmeg are at contiguous *virtual* addresses. A single pmeg addresses 128 Kbytes on machines with 8 Kbyte pages, and 64 Kbytes on SPARC desktop machines with 4 Kbyte pages.

If the process has "holes" in its address space, the pmeg mapping will be less efficient because the pmeg will not point to a full complement of page table entries. Sparse address spaces effectively reduce the size of the MMU, which is itself a cache for virtual address translations.

Inefficient use of *malloc*() or *valloc*() may lead to fragmented address spaces. Alternately allocating and freeing buffers of varying sizes may leave holes in the application's address space if requests of varying sizes cannot be filled with any free cells. For example, allocating buffers of 32 bytes, 4 Kbytes and another 32 bytes will leave a page-sized hole in the virtual address space if the 4 Kbyte buffer is freed before the 32-byte cells. If small memory cells are left orphaned between large holes, the effects of fragmentation are exacerbated:



To identify processes that fragment their address spaces, watch the total size of the process during its execution using `ps`. If the process continually grows, then it is not re-using memory that was previously freed, possibly because the sizes of new requests do not match the sizes of holes in the *malloc*() managed heap. A possible solution to this problem is to define several commonly used cell sizes, and to always request the smallest cell that will hold the data. If the cell sizes are multiples of some common factor, then adjacent free cells will be coalesced into a cell of a larger size by *malloc*().

6.2. Data Structure Stride

All Sun systems (except the 3/400 series) have virtual address caches. On desktop machines, the cache size is 64 Kbytes and on servers it is 128 Kbytes. The cache line size is 32 bytes. The low-order 5 bits of the virtual address select a byte from the cache line; bits 5-20 of the address select a line from the cache and the high-order 12 bits are used as a cache tag. Any two virtual addresses with bits 5-20 in common - those that are separated by the cache size - will alias to the same cache line.

All references to memory go through the cache. To improve the cache hit rate and minimize the number of wasted line reloads, related data structures should be allocated to avoid cache aliasing. Related data structures are those that are frequently used in the same sections of code; for example, copying from one structure to another, or updating statistics in one array based on data in another. The stride used in accessing related data structures should not be a multiple of the cache size. If it is, then every reference to one structure will flush out a line loaded for one of the related structures.

Global data structure addresses can be examined with `nm`:

```
% nm a.out
00004098 B _stats
00044098 B _sample
```

For structures that are dynamically allocated, code profiling must be used to determine where the structures are placed at run time.

The following simple piece of code will cause a cache miss on every variable reference in the loop, increasing the effective memory access time.

```
struct {
      int hi;
      int lo;
      int n; } stats = { 0, 999, 0 };

...other declaractions...

int sample;

while (...) {
      if (sample > stats.hi)
            stats.hi = sample;
      if (sample < stats.lo)
            stats.lo = sample;
      stats.n++;
}
```

The address of `stats` and `sample` are exactly 64 Kbytes apart (0x40000). The two data items use the same cache line, and each reference to one structure flushes out the cache entry for the previously loaded one. The `if` constructions will force at least 2 cache flushes for each trip through the loop. While this example is trivial, other data structures may end up with cache-sized strides between them, adding significantly to the overhead of each operation performed on the structure.

If two related structures are separated by a multiple of the cache size, changing the order of their declaration will change their relative position in the process address space. To avoid aliasing, structures need to be separated by a cache line or more, so allocating an extra 32 bytes at the end of a dynamically allocated structure will skew related structures enough to eliminate cache aliasing.

7. Optimizing I/O

The SunOS 4.1 virtual memory management system maps pages associated with a *vnode* directly into a user's address space, so that all I/O to any device represented by a *vnode* is performed through page faults. If the application attempts to read a page that has not been loaded into memory, the virtual memory system recognizes this access as a page fault and retrieves the page from disk or the underlying physical device.

With mapped pages, there is no need to copy data into and out of kernel space for I/O operations. The kernel builds a buffer header, and points the buffer data address at the user's buffer. The user's buffer is remapped into the kernel's address space, but the two mappings share the same underlying physical page frame. The net effect of mapping files is that the traditional file buffer cache is no longer needed. All of free physical memory is treated like a cache for *vnode* pages used by all applications.

7.1. I/O Caching

Shared mappings between user and kernel address imply that kernel address alignment requirements apply to user addresses as well. Sun 4/400 and 3/400 machines have an I/O cache with a 32-byte line size. To ensure that a buffer can be cached, it must be aligned on a 32-byte boundary.

Any application that performs I/O to non-disk devices should force its buffers to be aligned on 32-byte boundaries. The *malloc*() library call does not guarantee alignment, so *memalign*() or *valloc*() must be used. *valloc*() aligns allocated memory to a page boundary, while *memalign*() returns a buffer that is aligned to a boundary that is a power of two. Note that buffer alignment is not an issue for file operations or other I/O done through *mmap*(), because *mmap*() aligns its address range on a page boundary.

7.2. Sequential File I/O

The SunOS pageout daemon, like its Berkeley predecessor, is scheduled whenever the number of free pages of memory dips below some acceptable level. Pages are written to disk based on a pseudo-LRU algorithm implemented by a 2-hand clock scan of the pages. As the second hand touches a page, it's referenced bit is cleared; if the bit is still clear when the minute hand follows it, then the page is freed.

When performing sequential file I/O, however, LRU page replacement policy proves to be least efficient. After a sequential read (or write) operation, the application will not need to re-use that page. However, the page is least likely to be freed because it was most recently used. The problem is exacerbated by applications that use the *mmap*() system call because the pages of the file become part of the application's address space. Any application doing sequential file I/O through the *read*() and *write*() system calls will cause a severe shortfall in available memory because *read*() and *write*() are implemented using *mmap*().

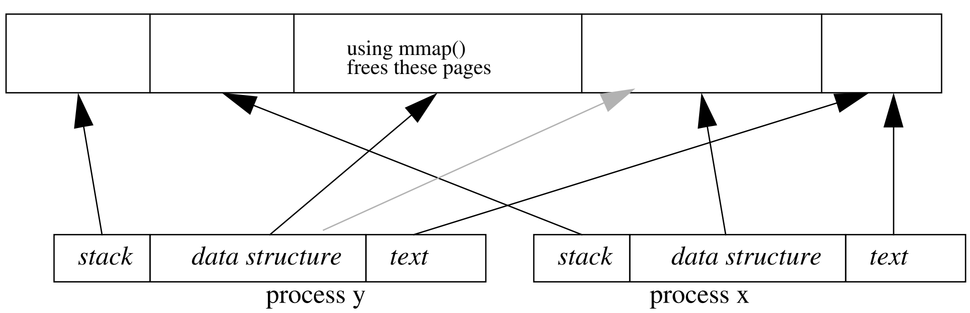
The paging policies of SunOS are based on performance, not fairness: a process will not be prevented from consuming all available physical page frames. Sustained periods of sequential I/O, therefore, will hurt interactive performance as processes are swapped out to meet the memory demands of the process faulting in (or out) file pages. The *madvise*() utility can be used with *mmap*() to supply hints to the virtual memory system that prevent processes performing sequential I/O from using all available memory:

```
int fd;
char *fp;

fd = open("/home/thud/stern/bigfile", O_RDONLY);
fp = mmap(0, 1024 * 1024, PROT_READ, MAP_SHARED, fd, 0);
for (i = 0; i < 1024; i += 64) {
      /* hint to keep the "working set" in core */
      madvise(fp + i * 64*1024, 64 * 1024, MADV_SEQUENTIAL);
      ...sequential processing on 64 Kbytes of file ...

      /* hint to explicitly free the pages just used */
      madvise(fp + i * 64*1024, 64 * 1024, MADV_DONTNEED);
}
```

The *madvise*() system call hints that an address range will be accessed in a sequential fashion so that the virtual memory system can pre-fetch pages in this range. When the application is done with the range of pages, the second *madvise*() call explicitly frees them so the physical page frames will not be held until the system runs out of memory and is forced to free pages.

7.3. Shared File I/O

Large, read-only files can be implicitly shared between processes using *mmap*(). When the first application maps the file into memory, it establishes a mapping that will be re-used by other applications that open the same vnode in the filesystem. Multiple instances of the same application can greatly reduce their aggregate memory requirements.



Consider the example of a large file of statistical samples; 6 different applications need to read this file and reduce the data in various ways. If each application opens the file and reads the data points into an array, then there will be 6 distinct copies of the array in core at any time. However, if the applications map the file directly into memory and operate on the address range returned by *mmap*(), a significant performance improvement will be realized. If the data is already in an acceptable format, there is no need to generate private copies of it.

8. Conclusions

Using features of SunOS 4.1, it is possible to tune desktop machines to meet the needs of individual users and to configure servers to provide the best response time over a range of conditions. Performance analysis and tuning, however, should not be a periodic event prompted by user dissatisfaction. Instead, it should be one component of a basic system management policy. Regular evaluations of capacity utilization and system response will highlight potential problems before software releases or due dates for student projects make them painfully obvious.

Acknowledgments

Many people within Sun and in Sun's user community contributed to this paper. Neal Nuckolls (Sun) and Vicki Lewolt Schulman (Auspex) provided outstanding examples and explanations of various networking phenomenon; Peter Galvin (Brown University) encouraged my examination of NFS client behavior and dynamic retransmission; and Kevin Sheehan, Janice McLaughlin and Patrick Stirling (Sun) filled in details on everything from STREAMS drivers to virtual memory system behavior. Thanks also to Tom Kessler and Chuck Kollars (Sun) for their comments.