Pinpointing System Performance Issues

Douglas L. Urner – Berkeley Software Design, Inc.

ABSTRACT

The explosive growth of the Internet in recent years has created a heretofore unprecedented demand for system performance. In many cases, system administrators have been left scrambling, trying to understand the performance issues raised by the loads and new technologies they are facing.

This paper suggests methods for analyzing system performance potential, prioritizing the search for bottlenecks and identifying problem areas. While many of the specific tuning suggestions are particular to BSD/OS and other 4.4BSD derivatives, the principles on which they are based should generalize well to any system being used to provide Internet services.

Introduction

This paper discusses methods for tracking performance problems, specific parameters that can be tuned, and data analysis tools available on almost all Unix systems and some other systems, as well.

Starting with a study of the overall architecture, this paper covers configuration, kernel tuning, disk subsystem analysis, networks, memory usage, application performance, and network protocol behavior.

The Big Picture

Methodology

A bit of method goes a long way: it saves time; it gets to the source of the problem quickly; and it helps avoid missing the problem. When trying to improve performance, start where the biggest gains can be had. It makes no sense to speed a system by a factor of 1.01 when a speedup of 5x can be gained. In fact, given today's rules of hardware speeds ever-increasing (at a constant cost) over time, it makes little sense to spend time on 1% speedups. On the other hand, mistuned systems might run at only 10% of their potential speed. Concentrating on the factors that can gain back the 90% is obviously far more productive than concentrating on 1% gains. Of course, figuring out which part of a system gets back the other 90% is the problem.

Overall Architecture

When a system's overall design is wrong, performance can really suffer. The overall design must be right if a system is to achieve its potential. Be sure that your assumptions about the system's use and potential are clear.

Consider the model used by many Unix systems for network demons. The server forks a process to handle each network request it receives. This works reasonably well when the connections are infrequent or relatively long-lived (at least a handful of interactions). But, when connections are short-lived (a single interaction or two), the cost of the fork() can dominate the cost of handling the connection. This is too often the case for HTTP (the WWW protocol). Over the last few years, the connection rate for a 'busy' web server has gone from dozens of connections per hour to dozens of connections per second. A strategy of "pre-forking" HTTP servers or using a single process and multiplexing requests with select() can result in a 10x-100x improvement in performance [1].

Configuration Errors

In the big picture, most of the easy performance gains arise from system configuration improvements. Most systems come out of the box with a configuration that aims to be a "jack of all trades" and, as it turns out, master of none. This means that you can greatly improve performance by examining and improving items like:

- basic kernel tuning,
- disk layout,
- system memory sizing and allocation,
- hardware selection and configuration,
- configuration of system services (e.g., a local DNS cache), and
- computer center scheduling.

Best case improvements can approach the 100x range.

Application Tuning

Usually, it is difficult to increase application performance by 10x or more. Out of the box, an application is likely to perform acceptably, though it might not be a stellar performer under heavy load. The first rule for applications is to choose your applications carefully and with an eye toward performance.

In most cases, vendor's code is not modifiable or tunable to a great extent. Be sure to take the time to understand how an application uses system resources and to investigate the tuning options that are available. For example:

- Should the application's files be split amoung several disk drives?
- Can unused features be switched off?
- Is the application logging too much data?

Kernel Tuning

Kernel algorithms must be well-matched to the task at hand. Basic kernel tuning is a fundamental tool in performance improvement. Regrettably, beyond basic tuning, few gains can be had without technological innovation (usually at great cost unless amortized across a very large number of systems).

Developing Expectations For System Performance

It is hard to tune a system without a basic collection of performance reference points to help you analyze the data provided by your monitoring tools. For example: File transfers typically run at between 650 and 750 KB/second on a active Ethernet at your site. Is this good? Could it be better? Or, a mail server you manage is handling 100,000 messages a day and seems to be saturated. Should you expect more? Where do you start looking for problems?

Having an "intuitive" feel for the answers to questions like these makes the process of performance analysis considerably more productive and more fun. Run your performance monitoring tools and get a feel for the numbers you see under various kinds of loads. In many cases, you can generate useful synthetic loads with a few lines of C or a bit of perl code. This is the information that gives you a feel for what to expect from your machines. It will save you from having to individually calculate and analyze each problem you take on.

Take the time to think through some of your system's basic limits. For example, what kind of performance can you expect from TCP/IP running on a 10 Mbit Ethernet?

Starting from the raw speed of 10,000,000 bits per second (note that 10,000,000 is 10 x 1000 x 1000, not 10 x 1024 x 1024), you have to calculate the bandwidth that will actually be available for carrying data (payload). First, calculate the link layer and protocol overheads:

8 bytes	Ethernet preamble and start of frame delimiter
14 bytes	Ethernet header
20 bytes	IP header
20 bytes	TCP header (assuming no options)
4 bytes	Ethernet CRC
12 bytes	Ethernet interframe gap
76 bytes	Total per packet overhead

Then measure (or estimate) the average packet size on the network you are interested in. You can use *netstat*(8) to get a count of the number of packets and bytes sent and received by an interface.

Next, make an estimate of the acknowledgement (ACK) overhead. If the traffic is predominantly in one direction, there will be ACK traffic that will reduce throughput. In the worst case, you would see one ACK for every two packets. Each ACK will consume 76 bytes unless it can be 'piggybacked' with data.

Pulling this information together into a table provides a picture of the performance to expect from TCP/IP on Ethernet.

Average Packet Size (bytes)	Uni-directional Traffic (KB/sec)	Bi-directional Traffic (KB/sec)
64	439	558
128	646	766
256	845	941
512	998	1063
1024	1098	1136
1500	1134	1162

Similar tables could be produced for UDP or any other protocol of interest.

The above numbers are an upper bound. A system might get close to these numbers during bulk transfers with little competing traffic. In an environment with several busy machines on the same wire, throughput will be somewhat lower, perhaps 70-90% of these numbers.

With these calculations in hand, you can try some experiments to see what happens in your environment. Most systems have rcp available, which offers an easy way to do a quick experiment. Contrary to what many folks think, ftp is usually not the best test of raw network capacity. Many ftp implementations use small buffers and are actually poor performers. Copy a big file (at least a megabyte) from one idle machine to another and keep track of the real-time spent. Do this two or three times to get an idea of your network's consistency. This experiment should reveal approximately the maximum throughput for your system as configured.

Try the experiment again on a loaded network. Note how badly (or not) the performance degrades in your environment.

It is worth doing this sort of analysis and experimentation for any subsystem you are interested in really understanding. Aaron Brown and Margo Seltzer presented a very interesting paper reporting their findings on the performance of Intel hardware at the 1997 USENIX conference [2].

Protocol Performance

Analysis and experimentation assist in understanding performance. Hard facts and knowledge about underlying algorithms can also contribute. Here are some interesting facts that contribute to overall performance of various network protocols:

- TCP/IP packets tend to be small, the average (according to one router vendor) is around 300 bytes.
- When a user's packets are being dropped (due to network load), network performance will be noticeably bad for that user (i.e., 20 second or more response time or 300 b/s FTP

throughput). This is because TCP backs off exponentially when retransmitting lost packets. This is good for the network as a whole, but bad for the affected connection. Good performance requires very low packet loss rates.

- HTTP and SMTP connections tend toward a small number of interactions that move a small or medium amount of data. For example, consider a WWW page that sets up and tears down 14 connections to load 14 small pictures (e.g., a bullet). This interaction pattern means that the TCP 'slow start algorithm' (which initially responds to requests a bit slower than possible to ensure that the network is not overwhelmed) will probably keep performance lower than is theoretically possible.
- New features often break things. While the design of TCP/IP supports adding new features without breaking existing implementations, it doesn't always work that way. It is worth keeping track of what's new on your network (and beyond), although the new code is not always where the bug lies. A common example of this phenomenon occurred a few years ago when TCP connections would fail when RFC1323 options were used. The problem was that some versions of Linux failed to handle the options correctly and corrupted packets. Fortunately, the folks implementing the support for RFC1323 had the forethought to implement an additional mechanism for turning them off on the fly. Another example concerned some Microsoft PPP implementations that would negotiate larger packet sizes than they could handle.

Understanding the Application

To improve overall performance, including applications, you're going to need to dig into the application and understand how it works and what it needs from the system.

Sadly, the application's documentation is usually not where to start. Tools like *ktrace*(1), *ps*(1), *top*(1), *netstat*(8), *tcpdump*(1) (or *etherfind*(1) in the Sun world) let you see how the application is interacting with your system and will yield real clues to the application's behavior.

INN (the news server) is an excellent example of an application that benefits greatly from study and understanding. Steve Hinkle's paper on INN tuning [3] is an excellent example of how to understand an application and then tune it for maximum performance.

Reference Points

As you build up a base of experience, you also build a set of reference points for what to expect from a system in real environments. These expectations can be really useful when trying to make initial "off the cuff" evaluations of system performance. For example, a good operating system can enable these levels of performance:

- A 90 MHz Pentium with 80 MB of RAM buffer cache can feed more than 3 million news articles (over 9GB) per day.
- A 150 MHz Pentium Pro can deliver over 1 million mail messages per day.
- A 133 MHz Pentium can serve static HTML from a cache at T3 speeds.
- A 150 MHz Pentium Pro can handle more than 3 million direct web hits per day, transferring 27 GB of content.
- A 133 MHz Pentium can only handle about 60 HTTP requests per second with a forking proxy.
- A 150 MHz Pentium running screend (user space packet filter) won't be able to handle a T1.
- A 266 MHz Pentium-II can't quite route a 100 Mbit Ethernet running at full load (full routing table, 256 byte packets). It can handle a T3.

These reference points aren't complete; they assume that you have the OS, hardware configuration, RAM, and other support (these reflect observations using BSD/OS 2.1 or 3.0, DEC or Intel Fast Ethernet Adapters, adequate memory and SCSI disks). Collect a set of reference points that are meaningful to you based on the systems that you are working with.

Collecting Data

Here are some hints for collecting performance data.

- **Get baseline data.** Without it you are lost, you won't be able to tell how much good (or bad) you have done.
- **Make sure you have enough.** For example, if you're trying to track down UDP packets "dropped due to no socket," make sure that your tcpdump has run for long enough that you expect to have collected several of these packets (*uptime*(1) and 'netstat -s' will help you pick an interval).
- **Make sure the data are representative**. Continuing with the above example, you also need to consider the possibility that the traffic you are interested in is not evenly distributed, and make sure that the data you collected represents an interval during which you could reasonably expect interesting traffic. To do this, you could start using *cron*(8) and netstat -s to sample every hour to determine when the traffic you are interested in occurs.
- **Make sure you sample frequently enough.** When you're looking for things like loading trends, you need to sample often enough to actually catch the event you're interested in. As a rule of thumb, you want to sample at at least twice the frequency of the event you are trying

to catch. If you don't know the frequency of the event, initially you may need to use a significantly higher sampling rate.

- **Automate your initial data reduction**. Tools like *tcpdump*(1) can generate huge amounts of data. Start by using tools like perl (and in the case of tcpdump, tcpdump's built-in filtering facility) to eliminate extraneous data. When you find no interesting data, don't forget to check your automated tools.
- **Check the log files**. Don't forget to check your system's log files and any logs files kept by the applications for clues.

Tuning the Subsystems

It is easier to look at the individual subsystems in relative isolation from each other.

The Disk Subsystem

Disk subsystem performance is often the single biggest performance problem on Unix systems. If your disks are busy at all (web, mail and news servers all qualify), this is the place to start.

Since the virtual memory system uses the disk when it is out of RAM, start by checking with *vmstat*(8) to ensure your system is not paging (the pages out 'po' column should be zero most of the time; it's OK to page in programs occasionally). If your system is paging, then skip the disk tuning for now and start by looking at RAM availability.

If your application has significant disk I/O requirements, the disk is almost certain to be your bottleneck. On a busy news or web server, correct disk configuration can easily improve performance by an order of magnitude or more.

Disk subsystem optimization has two main goals:

- minimize seek time, and
- match disk throughput to your system's demands.

When thinking about disk throughput, keep your system's load mix in mind. For some applications (e.g., streaming video), your system should maximize the rate at which data flows from the disk to the application. Rotational speed (the limiting factor on transfer rate) will likely dominate all other considerations. Other applications (e.g., mail and news servers) will see file creation and deletion operations dominate performance. File creation and deletion, which use synchronous operations to ensure data integrity, cause seek time to be the dominant factor in disk performance.

While modern SCSI disks are capable of transfer rates that approach or even exceed the speed of the SCSI bus (e.g., 10 MB/sec for Fast SCSI), the limiting factor for Unix systems is typically the number of file system operations that can be performed in a unit of time. For example, on a machine functioning as a mail relay, the limiting factor is going to be the speed of file creation and deletion. See Table 1 for a listing of SCSI speeds.

SCSI Version	MB/s
SCSI-I (async)	3.5
SCSI-I (sync)	5.0
Fast SCSI	10.0
Fast Wide SCSI	20.0
Ultra SCSI	20.0
Ultra Wide SCSI	40.0

Table 1: SCSI Bus Bandwidth.

In both cases, the only ways to increase performance (once you've hit the limit of the disk's physical characteristics) are:

- add disk spindles to increase parallelism,
- add hardware like PrestoServe to decrease seeks,
- identify and purchase a new file system technology that reduces seeks (test it carefully!)

In some cases it will be necessary to make changes to the application to make it aware of multiple directories or to use hardware that will transparently distribute the load across multiple spindles (e.g., RAID).

Disk Performance Factors

SCSI vs. IDE: For machines with heavy disk and/or CPU loads, SCSI is superior to IDE. A single system generally supports far larger numbers of SCSI disks than IDE; this can also be a consideration. With good host adapters, SCSI driver overhead is lower and 'disconnect' (the ability to issue a command to a drive and then 'disconnect' to issue a command to a different drive) is a big win. For machines that need only one or two disks and that have CPU cycles to spare, the lower cost of IDE is attractive.

Drive RPM: Rotational speed determines how fast the data moves under the heads, which places the upper bound on transfer speed. Today's really fast drives spin at 10,033 RPM (Seagate Cheetahs) and deliver about 15 MB/sec on the outside tracks. Last year's fast drives spun at 7200 RPM; inexpensive drives in use today spin in the 3600 to 5400 RPM range.

Average seek time: The average interval for the heads to move from one cylinder to another. Typically, this is quoted as about half of the maximum (inside track to outside track) seek time. Lower average seek times are very desirable. Currently, a 7-8 ms average seek time is really good, 8-10 ms is typical and much more than 10 ms starts to impact loaded system performance. Unix-like systems perform many seeks between the inode blocks and actual file data. To ensure file system integrity, writes of significant inode data are not cached, so these seeks are a significant part of the expense of operating a disk. In general, if a tradeoff must be made, a lower average seek time is better than a higher rotational speed.

The price of fast seek times and high RPMs is increased heat generation; take care to keep them cool or you will sacrifice performance for reliability. Use plenty of fans and make sure that there is good air flow through the box, or put the disks in their own enclosure(s).

On drive caches allow the drive to buffer data and therefore handle bursts of data in excess of media speed. Most modern drives have both read and write caches that can be enabled separately. The read cache speeds read operations by remembering the data passing under the heads before it is requested. A write cache, on the other hand, can cause problems if the drive claims that the data is on the disk before it is actually committed to the physical media (although some drives claim to be able to get the data to physical media even if power is lost while the data is still in the cache). Understand your hardware fully before enabling such a feature.

SCSI Tagged Command Queueing enables multiple commands to be issued serially to a single drive. Tagged command queueing increases disk drive performance in two ways: it enables the drive to optimize some mixes of commands overlap command decoding with command processing.

Avoiding Seeks

Some of best ways to improve performance involve reducing the number of operations to be performed. Since seek operations are performance eaters and relatively common, avoiding seeks can improve performance. Here are some methods to reduce the impact of disk seeks.

Use Multiple Spindles

Systems that concurrently access multiple files can improve their performance by putting those files on separate disk drives. This avoids the expense of seeks between files. Any busy server that maintains logs would do well to separate the log files from the server's data.

If you must store several Unix-style partitions on a large disk, try to put the most frequently used partition (swap or /usr) at the middle of the disk and the less frequently used ones (like home directories) at the outside edges.

Turn Off Access Time Updating

If maintaining a file's most recent access time (i.e., the time someone read it – not the time someone wrote it) is not critical to your site and, additionally, your data is predominantly read-only, save many seeks by disabling access time updating on the file system holding the data. This particularly benefits news servers. To turn off access time updates on BSD/OS, set the "noaccesstime" flag in the /etc/fstab file for the file system in question:

```
/dev/sp0a /var/news/spool
     ufs rw,noaccesstime 0
```

The same functionality exists in other Unix variants, but the names have been changed to confuse the innocent.

Use RAM Disk For /tmp

Many programs use the /tmp directory for temporary files and can benefit greatly from a faster /tmp implementation. On systems with adequate RAM, using 'RAM disk' or other in-memory file system can dramatically increase throughput. BSD/OS and other 4.4BSD derivatives support MFS, a memory file system that uses the swap device as backing store for data stored in memory. The following command creates a 16 MB MFS file system for /tmp:

```
mount -t mfs -o rw -s=32768 \
     /dev/sd0b /tmp
```

Or you could put this line in /etc/fstab:

```
/dev/sd0b /tmp mfs rw,-s=32768 0 0
```

The data stored on a MFS /tmp is lost if /tmp is unmounted or if the system crashes. Since /tmp is often cleared at reboot even when the file system is on disk, this does not seem to represent a significant problem.

Increasing Disk and SCSI Channel Throughput

Size The Buffer Cache Correctly

The best solution for disk bottleneck is to avoid disk operations altogether. The in-memory buffer cache tries to do this. On most 4.4BSD-derived systems, the buffer cache is allocated as a portion of the available RAM. By default, BSD/OS uses 10% of RAM. For some sites, this is not enough and should be increased. However, increasing the size of the buffer cache decreases the amount of memory available for user processes. If you increase the size of the buffer cache enough to force paging, all will be for naught.

Few systems support tools to report buffer cache utilization.

Use Multiple Disk Controllers

Adding extra disk controllers can increase throughput by allowing transfers to occur in parallel. Multiple disks on a single controller can cause bus contention, and thus lower performance, when both disks are ready to transfer data.

Striping (RAID 0)

RAID 0 distributes the contents of a file system among multiple drives. The result (when properly configured) is an increase in the bandwidth to the file system. The downside is that increasing the number of drives and controllers used to support a file system reduces the MTBF; RAID 0 does nothing to defend against hardware failures.

Higher Levels of RAID

RAID 1 (mirroring) provides the same write performance as RAID 0 at the expense of redundant disk drives (and, hence, higher cost per storage unit). Where performance is a major constraint and file system activity consists of many small writes (e-mail, news), RAID 1 may be the way to go. Of course, the redundancy can dramatically increase mean time to data loss.

Where read access dominates a file system's activity, RAID levels 3 (dedicated parity disk) and 5 (rotating parity disk) offer reliability and performance that is not significantly worse than RAID 0. RAID 3 and 5 offer reasonable write performance on large (relative to the stripe) files.

At the very high end, RAID hardware is implemented with a high degree of parallelism and sustained throughput can approach, or exceed, that of the system bus. Note, however, that write performance (particularly for smaller blocks) can be poor.

Monitoring and Tuning Disk Performance

The *iostat*(8) command can provide some help with optimizing disk performance. For each drive, it reports three statistics: sps (sectors transferred per second), tps (transfers per second), and msps (milliseconds per "seek," including rotational latency in addition to the time to position the heads). The accuracy of these numbers, especially msps, varies considerably among implementations.

The *tunefs*(8) command is used to modify the dynamic parameters of a file system. Disks whose contents are primarily read and that exploit read caching will often benefit from setting the rotational delay parameter to zero:

```
# tunefs -d 0 file system
```

When adjusting file system parameters with *tunefs*(8), remember that only new contents are affected by the change. Since the existing contents of the file system may constrain the allocation algorithms, it is best to use a scratch file system while experimenting with *tunefs*(8) options.

RAM

Two major performance factors related to RAM are:

- Avoiding paging (make sure that there is enough memory in the system to handle your load)
- Ensuring that enough memory has been allocated to the kernel.

Sizing Main Memory

If your system is short on main memory, it will end up swapping and the speed of your memory subsystem will degrade to the speed of your disk (usually about three orders of magnitude difference). The most useful tools for diagnosing paging problems are *vmstat*(8) and *pstat*(8). The 'po' column of *vmstat*(8) shows you 'page out' activity. Any value other than 0 on anything more than an occasional basis means that performance is suffering.

As you increase the amount of main memory, you are also increasing the likelihood of memory errors. The current crop of 64 MB SIMMs are especially prone to errors. You should plan on running Error Correcting Code (ECC) memory if your hardware supports it (and you should only be buying hardware that does!). Current ECC implementations on PC hardware cost about 5% in memory bandwidth when accessing main memory (and the CPU tends to go to main memory less frequently than one might expect).

When sizing a system's RAM, it is often useful to know the memory usage of a typical user or a particular process. On BSD derivatives, you can get some of this information with 'ps -avx.'

The RSS ('resident set size') column reports the number of kilobytes in RAM for a process. Since the shared libraries and the text are common to all of the instances of a program (and in the case of the shared libraries to all of the users of the library), you can't use this information to determine the amount of memory that would consumed by an additional instance of the program. It does give an upper bound for making estimates.

The TSZ ('text size') column reports the size of the program's text, but it does not tell you how much of the text is resident (i.e., how many pages are not in the swap area).

After observing these parameters for a while, you can get a pretty good idea of how much memory a particular process typically uses. It may also be worth watching the numbers while you subject a process to both typical and extraordinary loads. Using this information, you can then estimate how much memory you would need to support a particular mix of applications. Perl and cron can help to automate this task.

Cache Size

Currently, tools don't exist to directly monitor real life cache performance. Within reason, more cache is better. Run real-life benchmarks to see if different hardware can improve your site's performance.

Kernel Memory Tuning

Most modern kernels can be tuned for increased loads using only a "knob" or two, but for some applications you will need to do additional tuning to get peak performance.

On BSD/OS, the size of kernel memory has a default upper bound of 248 MB. Major components of kernel memory are the buffer cache (BUFMEM) and the mbufclusters (NMBCLUSTERS). Both are in addition to the memory allocated by KMEMSIZE. On most systems, these memory allocations are completely adequate, but if the size of the buffer cache is increased beyond 128 MB it may get tight.

The limit on kernel memory can be increased by modifying the include file machine/vmlayout.h (with suitable knowledge of the processor architecture in question) and then recompiling the entire kernel. In addition to the kernel, you'll also need to recompile libkvm (both the shared and static versions), gdb, ps and probably a few other programs.

maxusers

On BSD systems of recent vintage, the place to start kernel memory tuning is with the 'maxusers' configuration parameter. The maxusers parameter is the "knob" by which the kernel resources are scaled to differing loads. Don't think of this in terms of actual numbers of humans, but just as a knob that can request "more" or "less."

As a rough rule of thumb, you can start by setting maxusers to the size of main memory (e.g., for a system with 64 MB of main memory start by setting maxusers to 64).

maxbufmem

The kernel variable maxbufmem is used to size the buffer cache in systems without a unified user space/buffer cache. A zero value means "use the default amount (10% of RAM)." Otherwise, it is set to the number of bytes of memory to allocate to the cache. You can set this at compile time:

```
options "BUFMEM=\(32*1024*1024\)"
```

Or you can patch the kernel image (with *bpatch*(1) or a similar tool):

```
# bpatch -l maxbufmem 33554432
```

and reboot. It is not safe to change the size of the buffer cache in a running system.

A system running a very busy news or web server is an obvious candidate for increasing the size of the buffer cache.

When increasing the size of the buffer cache, the memory available for user processes is decreased. Ideally, a tool would report buffer cache utilization. Unfortunately, such a tool doesn't seem to exist, so tuning is sort of hit-or-miss – increase the size of the buffer cache until you start paging, then back off.

mbufs and NMBCLUSTERS

In the BSD networking implementation, network memory is allocated in mbufs and mbuf clusters. An mbuf is small (128 bytes). When an object requires more than a couple of mbufs, it is stored in an mbuf cluster referred to by the mbuf. The size of an mbuf cluster (MCLBYTES) can vary by processor architecture; on Intel processors, it is 2048 bytes.

The number of mbufs in the system is controlled by their type allocation limit (reported by vmstat -m). The configuration option NMBCLUSTERS is used to set the number of mbuf clusters allocated for the network. The default value for NMBCLUSTERS in the kernel is 256. If you have GATEWAY enabled, NMBCLUSTERS is increased to 512. Systems that are network I/O intensive, such as web servers, might want to increase this to 2048.

On recent BSD systems this parameter can be dynamically tuned with *sysctl*(8):

```
# sysctl -w net.socket.nmbclusters=512
```

The upper bound on the number of mbuf clusters is set by MAXMBCLUSTERS. Usually MAXMBCLUSTERS is set to 0 and the limit is calculated dynamically at boot time.

If a BSD kernel runs out of mbuf clusters, the kernel will log a message ("mb_map full, NMBCLUSTERS (%d) too small?") and resort to using mbufs. This can also be seen by an increase in the number of mbufs reported by vmstat -m.

KMEMSIZE

The size of kernel memory (aside from the buffer cache and mbuf clusters) is set by KMEMSIZE. Normally, it is scaled from "maxusers" and the amount of memory on the system, but it can also be set directly.

The default KMEMSIZE is 2 MB. At a maxusers value of 64 (or if there is more than 64 MB of memory on the system), KMEMSIZE will increase to 4MB. At a maxusers of 128, KMEMSIZE will increase to 8MB. Beyond that, use the KMEMSIZE option (in the kernel configuration file) to increase the kernel size.

Routing Tables

To run a full Internet routing table in the Spring of 1997, it is necessary to increase the amount of kernel memory to at least 16 MB. The BSD/OS config file for the GENERIC has notes on this:

```
# support for large routing tables,
# e.g., gated with full Internet
# routing:
# options "KMEMSIZE=\(16*1024*1024\)"
# options "DFLDSIZ=\(32*1024*1024\)"
# options "DFLSSIZ=\(4*1024*1024\)"
```

Since the gated process is also likely to get quite large, it also makes sense to increase the default process data and stack sizes.

The *vmstat*(8) can give an overwhelming amount of information about kernel memory resources. Some points worth noting: In the "Memory statistics by type" section, the "Type Limit" and "Kern Limit" columns report the number of times the kernel has hit a limit on memory consumption. Entries that are nonzero bear some investigation. These statistics are collected since system boot, so you'll probably want to look at the difference between two runs that bracket an interesting load. Many of these limits scale with maxusers, so a quick experiment can be done by recompiling the kernel with a higher maxusers value.

Network Performance

For network performance analysis, *netstat*(8) is the command you want. The raw output of a single *netstat*(8) run is not terribly useful, since its counters are initialized at boot time. Save the output in a file and use diff or some fancy perl script to show you the changes over an interval.

For understanding why your network performance sucks, *tcpdump*(1) or some other sniffer/protocol analyzer is the tool of choice. Data collected with *tcpdump*(1) can be massaged with perl (or sed and awk) to make it easier to see what is going on.

Too Much Traffic

Take time to check that you've only got the traffic you expect on the network. Look for things like DNS traffic (if you have much, configure a local cache), miscellaneous daemons you don't need (not only are they costing you network traffic, they are probably costing you context switches), etc. If you're really trying to squeeze the last little bit out of the wire, consider using multi-cast for things like time synchronization. Use *tcpdump*(1) to see what kind of traffic there is. Watch for unexpected activity, or unexpected amounts of activity, on your hubs. Compare interface statistics with similar machines.

Too Little Bandwidth

With Ethernet, as the amount of traffic on the network increases, the instantaneous availability of the network decreases as the hardware experiences collisions. Use 'netstat -I' to monitor the number of collisions as a percentage of the traffic on the wire. Ethernet switches can reduce the number of hardware collisions and increase apparent number of segments. Full duplex support (e.g., 10baseT) can also help.

Errors

Significant error rates (anything more than a fraction of a percent) are often an indication of hardware problems (a bad cable, a failing interface, a missing terminator, etc.).

Use 'netstat -I' to monitor the error rate.

Timeouts and Retransmissions

Don't forget to watch the activity lights on your interfaces, hubs, and switches. Look for unexpected traffic, pauses, etc. and then get busy with *tcpdump*(1) to see what is going on. This paper will cover some very basic concepts; for more detail, see Richard Steven's excellent book, *TCP/IP Illustrated, Volume 1* [4].

Buffer Sizes

Network buffer sizes can have a significant impact on performance. With FDDI, for example, it is often necessary to increase the TCP send and receive space in order to get acceptable performance.

On BSD/OS and other 4.4 BSD derivatives this can be done with *sysctl*(8); see Listing 1. To a first approximation, think of net.inet.tcp.recvspace as controlling the size of the window advertised by TCP. The net.inet.tcp.sendspace variable controls the amount of data that the kernel will buffer for a user process.

As a general rule of thumb, start sizing the kernel buffers to provide room for five or six packets "in flight" (one packet for the sender to be working on, one packet on the wire, one packet being processed at the receiver – then multiply by two to account for the returning acknowledgements). Since Ethernet packets are 1500 bytes (or less), 8 KB of buffer space is about right. For FDDI, with 4 KB packets, 20 to 24 KB is likely to be necessary to get FDDI performance to live up to its potential.

Another way to think about this is in terms of a "bandwidth delay product:" multiply bandwidth times the round trip time (RTT) for a packet and specify buffering for that much data. If you want to be able to maintain performance in the face of occasional lost packets, figure the bandwidth delay product as:

buffer bytes = bandwidth bytes/second *
(1 + N packets) * RTT seconds

Where N is the number of lost packets you want to be able to sustain without losing performance. This is a bit harder to calculate as you have to be able to measure RTT.

Increasing the size of the kernel send buffers can improve the performance of applications that write to the network, but it can also deplete kernel resources when the data is delivered slowly (since it is being buffered by the kernel instead of the application).

The size of application buffers should also be considered. If the buffers are too small, considerable additional system call overhead can be incurred.

Using netstat(8)

netstat -I

Using 'netstat -I' yields a basic report on the amount of traffic and the number of errors and collisions seen by a network interface. Here are some of the statistics reported:

- **Input Errors** tells you that a bad packet was received – something is wrong somewhere between you and the sender.
- **Output Errors** means something is wrong with the local hardware (interface card, cables, etc.).

```
# sysctl -w net.inet.tcp.sendspace = 24576
# sysctl -w net.inet.tcp.recvspace = 24576
```

Listing 1: Changing network buffer sizes.

If your hardware supports full duplex operation, another possibility is that one end is configured for full duplex while the other is not.

- **Collisions** tells you how busy the network segment you're attached to is. Lots of collisions (say more than 10%) on a regular basis tell you it's time to think about reworking your network architecture.

Another use of 'netstat -I' is tracking down lost packets. Say you are trying to ping a remote machine and are not receiving any replies. You can track down the location of the problem by observing the input and output packet counts of the machines along the way. Each machine that processes the packet will increment the appropriate counters. On the machine where the packet was "lost" you would expect to see an increasing error count, either on input or output.

netstat -s -I

Specifying 'netstat -s -I' yields more detail on the traffic seen by the interface, including the number of dropped packets, the number of octets (bytes) sent and received (which, along with the packet count, enables you to compute average packet size) and data on multicast traffic.

netstat -s

With 'netstat -s,' you get voluminous statistics on the whole networking subsystem. These statistics are maintained over the life of the system, so to see current trends, look at a pair of reports and compare the numbers. A quick and dirty way to do this is to save the output into a pair of files and use diff. Of particular interest are the counts of:

- **Dropped packets** indicates that the local machine is receiving network traffic but does not have the resources to handle that traffic. The two most common causes are: running out of mbufs or running out of CPU power. Another possibility is that you've run out of mbuf clusters and the kernel is substituting mbufs, but the system is marginal on CPU power and the extra processing involved has pushed it over the edge.
- **Fast Retransmits**. The TCP spec requires that an immediate ACK be sent when a packet is received out of order. This ACK will duplicate an ACK that has already been sent for the last in-order data that the receiver received. When the sender has seen three duplicate ACKs, it assumes that the next packet has been lost and it will retransmit the packet without waiting for a retransmit timeout. This is a "fast retransmit" and, if the receiver's window is big enough, it will prevent TCP from waiting for a timeout when a packet is lost.
- **Duplicates**. Duplicate packets are packets that arrived twice. Usually this means that an acknowledgement was delayed and the sender retransmitted.
- **Retransmit Timeouts**. The retransmit timer expired while we were waiting for an ACK.
- **IP Bad Header Checksums and TCP Discarded For Bad Checksums**. Packets that were "damaged in transit" should be thrown out by the link layer checksum and not seen by any of the higher layers. This is a possible sign that an intermediate gateway is corrupting packets. If you are seeing significant numbers of these, it might be worth the trouble to try to track down the source. To do this, you will need to probe methodically, starting with the nearest gateways (routers) and working outward.
- **UDP with no checksum**. These are interesting because running UDP without checksums is asking for trouble. If you are seeing many of these, it would be worth tracking down the source.
- **UDP with no socket** means that you're getting UDP traffic, but nobody is listening for it. At the very least, this is a waste of resources, and it may be indicative of a configuration error or perhaps even something malicious. Track the source using *tcpdump*(8) if many packets show up here.

When the network is performing well, all of the above statistics should represent a small fraction of the traffic that the system sees (well under a fraction of a percent).

netstat -a

With 'netstat -a,' you get information on all of the active network connections on the system. Things of interest include:

- **The number of connections.** This gives you a feel for the amount of network traffic the system is seeing.
- **Connections with with data in the Send-Q or Recv-Q**. You will see this occasionally, but significant numbers of connections with queued data would indicate that something is amiss.
- **Large numbers of connections in states other than ESTABLISHED.** On a busy web server, many connections will be in the TIME_WAIT state. But, observing significant numbers of connections in SYN_SENT (indicative of a SYN-flood attack) or FIN_WAIT_2 (waiting for a FIN from the other side after we've sent our FIN, sometimes this indicates a kernel bug) bear investigation.

netstat -m

With 'netstat -m,' you get a report on the memory usage of the networking subsystem. When investigating performance problems, the counts of "requests for memory denied/delayed" and of "calls to the protocol drain routines" are of particular interest. They are indications that the network is running low (or out) of memory.

Using tcpdump(1)

The *tcpdump*(1) program is a powerful tool for analyzing network behavior and is available on most Unix-like systems. With *tcpdump*(1), you can watch the network traffic between two machines. The packet contents are printed out in a semi-readable form (which is easy to massage with perl to help you see the info you're interested in). Each packet is printed, along with a time stamp, on a separate line.

One of the most useful features of tcpdump is the ability to filter traffic. For example, if you're curious about stray traffic on the network with your web servers, you could run something like this:

```
# tcpdump -i de0 not port http
```

If you had connected to the machine by telnet you might use:

```
# tcpdump -i de0 not port http \
     and not port telnet
```

The output from tcpdump would only show non-http and non-telnet traffic, thus enabling you to focus on the traffic that you're interested in.

You can save the data for later analysis by having tcpdump write it to a file:

```
# tcpdump -i de0 -w tcpdump.out
```

When you examine the data later, you'd use:

```
# tcpdump -r tcpdump.out not \
  port http and not port telnet
```

Aside from making analysis of the data more flexible, this approach also consumes fewer machine resources, so you're less likely to drop packets or otherwise impact the system under test. By default, tcpdump only grabs the first 76 bytes of the packet (which is enough to get the headers, but you won't get much of the packet content). You can add '-s 1600' to save the whole packet.

One of the most common things that you'll be looking for is an explanation for why the network is pausing every so often.

Here is a recipe for examining tcpdump output:

1. Collect the data, and post-process it if you want to (for example, you may want to convert the time stamps into relative intervals).
2. Browse through the tcpdump output with your favorite editor, looking for places where the timestamp jumps by more than the "usual" amount.
3. You're probably in the vicinity of a dropped packet, and the timestamp is due to the timeout before the packet was retransmitted.
4. Take a quick look around and then start looking for the next timestamp jump. What you're looking for is a pattern. If the network behavior was bad enough to get you looking at packet traces there is almost certain to be one.
5. After some looking, a pattern will emerge. If you don't know what the problem is by now, you'll need to find somebody who knows more about the protocol in question. But, armed with a description of the problem, the tcpdump output, and the patterns you found, it should be easier to get some help.
6. As you're looking at the jumps in the tcpdump output, an important part of the story is the size of the jumps. Are they the same? Are they increasing? There are people who know this stuff well enough to make a good guess as to the source of the problem just by the size of the delay.

In some cases, you will need to arrange for the machine running *tcpdump*(1) to be in the middle of the wire between the machines that are having problems. With some switches this is not possible, and you will either need to substitute a hub (you can also just add a hub at one of the switch ports and connect your monitor and one of the machines under test to the hub) or run tcpdump on the machines at both ends of the link and analyze both sets of data. The hub trick doesn't work if you're trying to look at a link that is running full duplex.

CPU

In performance analysis, you're usually up against one of two things: either CPU utilization is too high or else it's too low. Too high means that you've got to figure out a way to free up more cycles in order to get more work done. Too low means that you've got to figure out what you're waiting on, usually some I/O device.

Start CPU analysis with a tool that will tell you how your CPU is being utilized: *top*(1), *vmstat*(8),) and *iostat*(8) will all tell you how much of your CPU is being used and whether the cycles are being spent in user or kernel (system) code. If most of the CPU's time is being spent executing user code, *top*(1) and *ps*(1) will help you identify the processes of interest. If your CPU is close to 100% utilized, you need to figure out what it's doing. Start by identifying the process or processes that is using most of the user time.

Even if the CPU is spending most of its time executing in the kernel (high percentage of system time), these processes are likely to be the cause. Once the processes in question have been identified, *ktrace*(1) can be a very useful tool for figuring out what is going on. So are the 'in' (interrupts), 'sy' (system calls) and 'cs' (context switches) columns in the output of *vmstat*(8). You should be developing a rough idea of the performance bounds of the systems you support. This is the information you will need to read the output of commands like *vmstat*(8).

Very simple benchmarks (say 10 lines of code) can help you calibrate your expectations of the machine's performance. For example: A 200 MHz P6 can perform about 500 fork/exit/wait (i.e., null fork with two context switches) operations per second. It can also perform about 400,000 getpid() operations per second (getpid() is arguably the most trivial system call). From these benchmarks, you can make an educated guess that a process that is fork()ing 10s of times per second should not be putting too much of a load on the system due to the forks, but a process that is doing 100s of fork()s per second may well benefit from a redesign.

The *ktrace*(1) program can give insight into the behavior of the processes at the system call level. Since system calls are relatively expensive, this can be very useful. Often the information revealed by ktrace will be a surprise, revealing details of the program's operation that were neither documented nor intuitive.

For example, some (maybe all) versions of Apache search from the root for a .htaccess file. If you're not using these files, turning off the check can get you a 5 to 10% increase in performance. If you are using them, you can decide if it would be worth the trouble to use a "shallower" directory hierarchy. This is also a great way to figure out error messages (like "file not found" with no file name or without a full path).

Other things to look for are frequent read() and write() calls or such calls with small byte counts, frequent system calls of any type (perhaps the application can cache the data), etc.

If all else fails, you may have to resort to profiling your application or your kernel. Typically this means that you have to either have source code or vendor cooperation. Enabling profiling usually means you need to recompile. You can get clues to the application architecture, and determine if the problem lies in the application or in the system libraries.

If the problem appears to be in the kernel, profiling can be a very useful tool. Even if you don't do kernel work, useful information can be gained. For example, if you discover that a significant amount of time is being spent in a device driver, you might want to try a different device. If you have a cooperative OS vendor, being able to tell them the location of your kernel's hot spots might be just the incentive they need to work on that particular code.

The *time*(1) command can help you distinguish between CPU under-utilization due to outside delays (disk or network) and having an excess of CPU capacity. When time reports a wall clock (real) time that is significantly longer than the combined user and system times, you should suspect that you're seeing an I/O (or memory) problem. Small variances are to be expected due to other processes being given time to run.

Some types of loads will result in misleading numbers from *vmstat*(8) and its friends. For example, on a gateway, the load of handling the network interfaces will not be reported by *top*(1) or *vmstat*(8). In such cases a "cycle soaker" can be a useful thing. It will tell you how much CPU is actually available for computation.

Profiling User Code

If most of your CPU time is being spent in user mode and *ktrace*(1) doesn't give you enough information to improve the situation (or to place the blame and move on), then profiling the program in question might be the way to go. To profile the application you (or the vendor) will have to recompile the code. If this is not possible, you will need to choose another strategy.

The procedure for profiling user code is slightly different than that for profiling the kernel:

- Arrange to run the C compiler with the -pg flag. Probably the easiest way to do this is by adding it to the CFLAGS in the Makefile for the program. You may also need to arrange to link statically.
- Run the program. When it terminates, the file gmon.out will be produced in the directory where program was run. You won't get gmon.out until the program terminates.
- Display the profiling data with *gprof*(1). The profiling data will not include kernel time spent on the process's behalf. It may be necessary to do a bit of piecing together evidence from kernel and user profiling in order to get a full picture of where your cycles are going.

One of the challenges of profiling servers is to get the server to run in a mode where you can collect useful data. Servers that fork to handle requests will have one instance of the server process that will typically run for the life of the system, forking a new instance each time a request comes in. In this case, two different threads of execution will need to be profiled. The easy one to profile is the "master" instance that accepts the incoming requests and then forks to handle them. The processes created to handle the requests are harder to profile, and due to their short life may not produce statistically valid data. To get around this you may have to figure out a way to get the server to handle requests directly without forking.

Profiling the kernel

Typically, this is only interesting when you're out of CPU and want to figure out where the cycles went **and** you're really sure that the kernel is at fault. To do this, you'll need to build a profiling kernel and then reboot with that kernel. Profiling can be turned on and off, allowing you to collect data representative of the load you are interested in. Running a profiling kernel does add a bit of overhead to the system.

Here are the steps for doing this on BSD/OS:

- Configure and compile a profiling kernel:

```
# cd /usr/src/sys/i386/conf
# config -p MY_KERNEL
# cd ../../compile/MY_KERNEL.PROF
# make depend all
```

- Install the new kernel and reboot.
- When you are ready to collect profiling data, enable profiling with:

```
# kgmon -rb
```

- Run your load.
- When you're done collecting data, disable profiling with:

```
# kgmon -h
```

- Then dump the collected data with:

```
kgmon -p
```

The data is left in a file called gmon.out in the current directory.

- The profiling data is display by *gprof*(1). gprof has almost as many options as ls, but the following should work pretty well for starters:

```
# gprof /bsd gmon.out
```

You may want to redirect output to a file so that you can use perl to help you make sense of the data.

For more details, take a look at the *config*(8), *kgmon*(8) and *gprof*(1) man pages.

Typically, what you're looking for in the profiling data is a rough idea of where the kernel is spending its time. Is it a device driver (which one)? The file system? The network, etc.? Often the answer is surprising. What you do next depends on where the data leads you and the amount of kernel hacking you want (or are allowed) to do.

For example, if the data points to the Ethernet driver (as in the first example below), you will either want to experiment with a different card or dig deeper into the profiling data to see if you can find some room for improvement in the driver (or suggest that your OS or Ethernet vendor do the same). On the other hand, if the data points you to fork(), the place to look is at your load.

Kernel Profiling Example

See Figure 1 for sample kernel profiling output. The data is from a profiling run on my laptop with a heavy load of network traffic. The Ethernet interface is a 3com 3c589B which uses the ef driver.

Looking at the data, one of the first things to notice is that the efintr() routine is right up near the top, so we can pretty reasonably focus our interest on the Ethernet interface. One of the next things to notice is that most of the time in efintr() is attributed to insw(). insw() is part of the machine dependent assembler code in locore.s, doing signed 16 bit reads from the Ethernet interface. This is not surprising; since the 3c589B uses Programmed I/O (PIO), it is also a reasonable bet that insw() was written with considerable concern for efficiency. In this case, it looks like the right thing to do is to try a different Ethernet card, looking for one that does DMA. NH Code Availability

A collection of useful benchmarks (and pointers to more) can be found at ftp://ftp.bsdi.com/bsdi/benchmarks. Source code for tcpdump can be found at ftp.ee.lbl.gov.

```
                                 called/total    parents
index  %time    self descendants  called+self    name           index
called/total    children
                                                 <spontaneous> [1]     53.0
0.19    85.75                          doreti [1]
                0.19    80.64            97968/97968     _isa_intrswitch [2]
                0.76    4.02             22237/43750     _ipintr [11]
                0.00    0.14              2264/3158      _softclock [41]
                0.00    0.00                60/971       _trap [53]
                0.00    0.00                 1/1 _arpintr [403]
-----------------------------------------------
                0.19    80.64            97968/97968     doreti [1]
[2]     49.9    0.19    80.64            97968         _isa_intrswitch [2]
                25.63   54.99            97893/97893     _efintr [3]
                0.00    0.01                75/75        _wdcintr [91]
-----------------------------------------------
                25.63   54.99            97893/97893     _isa_intrswitch [2]
[3]     49.8    25.63   54.99            97893         _efintr [3]
                53.68   0.00 12626901/12626907    _insw [5]
                0.74    0.00             97893/97893     _ef_newm [24]
                0.56    0.01             97893/97893     _ether_input [27]
```

Figure 1: Kernel profiling output.

Author Information

Douglas Urner has been working with Unix-like systems for 15 years. He was a project leader in the Small Systems Support Group at Tektronix (when a VAX 780 was a "small" system). Today he is a member of the technical staff at Berkeley Software Design, Inc. (BSDI) where he mediates the battle between good and evil (or is that engineering and sales?).

References

[1] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz and Kurt J. Worrell. *A Hierarchical Internet Object Cache.* Technical Report 95-611, Computer Science Department, University of Southern California, Los Angeles, California, March 1995. Also Technical Report CU-CS-766-95, Department of Computer Science, University of Colorado, Boulder, Colorado, http://harvest.transarc.com/afs/transarc.com/public/trg/Harvest/papers.html#cache .

[2] Aaron Brown and Margo Seltzer, "Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture," *Invited Talk Notes of the 1997 USENIX Conference,* Anaheim, CA, http://www.eecs.harvard.edu/vino/perf/hbench.htm .

[3] Steven Hinkle, "Tuning INN News on BSD/OS 3.0," http://www.bsdi. com/ .

[4] W. Richard Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison Wesley, 1994.