

# Successful Solaris™ Performance Tuning

*Ken Gottry*

In real estate, the three top success factors are location, location, location. In today's distributed, multi-tiered, Web-based systems, the three top success factors are performance, performance, performance. It's great to design an elegant system and to use the latest technology, but if the system doesn't perform well, then you and your client aren't successful. In this article, I share some of my secrets for success.



I tackle performance problems in a specific order. First, I look for network problems, because they can limit the amount of work reaching a server. Next, I look for memory problems, because they can impact I/O performance. I then look for I/O problems, because they can cause CPU degradation. And finally, I look for CPU problems, because they can cause memory problems. In this article, I present three real-world performance problems: one network, one memory and CPU, and one I/O. I will explain how I modified Solaris, and why.

## Solaris Tunable Parameters

Solaris is self-adjusting to system load and normally demands minimal tuning. In some cases, however, tuning is necessary. By changing the setting of one or more kernel variables, you can affect the performance of Solaris -- sometimes positively, sometimes negatively. Be sure to thoroughly research a variable before changing its settings. Also, be sure to experiment with changes on a test server where the only one impacted by mistakes is you.

Another thing to remember is that one system's **/etc/system** settings might not be applicable, either wholly or in part, to another environment. Carefully consider the values in the file with respect to the environment in which they will be applied. Make sure that you understand the behavior of a system before attempting to apply changes to system variables. The changes described in this article may not be appropriate for your system. However, the logic behind my troubleshooting and my explanation of Solaris behavior should prove helpful for most systems.

## Network Bottleneck

In the first scenario, I had a computer running iPlanet Web server with an NSAPI plug-in. The NSAPI module accepted JSPs and routed them to another computer running an application server. I found during a stress test that the computer running the Web server was dropping incoming requests. There were two possible answers to why the Web server computer couldn't handle the workload -- either it was undersized, or it was not properly tuned.

## How I Investigated the Problem

I ran **netstat -s -P tcp** and noticed that the **tcpListenDrop** value was nonzero. Because the **tcpListenDropQ0** and **tcpHalfOpenDrop** values were zero, I didn't suspect a SYN flood Denial of Service attack. When I ran **netstat -na -P tcp**, I noticed thousands of sessions in **TIME\_WAIT** and **FIN\_WAIT\_2** state. I used the **ndd /dev/tcp tcp\_conn\_hash\_size** command to determine the size of the TCP connection hash. I noticed that the hash size was set to the default of 256 entries.

## What Those Numbers Mean

To address the problem, I had to understand what happens when a TCP connection is closed. [Figure 1](#) illustrates these steps. After the Web server returns the requested Web page to a browser, it sends a connection termination

request (**FIN**) to indicate that its end of the connection has been closed. When the browser acknowledges (**ACK**) the Web server's **FIN**, the Web server enters **FIN\_WAIT\_2** state. The browser then sends a connection termination request (**FIN**) to the Web server indicating that it has closed its end of the connection. The Web server acknowledges (**ACK**) the browser's **FIN**. The Web server enters **TIME\_WAIT** state and stays that way for **time\_wait** interval. During this period, the kernel data structure resources associated with the TCP connection remain assigned. This is done in case any straggling packets are lingering on the network.

This delay introduces a problem. Solaris consumes CPU time searching through closed connections looking for the kernel data structures associated with the open TCP connection every time a packet arrives. As the search takes longer and longer, it is possible for incoming requests to be refused (nonzero **tcpListenDrop** values).

Solaris uses a connection hash to quickly locate kernel data structures associated with TCP connections. The **tcp\_conn\_hash\_size** defaults to 256 entries. When Solaris has more than 256 connections, the hash is bypassed and a linear search of memory is required to locate the appropriate TCP data structure. Entries remain in the hash even when the connection is closed and is in the **TIME\_WAIT** and **FIN\_WAIT\_2** state. To illustrate the impact of this point, consider an example. At 100 connections per second, the default **TIME\_WAIT** parameter of 240000 ms means there are 24,000 closed, but not released, connections still being maintained in TCP data structures within the kernel.

## What Parameters I Changed and Why

I changed the **TIME\_WAIT** interval from the default of 240000 to 60000. This value releases the socket one minute after it is closed. I changed the **FIN\_WAIT\_2** interval from the default of 675000 to 67500 (one less zero). This releases the socket resources after about one minute even if the browser's **FIN** has not been received. I changed the connection hash size from the default of 256 to 8192. This value allows Solaris to quickly locate the TCP data structure associated with up to 8-K connections.

## How I Changed the Parameters

Some TCP parameters can be changed on the fly with the **ndd** command. These parameters revert to their original values during the next reboot. Other parameters can only be changed by adding lines to **/etc/system** and rebooting. I decided to skip the on-the-fly testing. I added the line shown in [Listing 1](#) to **/etc/system** and to **/etc/rc2.d/S69inet** and rebooted. (Prior to Solaris 7, the **TIME\_WAIT** parameter was incorrectly named **tcp\_close\_wait\_interval**.)

## One More Battle Scar

The above analysis dealt with passive opens (i.e., TCP connections initiated by the browser) and I also had to deal with problems associated with active opens (i.e., TCP connections initiated by the Web server). As I mentioned, the iPlanet Web server had an NSAPI plug-in that routed JSPs to the application server running on another computer. The NSAPI module did not support TCP connection pooling, instead it opened a new TCP connection (active open) to the app server for every JSP. Once the app server had responded, the NSAPI module closed the TCP connection. Thus, the Web server computer was closing TCP connections in two directions, as shown in [Figure 2](#). The ones between the Web server and browser remained in **TIME\_WAIT** state while the ones between the Web server computer and the app server remained in **FIN\_WAIT\_2** state, never receiving the **FIN** from the app server.

The default setting for **FIN\_WAIT\_2** interval is 675,000 ms, which means that the anonymous ports used for connections to the app server remain assigned for more than ten minutes after they are closed awaiting the **FIN**. At 100 connections per second, that means that 67,500 anonymous ports are allocated but not being actively used. That's an impossible condition since there are no more than 32,000 anonymous ports available on Solaris unless tunable parameters have been altered.

I never figured out if the lost **FINs** (i.e., the **FIN\_WAIT\_2** states) were due to something in the configuration of the internal firewall or in the way the application was coded. However, I found that changing the **tcp\_fin\_wait\_2\_flush\_interval** helped.

## Memory/CPU Bottleneck

The second scenario I will describe involved a Sun E3500 with 3 GB of memory. There were 100 users who **telneted** to the server to run a broker application. As business grew, the number of users increased to 150 and the server couldn't keep up with the workload.

## How I Investigated the Problem

I started by running **vmstat** and noticed three things. First, the CPU utilization (%usr + %sys) was 90-100% when 150 users logged in. Second, I noticed the deficit memory (the "de" column) was constantly between 4000-6000. Third, the scan rate (the "sr" column) was constantly between 1000-15000.

## What Those Numbers Mean

The CPU utilization numbers indicated that the 150-user workload consumed 90-100% of the available processor power. A server should rarely consume 60% except for periodic spikes. The deficit memory column constantly reading non-zero meant that the server was repeatedly being asked to allocate large chunks of memory. Solaris reacts to these requests by grabbing additional free memory and setting it aside in anticipation of more bursts of memory demands. The deficit column should be zero with a few bursts that last one to five seconds. The scan rate column constantly being high meant that the pageout daemon was looking at many pages to find enough to free. A good rule of thumb is that the scan rate should be zero with a few bursts where it reaches 100 times the number of CPUs.

## What Parameters I Changed and Why

I'm like any other UNIX sys admin -- as soon as I saw ugly numbers in the memory columns in **vmstat**, I assumed that more memory would improve things. In my defense (also known as a rationalization) I had to do something quickly to help the client keep up with the ever-increasing workload. I increased the memory from 3 GB to 6 GB and saw almost no change in the behavior of the server. Undaunted, I slapped more memory into the E3500 bringing the total to 10 GB. When this also had no impact on the poor performance, I decided I better investigate kernel-tunable parameters.

First, I activated priority paging. You need patch 105181-13 to use this feature with Solaris 2.6. Solaris 8 has a new file system caching architecture called cyclical page cache that eliminates the need to tinker with priority paging. This parameter alters Solaris's paging algorithm. Solaris uses memory to hold the kernel, application programs, and file buffers (which is why you always ignore the **freemem** column in **vmstat** prior to Solaris 8). Priority paging tells Solaris to page out file buffer pages before paging out application code pages.

Second, I increased **slowscan** to 500. When the pageout daemon wakes up every 1/4 of a second, it tries to free up **slowscan** number of pages. The default of 100 pages was a reasonable amount of work to be performed by older Sun computers; however, today's systems can easily do more work. By increasing **slowscan** from 100 to 500, I was asking the pageout daemon to do five times more work each time it woke up.

Third, I increased **maxpgio** to 25468. When the pageout daemon looks for pages to free, the first requirement is that the page is not currently referenced. Next, the daemon must determine whether the page has been modified. Dirty pages must be written to disk before being removed from memory. The **maxpgio** parameter controls the amount of I/O done by the pageout daemon. After the daemon has flushed **maxpgio** dirty pages, then only unmodified pages can be freed. The default setting of 64 was a reasonable amount of work to be performed by older 3600-rpm disks; however, today's faster disks can easily do more work.

## How I Changed the Parameters

Some parameters can be changed dynamically using the **adb** utility, but the **maxpgio** parameter can only be changed in **/etc/system**. Therefore, I usually change all parameters at once by adding the lines below to **/etc/system** and rebooting:

```
set priority_paging=1
set maxpgio=25468
set slowscan=500
```

With 150 users, the CPU utilization dropped from 90-100% to 5% and the scan rate fell from 10000 to 0. Periodic spikes still showed up in the deficit memory column but only for a couple of seconds. When the workload was increased to 250 users, the CPU utilization on the E3500 only increased to 15-20% and the scan rate stayed near 0. The client's business was growing so fast that it was imperative for the number of interactive users to increase quickly. The client opted to purchase an E10000 because I was initially unable to correct the problem on the E3500. Two days before the E10000 was delivered, I corrected the memory/CPU performance problem, which obviously made my client both pleased and upset. Now I don't wait for performance problems to surface. Every time I configure a Sun server, I add the lines above to **/etc/system**.

## I/O Bottleneck

In the third and final scenario, I had an Ultra10 with 512 MB that was functioning as an NFS server for a small department. As more users were added to the server, performance deteriorated. Everyone was demanding the purchase of a more powerful server, but since money was scarce I decided to experiment.

## How I Investigated the Problem

I started by looking at **iostat -x**. Nothing awful jumped off the page at me; that is, no ugly values for %w or %b. Next, I looked at the directory name lookup cache (DNLC) using **vmstat -s**. I saw that the cache-hit ratio was 60% since the reboot that morning. I looked at the daily SAR reports (**sar -a**) to see whether this pattern held for previous days and weeks. The **namei/s** column shows the total number of calls, while the **iget/s** column shows the number of misses. The DNLC hit ratio =  $1 - (\text{misses}/\text{total})$ . As I suspected, the daily DNLC hit ratio was consistently under 80%, with periods during the day around 50%.

I also checked the inode cache by running **netstat -k**. The **maxsize** value is the size of the inode cache, while the **maxsize\_reached** value is the high-water mark. The cache had been exceeded.

## What Those Numbers Mean

The DNLC is used whenever a file is opened. If a directory entry is not found in the cache, then a slower method is needed to obtain file information, perhaps even a disk I/O. As shown below, a 99% hit ratio can equate to a 29% performance degradation:

A hit costs 20 units

A miss costs 600 units (30x)

At 100% hit ratio,  $\text{cost} = (100 * 20) = 2000$  units

At 99% hit ratio,  $\text{cost} = (99 * 20) + (1 * 600) = 2580$ , a 29% degradation

## What Parameters I Changed and Why

The **ncsize** parameter defines the size of the DNLC. Its default setting is based on memory and is usually sufficient; however, I set this to 5000 on NFS servers. I've seen this set as high as 16000 during NFS benchmark testing. Another trick to improve the DNLC hit ratio is to reduce the depth of the tree for directories served by NFS. A more shallow structure means fewer entries in the DNLC, which has a similar effect to increasing the size of the DNLC. The **ufs\_ninode** parameter defines the number of inactive inode entries, which impacts the size of the inode cache. Since stateless NFS clients don't keep inodes active, I set **ufs\_ninode** to two times the **ncsize** value on NFS server.

## How I Changed the Parameters

The **ufs\_ninode** parameter can be changed using the **adb** utility, but the **ncsize** parameter can only be changed in **/etc/system**. Therefore, I usually change both parameters at once by adding the lines below to **/etc/system** and rebooting:

```
set ncsiz=5000
set ufs_ninode=10000
```

These small changes improved NFS performance on the little Ultra10 without spending any money.

## Conclusion

The performance problems described in this article are common, so much so that I improve my chances of success by making the parameter changes every time I set up a server without waiting for a problem to surface. If you suspect network problems, refer to Jens Voeckler's Web site:

<http://www.sean.de/Solaris>

If you have memory, CPU, or disk problems, a good resource is Adrian Cockroft's book, *Sun Performance and Tuning SPARC and Solaris*, Second edition (Sun Microsystems Press, ISBN 0-13-149642-5). It's not easy reading if you're just grazing, but if you're investigating a specific problem, this book is the place to go. In October 2000, Sun published a manual describing all the kernel-tunable parameters, which is available at:

<http://docs.sun.com>

Search for 806-4015 and 806-6779.

*Ken Gottry is a Sr. Infrastructure Architect with NerveWire, Inc. He has 30 years experience with systems ranging from mainframes, to minicomputers, to servers, to desktops. For the past 10 years, his focus has been on designing, implementing, and tuning distributed, multi-tier, and Web-based systems. As a performance engineer consulting to numerous G2K firms, he has assessed and tuned Web servers, app servers, and JVMs running on Solaris.*

Copyright © 2001 Sys Admin, [Sys Admin's Privacy Policy](#). Comments about the Web site:  
[webmaster@sysadminmag.com](mailto:webmaster@sysadminmag.com)