

Unix System Tuning Made Difficult

By: Micah Altman, (Copyright 1994-8)

Last Revision: Mar 8, 1998

Spotting Bottlenecks

On most systems resources are unequally used. The system (as a whole) is likely to be waiting on one resource or another in order to proceed — this resource is a bottleneck. Effective tuning looks for system bottlenecks and removes them.

Monitoring Tools

Unix generally provides two types of monitoring tools, tools for monitoring processes and tools for monitoring system resources:

- `top` and `ps` - will show you information about the CPU and memory use of individual processes
- `sar` (Sys V), `iostat/vmstat` (BSD), and `osview`, `gr_osview` and `gmemusage` (IRIX) - show system wide resource use.
- `time/timex` - shows the total CPU time in user and system time, plus total wall-clock time for program run (`timex -s` will also show total sar-like system stats for duration of program run)
- `ssusage` - shows page faults and i/o activity as well as execution time for the individual program run (not system wide activity like `timex -s`)
- `par/` (`trace/truss/ptrace` on other systems) - shows system call activity and process scheduling
- `prof/gprof` - program profiling, shows details of the program run by line or subroutine

Symptoms of Resource Bottlenecks

- CPU Bottlenecks — CPU User+Sys activity near 100%, no CPU idle, no CPU wait on I/O, (`sar -u`) low context switch (`pswch` in `sar -p`) activity . Check the run-queue (`sar -q`) to see how many processes are waiting to run, a high run-queue can indicate a CPU bottleneck, but you can have a CPU without a high run-queue (if only one large process is running.) If CPU Sys activity is high, suspect inefficient use of system calls, or borderline memory or i/o bottlenecks.
- Memory Bottlenecks — in severe bottlenecks you will see the CPU waiting on swap and heavy swap activity. ("`wio`" & "`wswp`" in `sar -u`). In less severe cases you'll see no wait on swap, but free memory will drop (`sar -r`) and you will see increased validity faults (`sar -p`), context switches and system calls/system CPU time usage. Some architectures offer additional tuning opportunities:

- Cross-bow architecture — for systems that use this architecture, run `xbowstats` to look for errors on the bus. Errors on the bus will slow down access to memory. Systems with frequent errors should be investigated for hardware faults.

- Non-Uniform Memory Architecture (NUMA) Bottlenecks — Beyond checking for errors along the interconnections between modules with `linkstat -a`, and fixing hardware if errors exist, NUMA tuning is effective only at the applications level. You can use `sn` to look at activity related to NUMA, such as page duplication and migration.

In almost all cases, however, tuning will only be necessary for multi-threaded shared-memory applications with non-standard memory access patterns. Even in such cases, profiling of the executable with `dprof` and `dplace` is highly recommended to obtain information in order to do any tuning.

If you cannot profile the code, and wish to experiment with tuning for a multi-threaded job, try activating page migration, and/or changing initial placement of data to the round-robin method using `sn` (view the results with `nstats`) Be aware that changes are likely to decrease performance for many multi-threaded applications.

If you know through profiling, or see through observation that there is a lot of sharing of pages between far nodes, and you page migration and duplication cannot eliminate this, consider enhancing the processor topology with the “express-link” cross-connector.

- Disk Bottlenecks — in severe bottlenecks you will see high wait on Filesystem (wfs) or physical (wph) i/o (in `sar -u`). You will also see the disk busy, and processes queued to use the disk (`sar -d`). In less severe case you may see no wait, but increased physical or block read and write activity (`sar -b`), context switches and system calls/system CPU time usage. (*Disk tuning is discussed in more detail in "Tuning and Troubleshooting Filesystems"*)

Kernel Variables

In rare cases, you may be able to improve performance by changing kernel kernel variables. You can change these through `sysctl`. Be cautious when imposing limits, as they may cause programs to fail when they encounter them, and are generally crude. A more precise way of allocating resources is to use the IRIX/Share or NQS products:

- CPU related variables.
 - The cpu limit variables (`rlimit_cpu_max/cur`) will limit individual processes to a maximum number of cpu seconds. This is a very crude way of limiting "cpu-hogging" and is likely to backfire by causing legitimate programs to fail.
 - Increasing the system time-slice will cause the scheduler to run less frequently, advantaging cpu-intensive programs over interactive programs. You can change the `slice_size` system-wide, or use `npri -t` to change the slice-size for individual

processes.

— You can limit the number of processes allowed for non-root users, with the `maxup` variable (remember that the table size itself is controlled by `nproc`).

- *Memory* related variables. In general, huge kernel tables can waste memory, but it is unlikely that the tables on your system are too large if you have not altered the defaults. On exceptional systems, you may want to change the parameters controlling the paging daemon, `vhand`, page sizes or `tlb`:

— if you have an unusually large amount of RAM installed (256MB+), change the low and high water marks that control when paging starts, `GPGSLO` and `GPGSHI`.

— if you are swapping over NFS, use local swap instead. If you cannot use local swap, decrease `MAXSC` to allow paging to take place in smaller increments.

— if you are experiencing a large amount of TLB activity in a large program, try increasing `TLBDROP`

— if you are running programs on your system that use very large contiguous data structures, you may want to allocate larger (16K-16MB depending on application) pages. You can do this with the dynamic page size kernel parameters, but for large page sizes it is more effective to use the parameters that pre-allocate large pages on boot-up.

— to prevent individual programs from “hogging” memory, you may adjust the system memory limits on virtual memory (total, data, or stack), resident memory (pages in RAM), or lockable memory pages (in RAM and never paged to disk) through `rlimit_vmem_cur/max`, `rlimit_data_cur/max`, `rlimit_stack_cur/max`, `rlimit_rss_cur/max`, and `maxlkmem`. Be especially cautious with the virtual memory limits, as they can cause applications to fail in unexpected and mysterious ways.

- *Filesystem* related variables.

— to decrease the overhead associated with file-system caching, increase `BDFLUSHR` and `AUTOUP`. Be aware that this will increase the probability of data-loss, if there is a power failure or system crash. (Also look for parameters controlling `vfs_sync`, `xfsd`, and `pd_flush` which perform analogous functions)

— if the system has a very large number of files open, increasing the filesystem path cache and the buffer header list (`NBUF`) can help with filename lookups and caching

— Troubleshooting. On systems that open many files, you may need to increase the in-memory inode table (`NINODE`), and the open-file descriptor table.

Increasing Throughput

The typical goal of system tuning is to increase throughput — the weighted total of work done on the system.¹ Tuning the system may require taking resources away from some piggish applications. Also, remember that the bottleneck for system activity may not be identical to the bottleneck for a particular application, especially if the system can do other things while that application is waiting.

There are some general principles for remedying bottlenecks:

- *Fix Errors First* — Fast answers are no good if they are wrong, so first make sure that software and hardware is operating correctly. Furthermore, correctable hardware faults can severely degrade performance:
 - floating point exceptions, a software error but it can result in a hardware interrupt, these usually appear as interrupts under `sar -u`
 - network packet corruption, shows as errors on `netstat -i`
 - errors along the internal bus, use `linkstat` or `xbowstat` to see these
 - memory parity errors, show up in `SYSLOG`
- *Waste not want not* — If you run into a bottleneck, look for things on the system that take up significant resources and that you don't need: unused network services, complicated screen-savers, auditing & accounting.
- *Put off til tomorrow what you don't need to do today* — most systems aren't used heavily at all times. Try to put off resource-intensive jobs until quieter times with `at` or `batch`. more elaborate batch job management is found in the commercial product `NQS` and in the `IRIX (6.5+)` product `miser`.
- *Economies of scale* — all resources are allocated in chunks, find what the chunk-size is and try to get applications to use even multiple of these chunks to decrease wastage and excess overhead.
- *Set your priorities*

¹ Other tuning goals might include:

- Guaranteeing individual users a minimum performance level. Consider using the `IRIX/Share` product, or a queueing system such as `NQS` or `MISER`.
- Minimizing latency for selected individual programs. This is explained in detail in the `IRIX REACT` (real-time) documentation.
- Maximizing the performance of an individual program. You can usually do this by boosting the priority, locking the memory of that program, and giving the program a `GRIO` (guaranteed rate I/O) stream.
- Maximizing peak performance. Often implemented through increasing buffers and caches.

- Process priorities (set with `nice`, `renice` (BSD) or `npri` (IRIX)) allow you to tell the system that some processes are more important than others. Important processes will tend to get a bigger share of all resources.
- On systems with multiple processors, you might want to use `runon`, `pset`, *Irix/Share* (an add-on product), or `miser` to allocate blocks of CPU's to individual processes.
- A queueing system can distribute jobs across multiple cpus or multiple systems. A generic version of NQS is available from www.gnu.org, and can be used to distribute jobs across the network.
- *Impose limits* — you can impose limits on the amount of memory, processes, cpu-time and disk space that individual users can use through changing kernel variables. This can help performance, but it can also cause programs to fail when they hit these limits. Don't be draconian.
- *Sales tuning* — Sometimes when you run out of resources it is reasonable to buy more. Know where your bottleneck is so that you can get the most bang for your buck.

Closing Thoughts

- *BottlenecksShift* — resource bottlenecks shift over time, with different use of the system, and as you tune. When you fix a big bottleneck, you may find a smaller one somewhere else.
- *Check for Errors Too* — floating point errors (which will show up as interrupts in `sar -u`), errors along the network backbone (`netstat -ia`), or along internal busses (`xbowstat` on O², `linkstat` for Origins) can reduce performance directly and indirectly. Fix errors first.
- *Avoid Overtuning* — a system that is overtuned may perform worse than an untuned system when patterns of use on the system changes, getting those last few cycles out of the processor, or the last few kilobytes of memory free is usually not worthwhile.
- *Remember the Network* — many applications operate over the network, if your network is slow, the network may be the bottleneck. (Network tuning is too large a subject to be dealt with here, but you might start off by looking at the the statistics from `netstat` and `nfsstat` or the network activity bars in `gr_osview` (IRIX))
- *Remember the Application* — all system tuning can do is to make sure that the system makes appropriate resources available to program, it cannot force them to use these resources efficiently. By far the greatest increase of performance comes from using applications that are efficiently designed to solve problems and tuned for the architecture within which they are running.