USE Method: Rosetta Stone of Performance Checklists

The following USE Method example checklists are automatically generated from the individual pages for: Linux, Solaris, Mac OS X, and FreeBSD. These analyze the performance of the physical host. You can customize this table using the checkboxes on the right.

☑ Linux
☐ Solaris
☐ FreeBSD
☐ Mac OS X
Redraw

There are some additional USE Method example checklists not included in this table: the SmartOS checklist, which is for use within an OS virtualized guest, and the Unix 7th Edition checklist for historical interest.

For general purpose operating system differences, see the Rosetta Stone for Unix, which was the inspiration for this page.

Hardware Resources

Resource	Metric	Linux
CPU	errors	`perf` (LPE) if processor specific error events (CPC) are available; eg, AMD64's "04Ah Single-bit ECC Errors Recorded by Scrubber" [4]
CPU	saturation	system-wide: `vmstat 1`, "r" > CPU count [2]; `sar -q`, "runq-sz" > CPU count; `dstat -p`, "run" > CPU count; per-process: /proc/PID/schedstat 2nd field (sched_info.run_delay); `perf sched latency` (shows "Average" and "Maximum" delay per-schedule); dynamic tracing, eg, SystemTap schedtimes.stp "queued(us)" [3]
CPU	utilization	system-wide: `vmstat 1`, "us" + "sy" + "st"; `sar -u`, sum fields except "%idle" and "%iowait"; `dstat -c`, sum fields except "idl" and "wai"; per-cpu: `mpstat -P ALL 1`, sum fields except "%idle" and "%iowait"; `sar -P ALL`, same as `mpstat`; per-process: `top`, "%CPU"; `htop`, "CPU%"; `ps -o pcpu`; `pidstat 1`, "%CPU"; per-kernel-thread: `top`/`htop` ("K" to toggle), where VIRT == 0 (heuristic). [1]
CPU interconnect	errors	LPE (CPC) for whatever is available
CPU interconnect	saturation	LPE (CPC) for stall cycles
CPU interconnect	utilization	LPE (CPC) for CPU interconnect ports, tput / max
GPU	errors	-
GPU	saturation	-
GPU	utilization	-
I/O interconnect	errors	LPE (CPC) for whatever is available
I/O interconnect	saturation	LPE (CPC) for stall cycles
I/O interconnect	utilization	LPE (CPC) for tput / max if available; inference via known tput from iostat/ip/...
Memory capacity	errors	`dmesg` for physical failures; dynamic tracing, eg, SystemTap uprobes for failed malloc()s
Memory capacity	saturation	system-wide: `vmstat 1`, "si"/"so" (swapping); `sar -B`, "pgscank" + "pgscand" (scanning); `sar -W`; per-process: 10th field (min_flt) from /proc/PID/stat for minor-fault rate, or dynamic tracing [5]; OOM killer: `dmesg | grep killed`
Memory capacity	utilization	system-wide: `free -m`, "Mem:" (main memory), "Swap:" (virtual memory); `vmstat 1`, "free" (main memory), "swap" (virtual memory); `sar -r`, "%memused"; `dstat -m`, "free"; `slabtop -s c` for kmem slab usage; per-process: `top`/`htop`, "RES" (resident main memory), "VIRT" (virtual memory), "Mem" for system-wide summary

Memory interconnect	errors	LPE (CPC) for whatever is available
Memory interconnect	saturation	LPE (CPC) for stall cycles
Memory interconnect	utilization	LPE (CPC) for memory busses, tput / max; or CPI greater than, say, 5; CPC may also have local vs remote counters
Network Interfaces	errors	`ifconfig`, "errors", "dropped"; `netstat -i`, "RX-ERR"/"TX-ERR"; `ip -s link`, "errors"; `sar -n EDEV`, "rxerr/s" "txerr/s"; /proc/net/dev, "errs", "drop"; extra counters may be under /sys/class/net/...; dynamic tracing of driver function returns 76]
Network Interfaces	saturation	`ifconfig`, "overruns", "dropped"; `netstat -s`, "segments retransmited"; `sar -n EDEV`, *drop and *fifo metrics; /proc/net/dev, RX/TX "drop"; nicstat "Sat" [6]; dynamic tracing for other TCP/IP stack queueing [7]
Network Interfaces	utilization	`sar -n DEV 1`, "rxKB/s"/max "txKB/s"/max; `ip -s link`, RX/TX tput / max bandwidth; /proc/net/dev, "bytes" RX/TX tput/max; nicstat "%Util" [6]
Network controller	errors	see network interface errors, ...
Network controller	saturation	see network interface saturation, ...
Network controller	utilization	infer from `ip -s link` (or /proc/net/dev) and known controller max tput for its interfaces
Storage capacity	errors	`strace` for ENOSPC; dynamic tracing for ENOSPC; /var/log/messages errs, depending on FS
Storage capacity	saturation	not sure this one makes sense - once it's full, ENOSPC
Storage capacity	utilization	swap: `swapon -s`; `free`; /proc/meminfo "SwapFree"/"SwapTotal"; file systems: "df -h"
Storage controller	errors	see storage device errors, ...
Storage controller	saturation	see storage device saturation, ...
Storage controller	utilization	`iostat -xz 1`, sum devices and compare to known IOPS/tput limits per-card
Storage device I/O	errors	/sys/devices/.../ioerr_cnt; `smartctl`; dynamic/static tracing of I/O subsystem response codes [8]
Storage device I/O	saturation	`iostat -xnz 1`, "avgqu-sz" > 1, or high "await"; `sar -d` same; LPE block probes for queue length/latency; dynamic/static tracing of I/O subsystem (incl. LPE block probes)
Storage device I/O	utilization	system-wide: `iostat -xz 1`, "%util"; `sar -d`, "%util"; per-process: iotop; `pidstat -d`; /proc/PID/sched "se.statistics.iowait_sum"
Resource	**Metric**	**Linux**

Linux Notes

- [1] There can be some oddities with the %CPU from top/htop in virtualized environments; I'll update with details later when I can.
- CPU utilization: a single hot CPU can be caused by a single hot thread, or mapped hardware interrupt. Relief of the bottleneck usually involves tuning to use more CPUs in parallel.
- `uptime` "load average" (or /proc/loadavg) wasn't included for CPU metrics since Linux load averages include tasks in the uninterruptable state (usually I/O).
- [2] The man page for vmstat describes "r" as "The number of processes waiting for run time", which is either incorrect or misleading (on recent Linux distributions it's reporting those threads that are waiting, *and* threads that are running on-CPU; it's just the wait threads in other OSes).

- [3] There may be a way to measure per-process scheduling latency with perf's sched:sched_process_wait event, otherwise `perf probe` to dynamically trace the scheduler functions, although, the overhead under high load to gather and post-process many (100s of) thousands of events per second may make this prohibitive. SystemTap can aggregate per-thread latency in-kernel to reduce overhead, although, last I tried schedtimes.stp (on FC16) it produced thousands of "unknown transition:" warnings.
- LPE == Linux Performance Events, aka perf_events. This is a powerful observability toolkit that reads CPC and can also use static and dynamic tracing. Its interface is the `perf` command.
- CPC == CPU Performance Counters (aka "Performance Instrumentation Counters" (PICs) or "Performance Monitoring Events" (PMUs) or "Hardware Events"), read via programmable registers on each CPU by `perf` (which it was originally designed to do). These have traditionally been hard to work with due to differences between CPUs. LPE `perf` makes life easier by providing aliases for commonly used counters. Be aware that there are usually many more made available by the processor, accessible by providing their hex values to `perf stat -e`. Expect to spend some quality time (days) with the processor vendor manuals when trying to use these. (My short video about CPC may be useful, despite not being on Linux).
- [4] There aren't many error-related events in the recent Intel and AMD processor manuals; be aware that the public manuals may not show a complete list of events.
- [5] The goal is a measure of memory capacity saturation - the degree to which a process is driving the system beyond its ability (and causing paging/swapping). High fault latency works well, but there isn't a standard LPE probe or existing SystemTap example of this (roll your own using dynamic tracing). Another metric that may serve a similar goal is minor-fault rate by process, which could be watched from /proc/PID/stat. This should be available in `htop` as MINFLT.
- [6] Tim Cook ported nicstat to Linux; it can be found on sourceforge or his blog.
- [7] Dropped packets are included as both saturation and error indicators, since they can occur due to both types of events.
- [8] This includes tracing functions from different layers of the I/O subsystem: block device, SCSI, SATA, IDE, ... Some static probes are available (LPE "scsi" and "block" tracepoint events), else use dynamic tracing.
- CPI == Cycles Per Instruction (others use IPC == Instructions Per Cycle).
- I/O interconnect: this includes the CPU to I/O controller busses, the I/O controller(s), and device busses (eg, PCIe).
- Dynamic Tracing: Allows custom metrics to be developed, live in production. Options on Linux include: LPE's "perf probe", which has some basic functionality (function entry and variable tracing), although in a trace-n-dump style that can cost performance; SystemTap (in my experience, almost unusable on CentOS/Ubuntu, but much more stable on Fedora); DTrace-for-Linux, either the Paul Fox port (which I've tried) or the OEL port (which Adam has tried), both projects very much in beta.

Solaris Notes

- CPU utilization: a single hot CPU can be caused by a single hot thread, or mapped hardware interrupt. Relief of the bottleneck usually involves tuning to use more CPUs in parallel.
- lockstat and plockstat are DTrace-based since Solaris 10 FCS.
- vmstat "r": this is coarse as it is only updated once per second.
- CPC == CPU Performance Counters (aka "Performance Instrumentation Counters" (PICs), or "Performance Monitoring Events"), read via programmable registers on each CPU, by cpustat(1M) or the DTrace "cpc" provider. These have traditionally been hard to work with due to differences between CPUs, but are getting much easier with the PAPI standard. Still, expect to spend some quality time (days) with the processor vendor manuals (what "cpustat -h" tells you to read), and to post-process cpustat with awk or perl. See my short talk (video) about CPC (2010). (Many years ago, I made a toolkit including CPC scripts - CacheKit - that was too much work to maintain.)
- Memory capacity utilization: interpreting vmstat's "free" has been tricky across different Solaris versions (we documented it in the Perf & Tools book), due to different ways it was calculated, and tunables that affect when the system will kick-off the page scanner. It'll also typically shrink as the kernel uses unused memory for caching (ZFS ARC).
- Be aware that kstat can report bad data (so can any tool); there isn't really a test suite for kstat data, and engineers can add new code paths and forget to add the counters.

- DTT == DTraceToolkit scripts, DTB == DTrace book scripts.
- CPI == Cycles Per Instruction (others use IPC == Instructions Per Cycle).
- I/O interconnect: this includes the CPU to I/O controller busses, the I/O controller(s), and device busses (eg, PCIe).

FreeBSD Notes

- [1] eg, using per-CPU run queue length as the saturation metric: `dtrace -n 'profile-99 { @[cpu] = lquantize(`tdq_cpu[cpu].tdq_load, 0, 128, 1); } tick-1s { printa(@); trunc(@); }'` where > 1 is saturation. If you're using the older BSD scheduler, profile `runq_length[]`. There are also the sched:::load-change and other sched probes.
- [2] For this metric, lets use time spent in TDS_RUNQ as a per-thread saturation (latency) metric. Here is an (unstable) fbt-based one-liner: `dtrace -n 'fbt::tdq_runq_add:entry { ts[arg1] = timestamp; } fbt::choosethread:return /ts[arg1]/ { @[stringof(args[1]->td_name), "runq (ns)"] = quantize(timestamp - ts[arg1]); ts[arg1] = 0; }'`. This would be better (stability) if it can be rewritten to use the sched probes. It would also be great if there were simply high resolution thread state times in `struct rusage` or `rusage_ext`, eg, cumulative times for each state in `td_state` and more, which would make reading this metric easier and have lower overhead. See the Thread State Analysis Method from my Velocity talk for suggested states.
- [3] eg, for swapping: `dtrace -n 'fbt::cpu_thread_swapin:entry, fbt::cpu_thread_swapout:entry { @[probefunc, stringof(args[0]->td_name)] = count(); }'` (NOTE, I would trace vm_thread_swapin() and vm_thread_swapout(), but their probes don't exist). Tracing paging is tricker until the vminfo provider is added; you could try tracing from swap_pager_putpages() and swap_pager_getpages(), but I didn't see an easy way to walk back to a thread struct; another approach may be via vm_fault_hold(). Good luck. See thread states [2], which could make this much easier.
- [4] eg, sampling GEOM queue length at 99 Hertz: `dtrace -n 'profile-99 { @["geom qlen"] = lquantize(`g_bio_run_down.bio_queue_length, 0, 256, 1); }'`
- [5] This approach is different from storage device (disk) utilization. For controllers, percent busy has much less meaning, so we're calculating utilization based on throughput (bytes/sec) and IOPS instead. Controllers typically have limits for these based on their busses and processing capacity. If you don't know them, you can determine them experimentally.
- PMC == Performance Monitoring Counters, aka CPU Performance Counters (CPC), Performance Instrumentation Counters (PICs), and more. These are processor hardware counters that are read via programmable registers on each CPU. The availability of these counters is dependent on the processor type. See pmc(3) and pmcstat(8).
- pmcstat(8): the FreeBSD tool for instrumenting PMCs. You might need to run a `kldload hwpmc` first before use. To figure out which PMCs you need to use and how, it usually takes some serious time (days) with the processor vendor manuals; eg, the Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B, Appendix A-E: Performance-Monitoring Events.
- DTT == DTraceToolkit scripts. These are in the FreeBSD source under cddl/contrib/dtracetoolkit, and dtruss is under /usr/sbin. As features are added to DTrace (see the freebsd-dtrace mailing list), more scripts can be ported.
- CPI == Cycles Per Instruction (others use IPC == Instructions Per Cycle).
- I/O interconnect: this includes the CPU to I/O controller busses, the I/O controller(s), and device busses (eg, PCIe).

Mac OS X Notes

- [1] eg: dtrace -x aggsortkey -n 'profile-100 /!(curthread->state & 0x80)/ { @ = lquantize(cpu, 0, 1000, 1); } tick-1s { printa(@); clear(@); }'. Josh Clulow also wrote a simple C program to dig out per-CPU utilization: cpu_usage.c.
- [2] Until there are sched:::enqueue/dequeue probes, I suspect this could be done using fbt tracing of thread_*(). I haven't tried yet. It might be worth seeing what Instruments uses for its "On run queue" thread state trace, and DTracing that.
- [3] eg: dtrace -n 'vminfo:::anonpgin { printf("%Y %s", walltimestamp, execname); }'.

- [4] the kernel source under bsd/vm/vm_unix.c describes this as "Memory pressure indicator", although I've yet to see this as non-zero.
- [5] the netstat(1) man page reads: "BUGS: The notion of errors is ill-defined."
- [6] it would be great if Mac OS X iostat added a -x option to include utilization, saturation, and error columns, like Solaris "iostat -xnze 1".
- atMonitor is a 3rd party tool that provides various statistics; I'm running version 2.7b, although it crashes if you leave the "Top Window" open for more than 2 seconds.
- Activity Monitor is a default Apple performance monitoring tool with a graphical interface.
- Instruments is an Apple performance analysis product with a graphical interface. It is comprehensive, consuming performance data from multiple frameworks, including DTrace. Instruments also includes functionality that was provided by separate previous performance analysis products, like CHUD and Shark, making it a one stop shop. It'd be wonderful if it included latency heat maps as well :-).
- Temperature Monitor: 3rd party software that can read various temperature probes.
- PMC == Performance Monitor Counters, aka CPU Performance Counters (CPC), Performance Instrumentation Counters (PICs), and more. These are processor hardware counters that are read via programmable registers on each CPU.
- DTT == DTraceToolkit scripts, many of which were ported by the Apple engineers and shipped by default with Mac OS X. ie, you should be able to run these immediately, eg, `sudo runocc.d`.
- [7] I haven't found a shipped tool to provide GPU statistics easily. I'd like a `gpustat` that behaved like `mpstat`, with at least the columns: utilization, saturation, errors. Until there is such a tool, you could trace GPU activity (at least the scheduling of activity) using DTrace on the graphics drivers. It won't be easy. I imagine Instruments will at some point add a GPU instrument set (other than the OpenGL instruments), otherwise, 3rd party tools can be used, like atMonitor.
- CPI == Cycles Per Instruction (others use IPC == Instructions Per Cycle).
- I/O interconnect: this includes the CPU to I/O controller busses, the I/O controller(s), and device busses (eg, PCIe).
- Using PMCs is typically a lot of work. This involves researching the processor manuals to see what counters are available and what they mean, and then collecting and interpreting them. I've used them on other OSes, but haven't used them all under Instruments → Counters, so I don't know if there's a hitch with anything there. Good luck.

Software Resources

Resource	Metric	Linux
File descriptors	errors	`strace` errno == EMFILE on syscalls returning fds (eg, open(), accept(), ...).
File descriptors	saturation	does this make sense? I don't think there is any queueing or blocking, other than on memory allocation.
File descriptors	utilization	system-wide: `sar -v`, "file-nr" vs /proc/sys/fs/file-max; `dstat --fs`, "files"; or just /proc/sys/fs/file-nr; per-process: `ls /proc/PID/fd | wc -l` vs `ulimit -n`
Kernel mutex	errors	dynamic tracing (eg, recusive mutex enter); other errors can cause kernel lockup/panic, debug with kdump/`crash`
Kernel mutex	saturation	With CONFIG_LOCK_STATS=y, /proc/lock_stat "waittime-total" / "contentions" (also see "waittime-min", "waittime-max"); dynamic tracing of lock functions or instructions (maybe); spinning shows up with profiling (`perf record -a -g -F 997 ...`, `oprofile`, dynamic tracing)
Kernel mutex	utilization	With CONFIG_LOCK_STATS=y, /proc/lock_stat "holdtime-totat" / "acquisitions" (also see "holdtime-min", "holdtime-max") [8]; dynamic tracing of lock functions or instructions (maybe)
Process capacity	errors	-
Process capacity	saturation	-

Resource	Metric	Linux
Process capacity	utilization	-
Task capacity	errors	"can't fork()" errors; user-level threads: pthread_create() failures with EAGAIN, EINVAL, ...; kernel: dynamic tracing of kernel_thread() ENOMEM
Task capacity	saturation	threads blocking on memory allocation; at this point the page scanner should be running (sar -B "pgscan*"), else examine using dynamic tracing
Task capacity	utilization	`top/htop`, "Tasks" (current); `sysctl kernel.threads-max`, /proc/sys/kernel/threads-max (max)
Thread capacity	errors	-
Thread capacity	saturation	-
Thread capacity	utilization	-
User mutex	errors	`valgrind --tool=drd` various errors; dynamic tracing of pthread_mutex_lock() for EAGAIN, EINVAL, EPERM, EDEADLK, ENOMEM, EOWNERDEAD, ...
User mutex	saturation	`valgrind --tool=drd` to infer contention from held time; dynamic tracing of synchronization functions for wait time; profiling (oprofile, PEL, ...) user stacks for spins
User mutex	utilization	`valgrind --tool=drd --exclusive-threshold=...` (held time); dynamic tracing of lock to unlock function time
Resource	**Metric**	**Linux**

Linux Notes

- [8] Kernel lock analysis used to be via lockmeter, which had an interface called "lockstat".

Solaris Notes

- lockstat/plockstat often drop events due to load; I often roll my own to avoid this using the DTrace lockstat/plockstat provider (there are examples of this in the DTrace book).
- File descriptor utilization: while other OSes have a system-wide limit, Solaris doesn't (at least at the moment, this could change; see my writeup about it).

FreeBSD Notes

- lockstat: you may need to run `kldload ksyms` before lockstat will work (otherwise: "lockstat: can't load kernel symbols: No such file or directory").
- [6] eg, showing adaptive lock block time totals (in nanoseconds) by calling function name: dtrace -n 'lockstat:::adaptive-block { @[caller] = sum(arg1); } END { printa("%40a%@16d ns\n", @); }'

Mac OS X Notes

- [8] eg, showing adaptive lock block time totals (in nanoseconds) by calling function name: dtrace -n 'lockstat:::adaptive-block { @[caller] = sum(arg1); } END { printa("%40a%@16d ns\n", @); }'

More Info

See the individual checklist pages listed at the top. This is a summary of their content.

Last Generated: 09-Feb-2014
See individual source pages for last update times