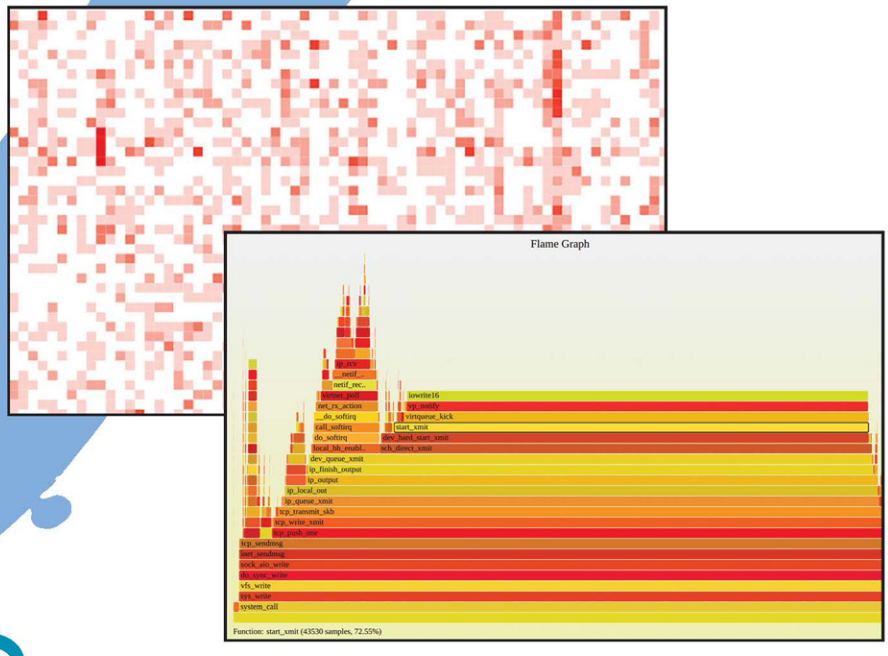
Systems Performance

Enterprise and the Cloud

Second Edition

Brendan Gregg



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Systems Performance

Second Edition

GPUs

Graphics processing units (GPUs) were created to support graphical displays, and are now finding use in other workloads including artificial intelligence, machine learning, analytics, image processing, and cryptocurrency mining. For servers and cloud instances, a GPU is a processor-like resource that can execute a portion of a workload, called the *compute kernel*, that is suited to highly parallel data processing such as matrix transformations. General-purpose GPUs from Nvidia using its Compute Unified Device Architecture (CUDA) have seen widespread adoption. CUDA provides APIs and software libraries for using Nvidia GPUs.

While a processor (CPU) may contain a dozen cores, a GPU may contain hundreds or thousands of smaller cores called *streaming processors* (SPs),[5] which each can execute a thread. Since GPU workloads are highly parallel, threads that can execute in parallel are grouped into *thread blocks,* where they may cooperate among themselves. These thread blocks may be executed by groups of SPs called *streaming multiprocessors* (SMs) that also provide other resources including a memory cache. Table 6.6 further compares processors (CPUs) with GPUs [Ather 19].

Table 6.6 **CPUs versus GPUs**

Attribute	CPU	GPU
Package	A processor package plugs into a socket on the system board, connected directly to the system bus or CPU interconnect.	A GPU is typically provided as an expansion card and connected via an expansion bus (e.g., PCIe). They may also be embedded on a system board or in a processor package (on-chip).
Package scalability	Multi-socket configurations, connected via a CPU interconnect (e.g., Intel UPI).	Multi-GPU configurations are possible, connected via a GPU-to-GPU interconnect (e.g., NVIDIA's NVLink).
Cores	A typical processor of today contains 2 to 64 cores.	A GPU may have a similar number of streaming multiprocessors (SMs).
Threads	A typical core may execute two hardware threads (or more, depending on the processor).	An SM may contain dozens or hundreds of streaming processors (SPs). Each SP can only execute one thread.
Caches	Each core has L2 and L2 caches, and may share an L3 cache.	Each SM has a cache, and may share an L2 cache between them.
Clock	High (e.g., 3.4 GHz).	Relatively lower (e.g., 1.0 GHz).

Custom tools must be used for GPU observability. Possible GPU performance metrics include the instructions per cycle, cache hit ratios, and memory bus utilization.

Other Accelerators

Apart from GPUs, be aware that other accelerators may exist for offloading CPU work to faster application-specific integrated circuits. These include field-programmable gate arrays (FPGAs)

[5] Nvidia also calls these *CUDA cores* [Verma 20].

and tensor processing units (TPUs). If in use, their usage and performance should be analyzed alongside CPUs, although they typically require custom tooling.

GPUs and FPGAs are used to improve the performance of cryptocurrency mining.

6.4.2 Software

Kernel software to support CPUs includes the scheduler, scheduling classes, and the idle thread.

Scheduler

Key functions of the kernel CPU scheduler are shown in Figure 6.12.

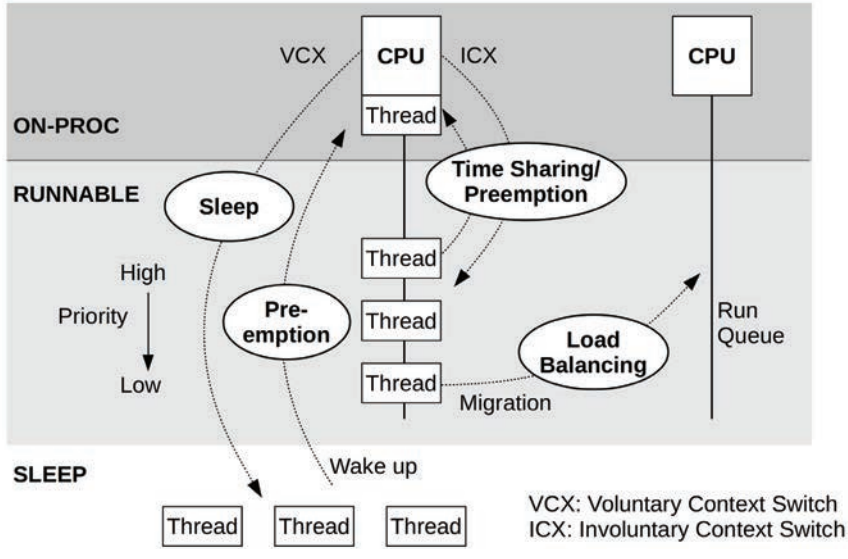


Figure 6.12 Kernel CPU scheduler functions

These functions are:

- **Time sharing**: Multitasking between runnable threads, executing those with the highest priority first.
- **Preemption**: For threads that have become runnable at a high priority, the scheduler can preempt the currently running thread, so that execution of the higher-priority thread can begin immediately.
- **Load balancing**: Moving runnable threads to the run queues of idle or less-busy CPUs.

Figure 6.12 shows run queues, which is how scheduling was originally implemented. The term and mental model are still used to describe waiting tasks. However, the Linux CFS scheduler actually uses a red/black tree of future task execution.

In Linux, time sharing is driven by the system timer interrupt by calling scheduler_tick(), which calls scheduler class functions to manage priorities and the expiration of units of CPU time called *time slices*. Preemption is triggered when threads become runnable and the scheduler class check_preempt_curr() function is called. Thread switching is managed by __schedule(), which selects the highest-priority thread via pick_next_task() for running. Load balancing is performed by the load_balance() function.

The Linux scheduler also uses logic to avoid migrations when the cost is expected to exceed the benefit, preferring to leave busy threads running on the same CPU where the CPU caches should still be warm (CPU affinity). In the Linux source, see the idle_balance() and task_hot() functions.

Note that all these function names may change; refer to the Linux source code, including documentation in the Documentation directory, for more detail.

Scheduling Classes

Scheduling classes manage the behavior of runnable threads, specifically their priorities, whether their on-CPU time is *time-sliced*, and the duration of those *time slices* (also known as *time quanta*). There are also additional controls via scheduling *policies*, which may be selected within a scheduling class and can control scheduling between threads of the same priority. Figure 6.13 depicts them for Linux along with the thread priority range.

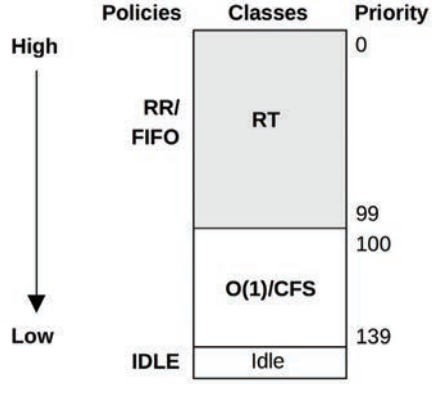


Figure 6.13 Linux thread scheduler priorities

The priority of user-level threads is affected by a user-defined *nice* value, which can be set to lower the priority of unimportant work (so as to be *nice* to other system users). In Linux, the nice value sets the *static priority* of the thread, which is separate from the *dynamic priority* that the scheduler calculates.

For Linux kernels, the scheduling classes are:

- **RT**: Provides fixed and high priorities for real-time workloads. The kernel supports both user- and kernel-level preemption, allowing RT tasks to be dispatched with low latency. The priority range is 0–99 (MAX_RT_PRIO-1).

- **O(1)**: The O(1) scheduler was introduced in Linux 2.6 as the default time-sharing scheduler for user processes. The name comes from the algorithm complexity of O(1) (see Chapter 5, Applications, for a summary of big O notation). The prior scheduler contained routines that iterated over all tasks, making it O(n), which became a scalability issue. The O(1) scheduler dynamically improved the priority of I/O-bound over CPU-bound workloads, to reduce the latency of interactive and I/O workloads.
- **CFS**: Completely fair scheduling was added to the Linux 2.6.23 kernel as the default time-sharing scheduler for user processes. The scheduler manages tasks on a red-black tree keyed from the task CPU time, instead of traditional run queues. This allows low CPU consumers to be easily found and executed in preference to CPU-bound workloads, improving the performance of interactive and I/O-bound workloads.
- **Idle**: Runs threads with the lowest possible priority.
- **Deadline**: Added to Linux 3.14, applies earliest deadline first (EDF) scheduling using three parameters: *runtime*, *period*, and *deadline*. A task should receive runtime microseconds of CPU time every period microseconds, and do so within the deadline.

To select a scheduling class, user-level processes select a *scheduling policy* that maps to a class, using either the sched_setscheduler(2) syscall or the chrt(1) tool.

Scheduler policies are:

- **RR**: SCHED_RR is round-robin scheduling. Once a thread has used its time quantum, it is moved to the end of the run queue for that priority level, allowing others of the same priority to run. Uses the RT scheduling class.
- **FIFO**: SCHED_FIFO is first-in, first-out scheduling, which continues running the thread at the head of the run queue until it voluntarily leaves, or until a higher-priority thread arrives. The thread continues to run, even if other threads of the same priority are on the run queue. Uses the RT class.
- **NORMAL**: SCHED_NORMAL (previously known as SCHED_OTHER) is time-sharing scheduling and is the default for user processes. The scheduler dynamically adjusts priority based on the scheduling class. For O(1), the time slice duration is set based on the static priority: longer durations for higher-priority work. For CFS, the time slice is dynamic. Uses the CFS scheduling class.
- **BATCH**: SCHED_BATCH is similar to SCHED_NORMAL, but with the expectation that the thread will be CPU-bound and should not be scheduled to interrupt other I/O-bound interactive work. Uses the CFS scheduling class.
- **IDLE**: SCHED_IDLE uses the Idle scheduling class.
- **DEADLINE**: SCHED_DEADLINE uses the Deadline scheduling class.

Other classes and policies may be added over time. Scheduling algorithms have been researched that are *hyperthreading-aware* [Bulpin 05] and *temperature-aware* [Otto 06], which optimize performance by accounting for additional processor factors.

When there is no thread to run, a special *idle task* (also called *idle thread*) is executed as a placeholder until another thread is runnable.

Idle Thread

Introduced in Chapter 3, the kernel "idle" thread (or *idle task*) runs on-CPU when there is no other runnable thread and has the lowest possible priority. It is usually programmed to inform the processor that CPU execution may either be halted (halt instruction) or throttled down to conserve power. The CPU will wake up on the next hardware interrupt.

NUMA Grouping

Performance on NUMA systems can be significantly improved by making the kernel *NUMA-aware*, so that it can make better scheduling and memory placement decisions. This can automatically detect and create groups of localized CPU and memory resources and organize them in a topology to reflect the NUMA architecture. This topology allows the cost of any memory access to be estimated.

On Linux systems, these are called *scheduling domains*, which are in a topology beginning with the *root domain*.

A manual form of grouping can be performed by the system administrator, either by binding processes to run on one or more CPUs only, or by creating an exclusive set of CPUs for processes to run on. See Section 6.5.10, CPU Binding.

Processor Resource-Aware

The CPU resource topology can also be understood by the kernel so that it can make better scheduling decisions for power management, hardware cache usage, and load balancing.

6.5 Methodology

This section describes various methodologies and exercises for CPU analysis and tuning. Table 6.7 summarizes the topics.

Table 6.7 **CPU performance methodologies**

Section	Methodology	Types
6.5.1	Tools method	Observational analysis
6.5.2	USE method	Observational analysis
6.5.3	Workload characterization	Observational analysis, capacity planning
6.5.4	Profiling	Observational analysis
6.5.5	Cycle analysis	Observational analysis
6.5.6	Performance monitoring	Observational analysis, capacity planning
6.5.7	Static performance tuning	Observational analysis, capacity planning
6.5.8	Priority tuning	Tuning
6.5.9	Resource controls	Tuning
6.5.10	CPU binding	Tuning
6.5.11	Micro-benchmarking	Experimental analysis

See Chapter 2, Methodologies, for more methodologies and the introduction to many of these. You are not expected to use them all; treat this as a cookbook of recipes that may be followed individually or used in combination.

My suggestion is to use the following, in this order: performance monitoring, the USE method, profiling, micro-benchmarking, and static performance tuning.

Section 6.6, Observability Tools, and later sections, show the operating system tools for applying these methodologies.

6.5.1 Tools Method

The tools method is a process of iterating over available tools, examining key metrics that they provide. While this is a simple methodology, it can overlook issues for which the tools provide poor or no visibility, and it can be time-consuming to perform.

For CPUs, the tools method can involve checking the following (Linux):

- **`uptime/top`**: Check the load averages to see if load is increasing or decreasing over time. Bear this in mind when using the following tools, as load may be changing during your analysis.
- **`vmstat`**: Run vmstat(1) with a one-second interval and check the system-wide CPU utilization ("us" + "sy"). Utilization approaching 100% increases the likelihood of scheduler latency.
- **`mpstat`**: Examine statistics per-CPU and check for individual hot (busy) CPUs, identifying a possible thread scalability problem.
- **`top`**: See which processes and users are the top CPU consumers.
- **`pidstat`**: Break down the top CPU consumers into user- and system-time.
- **`perf/profile`**: Profile CPU usage stack traces for both user- or kernel-time, to identify why the CPUs are in use.
- **`perf`**: Measure IPC as an indicator of cycle-based inefficiencies.
- **`showboost/turboboost`**: Check the current CPU clock rates, in case they are unusually low.
- **`dmesg`**: Check for CPU temperature stall messages ("cpu clock throttled").

If an issue is found, examine all fields from the available tools to learn more context. See Section 6.6, Observability Tools, for more about each tool.

6.5.2 USE Method

The USE method can be used to identify bottlenecks and errors across all components early in a performance investigation, before trying deeper and more time-consuming strategies.

For each CPU, check for:

- **Utilization**: The time the CPU was busy (not in the idle thread)
- **Saturation**: The degree to which runnable threads are queued waiting their turn on-CPU
- **Errors**: CPU errors, including correctable errors