Qualcomm Linux Performance Guide

80-70018-10 AA

April 4, 2025

Contents

1 Performance overview

This guide explains how to measure, fine-tune, and enhance the performance of Qualcomm® Linux®. Additionally, it discusses:

- Features that impact performance
- Tools to identify and analyze performance issues
- Options to configure and customize the Linux settings for enhanced performance
- Methods to troubleshoot performance issues
- Procedures to measure performance
- Performance dashboards for QCS6490 and QCS5430

Note: See Hardware SoCs that are supported on Qualcomm Linux.

The following QCS9075 and QCS8275-specific guides contain supplementary information and are available to licensed users with authorized access:

- Qualcomm Linux Performance Guide - Addendum for QCS9075
- Qualcomm Linux Performance Guide - Addendum for QCS8275

These guides describe device specifications, supported resource opcodes, chipset-specific configuration and customization settings, performance dashboards, and measurement procedures.

1.1 Subsystem dependencies

The performance of the software depends on the CPU, GPU, and DDR subsystems. Qualcomm Linux uses the Qualcomm® Kryo™ CPU, which has the following clusters:

- Prime cluster for high-performance CPU cores
- Gold cluster for balanced power and performance
- Silver cluster for low-power CPU cores, ideal for light-weight applications

Cache memory is categorized into three levels: L1, L2, and L3:

- L1 is the smallest and fastest cache level, storing both instructions (L1 I) and data (L1 D)
- L2 and L3 are larger but slower cache levels, mainly for data storage

The following tables list the specifications for the subsystems on QCS6490 and QCS5430:

QCS6490

Table : Specifications for QCS6490 CPU, GPU, and DDR subsystems

Specifications	**QCS6490**		
Core type	**Kryo Prime**	**Kryo Gold**	**Kryo Silver**
Number of CPUs	1	3	4
CPU maximum frequency	2.7 GHz	2.4 GHz	1.9 GHz
L1 I cache	32 kB	32 kB/core	32 kB/core
L1 D cache	32 kB	32 kB/core	32 kB/core
L2 cache	256 kB	256 kB/core	128 kB/core
L3 cache	2 MB		
GPU	Qualcomm® Adreno™ 643 GPU		
GPU maximum frequency	812 MHz		
DDRSS	• Supports dual-channel non-package-on-package high-speed LPDDR5/LPDDR4 SDRAM. • LPDDR5 SDRAM is designed for a 3200 MHz clock (2 $\times$ 16-bit). • LPDDR4 SDRAM is designed for a 2133 MHz clock (2 $\times$ 16-bit).		

QCS5430

Table : Specifications for QCS5430 (FP 1 and FP 2) CPU, GPU, and DDR subsystems

Specifications	**QCS5430 FP 1 (feature pack 1)**		**QCS5430 FP 2**		
Core type	**Kryo Gold**	**Kryo Silver**	**Kryo Prime**	**Kryo Gold**	**Kryo Silver**
Number of CPUs	2	4	1	3	4
CPU maximum frequency	2.1 GHz	1.8 GHz	2.2 GHz	2.1 GHz	1.8 GHz
L1 I cache	32 kB/core	32 kB/core	32 kB	32 kB/core	32 kB/core
L2 cache	256 kB/core	128 kB/core	256 kB	256 kB/core	128 kB/core
L3 cache	2 MB				
GPU	Qualcomm® Adreno™ 642L GPU				
GPU maximum frequency	315 MHz				
DDRSS	• Supports dual-channel non-package-on-package high-speed LPDDR5/LPDDR4 SDRAM. • LPDDR5 SDRAM is designed for a 3200 MHz clock (2 $\times$ 16-bit). • LPDDR4 SDRAM is designed for a 2133 MHz clock (2 $\times$ 16-bit).				

Table : Specifications for QCS5430 (FP 2.5 and FP 3) CPU, GPU, and DDR subsystems

Specifications	**QCS5430 FP 2.5**			**QCS5430 FP 3**		
Core type	**Kryo Prime**	**Kryo Gold**	**Kryo Silver**	**Kryo Prime**	**Kryo Gold**	**Kryo Silver**
Number of CPUs	1	3	4	1	3	4

Specifications	**QCS5430 FP 2.5**			**QCS5430 FP 3**		
Core type	**Kryo Prime**	**Kryo Gold**	**Kryo Silver**	**Kryo Prime**	**Kryo Gold**	**Kryo Silver**
CPU maximum frequency	2.38 GHz	2.4 GHz	1.8 GHz	2.38 GHz	2.4 GHz	1.8 GHz
L1 I cache	32 kB	32 kB/core	32 kB/core	32 kB	32 kB/core	32 kB/core
L2 cache	256 kB	256 kB/core	128 kB/core	256 kB	256 kB/core	128 kB/core
L3 cache	2 MB					
GPU	Qualcomm Adreno 642L GPU					
GPU maximum frequency	550 MHz					
DDRSS	• Supports dual-channel non-package-on-package high-speed LPDDR5/LPDDR4 SDRAM. • LPDDR5 SDRAM is designed for a 3200 MHz clock (2 $\times$ 16-bit). • LPDDR4 SDRAM is designed for a 2133 MHz clock (2 $\times$ 16-bit).					

2 Get started with performance tuning and optimization

To start developing your software using Qualcomm® Linux®, set up your infrastructure as described in the Qualcomm Linux Build Guide. The build guide also explains the common build workflows.

You can use the performance build to measure system performance. Additionally, you can compile the debug tools to analyze and effectively resolve any performance issues.

The following figure shows a workflow to achieve the expected system performance:

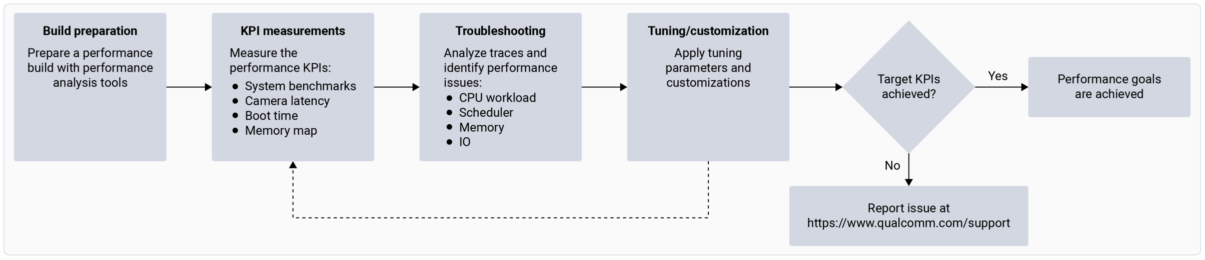

Build preparation
Prepare a performance build with performance analysis tools
KPI measurements
Measure the performance KPIs:
System benchmarks
Camera latency
Boot time
Memory map
Troubleshooting
Analyze traces and identify performance issues:
CPU workload
Scheduler
Memory
IO
Tuning/customization
Apply tuning parameters and customizations
Target KPIs achieved?
Yes
Performance goals are achieved
No
Report issue at https://www.qualcomm.com/support

Figure : Qualcomm Linux performance workflow

2.1 Prepare performance build

By default, the Qualcomm Linux build is the performance build. It's recommended to use this build for performance measurement or debugging.

The performance build uses the following kernel configuration fragments from the source code:

- `<kernel_src>/arch/arm64/configs/qcom_defconfig`
- `<kernel_src>/arch/arm64/configs/qcom_addons.config`

These kernel configurations are defined in the source code kernel recipe at:

- Base BSP: `layers/meta-qcom-hwe/recipes-kernel/linux/linux-qcom-base_6.6.bb`
- Custom BSP: `layers/meta-qcom-hwe/recipes-kernel/linux/linux-qcom-custom_6.6.bb`

```
KERNEL_CONFIG ??= "qcom_defconfig"
KERNEL_CONFIG_FRAGMENTS:append = " ${S}/arch/arm64/configs/qcom_
addons.config"
```

2.2 Compile performance tools

The tools to debug performance issues include LTTng, GCC, G++, htop, perf utility, iotop, and lmbench.

To compile the debug tools, add them to the `qcom-multimedia-image.bb` distribution image in the source code at the following path:

`layers/meta-qcom-distro/recipes-products/images/`

```
diff --git a/recipes-products/images/qcom-multimedia-image.bb b/
recipes-products/images/qcom-multimedia-image.bb
index 23a2f40..5295bf9 100644
--- a/recipes-products/images/qcom-multimedia-image.bb
+++ b/recipes-products/images/qcom-multimedia-image.bb
@@ -10,4 +10,20 @@ REQUIRED_DISTRO_FEATURES += "wayland"
  CORE_IMAGE_BASE_INSTALL += " \
      packagegroup-qcom-multimedia \
      packagegroup-qcom-iot-base-utils \
+     lttng-tools \
+     lttng-tools-dev \
+     lttng-modules \
+     lttng-modules-dev \
+     lttng-ust \
+     lttng-ust-dev \
+     liburcu-dev \
+     gcc \
+     g++ \
+     iotop \
+     htop \
+     perf \
+     lmbench \
"
```

2.3 Enable SSH

To set up SSH, see Sign in using SSH.

3 Features impacting performance

The Qualcomm® Linux® kernel includes features, such as the CPU scheduler, CPU frequency governor, dynamic voltage and frequency scaling (DVFS), and memory management. This guide provides an overview of each feature and related reference links. Additionally, Qualcomm has added a feature called PerfHAL to enhance the performance of Qualcomm Linux.

3.1 CPU scheduler

The CPU scheduler manages how the CPU time is distributed among the processes running on Linux systems.

It uses the earliest eligible virtual deadline first (EEVDF) scheduler, a feature provided by the Linux kernel. The EEVDF CPU scheduler uses per entity load tracking (PELT) to monitor the task load.

For more information, see:

- An EEVDF CPU scheduler for Linux
- Per-entity load tracking [LWN.net]

Utilization clamping (UCLAMP or util clamp) is a scheduler feature that helps manage performance requirements for tasks.

For more information, see:

- https://docs.kernel.org/scheduler/sched-util-clamp.html
- Customize CPU scheduler

3.2 CPU frequency governor

A CPU frequency governor adjusts the CPU frequency based on the task load. The CPU scheduler provides the necessary inputs for this process.

Qualcomm Linux uses the `schedutil` governor, a feature provided by the Linux kernel.

This governor increases the frequency when the system is heavily loaded and reduces it when the load is low, ensuring an optimal balance between power consumption and performance.

For more information, see:

- https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt
- Configure CPU
- Customize the CPU frequency governor

3.3 DVFS governors

DVFS governors control the frequencies of CPU caches (L3), the last level cache controller (LLCC), and the DDR based on the system workload.

These governors increase the frequency when the workload is high and decrease it when the workload is low, ensuring an optimal balance between power consumption and performance.

Qualcomm Linux supports the following two types of DVFS governors for L3 cache:

- LLCC
- DDR

Static map DVFS governor

This governor aligns the frequencies of the CPU L3 cache and the DDR with the current CPU frequency to balance the power and the performance requirements.

For example, if the CPU frequency is at its maximum, the L3 cache and DDR frequencies must also be at their maximum levels.

The static mapping is available in the source code at `arch/arm64/boot/dts/qcom/<target>.dtsi`.

For customization options, see Customize static map DVFS governor.

BWMON governor

The bandwidth monitoring (BWMON) governor dynamically adjusts the frequencies of the LLCC and DDR based on the measured traffic flow from the CPU to the LLCC and then to the DDR.

The BWMON hardware block measures this traffic. It monitors the data throughput between memory and the other subsystems within a specified sampling window and uses this information to scale the LLCC and DDR frequencies to meet the required bandwidth.

The BWMON governor driver is available in the source code at `drivers/soc/qcom/icc-bwmon.c`.

For more information, see:

- [PATCH v3 0/4] soc/arm64: qcom: Add initial version of bwmon

- Customize BWMON governor

3.4 PerfHAL

PerfHAL is a Qualcomm proprietary service that offers added functionality by making perflock APIs accessible. It's beneficial when you need short-term performance enhancements or power savings.

Perflocks help in modifying system behavior to manage intermittent workloads. For example, if a specific code segment must run at a higher CPU frequency for a certain duration, use perflocks within that code to boost the CPU frequency.

PerfHAL efficiently handles concurrent perflock requests from multiple clients. When several requests are aimed at the same resource, PerfHAL aggregates them to achieve the optimal performance level needed by the device.

When a perflock of a client is no longer active, PerfHAL releases all the perflocks associated with that client.

Perflock APIs

Perflock APIs allow applications to adjust system parameters for specific use cases, helping them meet their performance and power objectives.

User space applications use the `perf_lock_acq()` and `perf_lock_rel()` APIs to request specific values of system tunable parameters for both, a set time period or an indefinite time period.

Parameter	**Description**

Acquire perflock

Use `perf_lock_acq()` function to acquire a perflock with the necessary optimizations.

The syntax for this function is as follows:

```
int perf_lock_acq(int handle, int duration, int list[], int numArgs)
```

Table : perf_lock_acq API parameters

Parameter	**Description**
`handle`	Identifies the client request.
`duration`	• Indicates the maximum timeout period that a perflock must be held, in milliseconds. • `duration` parameter can be set for a definite time or for an indefinite time `(0)`. – Definite perflocks require a positive integer value to specify the maximum timeout period. A timer is created and the perflock is released when the timer expires. – Indefinite perflocks are held until the client calls the release function. To manually release a perflock that has been set for an indefinite duration, use the `perf_lock_rel()` function.
`list`	An array of resource opcodes and value pairs. Opcodes indicate a system parameter (resource) and the value to set it (level).
`numArgs`	Number of elements in the list array.

Table : perf_lock_acq API returns and result

Returns	Result
A non-zero integer	Success
-1	Failure

Perflock release

The `perf_lock_rel()` function is used to release a perflock that's held by the `perf_lock_acq()` API. Use this function only for the perflocks that are set for an indefinite time period. The syntax for this function is as follows:

```
int perf_lock_rel(int handle)
```

Table : perf_lock_rel API parameters

Parameter	Description
`handle`	• Tracks unique requests • Passes the same handle that `perf_lock_acq()` returns to release the lock

Table : perf_lock_rel API returns and result

Returns	Result
A non-zero integer	Success
-1	Failure

Resource opcodes

Perflock uses a combination of opcodes and their corresponding values to perform specific operations on a perflock resource.

To know the supported opcodes on QCS9075 and QCS8275, see the corresponding addendum. The following guides are available to licensed users with authorized access:

- Qualcomm Linux Performance Guide - Addendum for QCS9075
- Qualcomm Linux Performance Guide - Addendum for QCS8275

The following table lists the supported opcodes:

Table : Supported opcodes

Opcode	Purpose	Sysnode on device
0x44000000	Sets the minimum acceptable performance level for individual tasks and task groups.	`/proc/sys/kernel/sched_util_clamp_min`
0x44004000	Sets the maximum acceptable performance level for individual tasks and task groups.	`/proc/sys/kernel/sched_util_clamp_max`
0x44008100	Sets the minimum frequency of the Silver cluster.	`/sys/devices/system/cpu/cpufreq/policy0/scaling_min_freq`
0x44008000	Sets the minimum frequency of the Gold cluster.	`/sys/devices/system/cpu/cpufreq/policy4/scaling_min_freq`
0x44008200	Sets the minimum frequency of the Prime cluster.	`/sys/devices/system/cpu/cpufreq/policy7/scaling_min_freq`
0x4400C100	Sets the maximum frequency of the Silver cluster.	`/sys/devices/system/cpu/cpufreq/policy0/scaling_max_freq`
0x4400C000	Sets the maximum frequency of the Gold cluster.	`/sys/devices/system/cpu/cpufreq/policy4/scaling_max_freq`
0x4400C200	Sets the maximum frequency of the Prime cluster.	`/sys/devices/system/cpu/cpufreq/policy7/scaling_max_freq`

The following are some examples of the resource opcodes:

- 0x44008100, 1958400: This pair of opcode and value indicates that the minimum frequency of the Silver cluster must be set to 1958400 KHz.
- 0x44008100, 1958400, 0x4400C100, 2100000: This pair of opcode and value indicates that the minimum frequency of the Silver cluster must be set to 1958400 KHz. The maximum frequency of the Silver cluster must be set to 2100000 KHz.

For more information about how to use and debug perflock, see Customize perflock.

3.5 Memory

RAM is used for all memory allocations made by Qualcomm Linux. RAM must be effectively managed to meet performance requirements and ensure the smooth functioning of applications. The following figure shows memory partitioning:

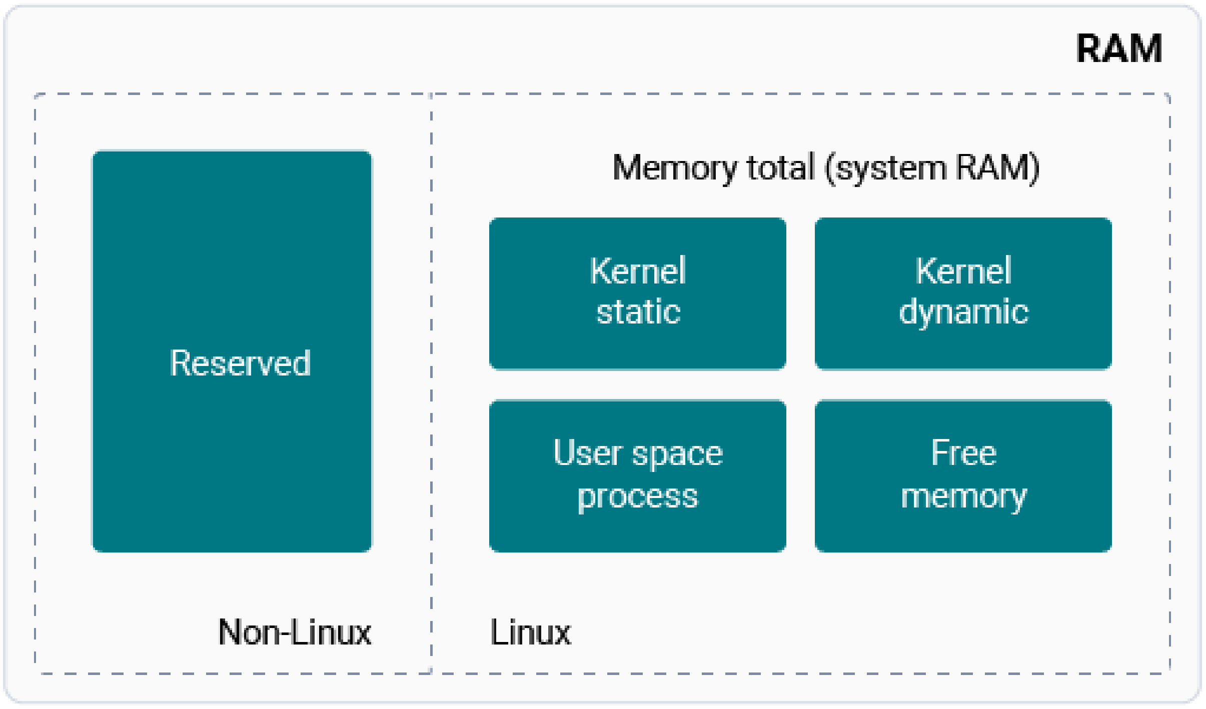


Figure : Memory partitioning

Certain sections of RAM are managed independent of the Linux system. For example, firmware such as modem, video, and audio run from these specific RAM partitions. The Linux kernel manages all other RAM partitions.

The Linux kernel features its own memory management subsystem, which includes:

- Implementation of virtual memory and demand paging
- Allocation of memory to both kernel internal structures and user space programs
- Mapping of files into the address space of the processes
- Other memory management operations

RAM memory partitioning

The following table describes various types of memory allocations.

Note: The commands specified in the following table should be run on the device.

Table : RAM classification

RAM classification	Memory segment	Allocation types	Description
Non-Linux	–	–	• Memory is reserved in the form of carveouts by various subsystems other than Linux. • These carveouts are specified in the respective DTSI files.
Linux (system RAM)	Kernel static	Vmlinux + kernel page structures	• The kernel reserves this memory at boot time for its own usage. • Vmlinux is the memory used to store the vmlinux image. • The size and breakdown of the vmlinux image can be obtained from the `dmesg` logs at boot: Memory: 3061872K/4134912K available (28800K kernel code, 2090K rwdata, 10688K rodata, 3072K init, 969K bss, 679824K reserved, 393216K cma-reserved) Kernel code + rwdata + rodata + init +bss indicates vmlinux kernel image size (28800k + 2090k + 10688k + 3072k + 969k) • The kernel page structure is the memory used by the kernel to maintain page structures for every page of RAM. This is calculated as 16 MB per GB of RAM size.

RAM classification	Memory segment	Allocation types	Description

	Kernel dynamic	Slab	• The slab is used by the kernel for faster and more efficient memory usage of frequently used data structures.
			• To understand the memory usage of the slab, run the following command:
			`cat /proc/meminfo | grep -i slab`
			• To understand the breakup of various slabs and their usage, enable `CONFIG_SLUB_DEBUG` in the kernel configuration, and then run the following command:
			`cat /proc/slabinfo`

		Kernel stack	• The kernel stack stores the call stack of every process. • To understand the memory usage of the kernel stack, run the following command: `cat /proc/meminfo | grep -i kernelstack`
		PageTables	• The kernel uses memory to store PageTables that map virtual addresses to physical addresses. • To understand the memory usage of PageTables, run the following command: `cat /proc/meminfo | grep -i PageTables`

RAM classification	Memory segment	Allocation types	Description
		Modules	• Represents the kernel entities that are dynamically loaded into the kernel in the form of kernel modules. • To display the list of loaded kernel modules and their memory usage, run the following command: `cat /proc/modules`
		Vmalloc	• Used to allocate contiguous memory. • To understand the Vmalloc memory breakup, run the following command: `cat /proc/vmallocinfo`
		Cached (kernel + user space)	• The amount of file-backed memory that resides in RAM. • To understand the cached memory usage, run the following command: `cat /proc/meminfo | grep -i cached`
		Buffers	• Buffers are of fixed size and contain blocks of information either read from disk or written to disk. • To understand the buffer memory usage, run the following command: `cat /proc/meminfo | grep -i Buffers`

RAM classification	Memory segment	Allocation types	Description
		Shmem	• Shared memory is a common block of memory that's mapped into the address spaces of two or more processes. • To understand the shared memory usage, run the following command: `cat /proc/meminfo | grep -i shmem`
	User space	ZUSED (ZRAM)	An anonymous memory post compression by ZRAM.
		CMA	• A physically continuous memory is typically mapped to other IPs, such as video and display, however it's allocated to the runtime. • The free memory that the system can use is reduced with the usage of more CMA reservations. Only the movable allocations, such as user space process allocations can use the CMA reserved free memory. However, it can't be used for the kernel allocations.
		ANON	• Memory that user space applications allocate using `malloc()` or `new()` function calls. • To understand the ANON memory breakup for a process, run the following command: `cat /proc/<pid>/smaps`

RAM classification	Memory segment	Allocation types	Description

		ION	• ION memory allows sharing buffers between hardware IPs such as video, camera, and Qualcomm Linux.
			• ION manages one or more memory pools, which can be set aside at boot time to combat fragmentation.
			• To understand the ION memory usage, run the following commands:
			`mount -t debugfs none /sys/kernel/debug`
			`cat /sys/kernel/debug/dma_buf/bufinfo | grep bytes`

		KGSL	• Memory allocated by the graphics driver.
			• To understand the overall kernel graphics support layer (KGSL) memory usage, run the following command:
			`cat /sys/class/kgsl/kgsl/page_alloc`
			• To understand the process level breakup, run the following command:
			`cat /sys/class/kgsl/kgsl/proc/<pid>/kernel`

RAM classification	Memory segment	Allocation types	Description
	Free memory	–	• Free memory is the memory that's not yet used and is available for any allocation. • To understand the free memory, run the following command: `cat /proc/meminfo | grep -i MemFree`

3.6 Real-time kernel

Real-time (RT) Linux is an optional feature that's not enabled by default on Qualcomm Linux. It can be enabled based on the product requirements.

RT Linux is designed to offer deterministic and predictable behavior for applications that are time-sensitive.

Set up workspace

In Qualcomm Linux, the RT Linux kernel recipes are referred to as follows:

- Base BSP: `linux-qcom-base-rt`
- Custom BSP: `linux-qcom-custom-rt`

The Qualcomm Linux kernel supports long-term support (LTS) RT kernel 6.6 version, which is maintained through the Yocto recipe in the `meta-qcom-realtime` layer at the following paths in the source code:

- Base BSP: `recipes-kernel/linux/linux-kernel-base-rt_6.6.bb`
- Custom BSP: `recipes-kernel/linux/linux-kernel-custom-rt_6.6.bb`

For more information about how to clone the workspace and acquire all the meta layers to use Qualcomm RT Linux kernel, see Sync and build with real-time Linux.

Enable RT kernel

Use the RT Linux kernel recipe to enable the RT kernel. This recipe fetches the kernel, downloads pre-empt RT patches and applies them to the kernel. It also allows a fully pre-emptible kernel with:

`CONFIG_PREEMPT_RT=y`

For more information, see Qualcomm Linux Kernel Guide.

Verify kernel type

After booting, verify the kernel type by running the following command on the device:

```
uname -v
```

The following is an output of the command:

```
SMP PREMPT_RT
```

Test RT Linux kernel

The RT Linux kernel test helps to obtain the following information:

- Real-time performance of the RT Linux kernel
- RT Linux kernel latencies and key performance indicators (KPIs)

Note: This section is only applicable for QCS6490.

Caution: Ensure that the system isn't rebooted during the RT Linux kernel test because this test runs for approximately more than 24 hours.

Cyclictest

Cyclictest tool is used for benchmarking the RT Linux kernel systems. It's used to evaluate the relative performance of the real-time systems. The Qualcomm Linux build has the cyclictest tool. For more information, see https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start/.

This guide describes the following cyclictests:

- Cyclictest with no-load: System load isn't added to perform this test.
- Cyclictest with stress-ng (next-generation): Specific percentage of load is added to perform this test to measure the worst case system latencies. For more information about stress-ng, see Kernel/Reference/stress-ng - Ubuntu Wiki.

Presetting

Ensure to complete the following presetting before you run a cyclictest:

1. Configure and isolate the CPU core 1 to core 3 for the RT tasks. You may configure any other CPU cores depending on your requirement.

   For example, you can configure the RT CPUs in the source code at the following path:

   `layers/meta-qcom-realtime/blob/scarthgap/conf/layer.conf`

   For example, configure the CPU as follows:

   ```
   KERNEL_CMDLINE_EXTRA:qcm6490 = "pcie_pme=nomsi net.ifnames=0
   pci=noaer kpti=off kasan=off kasan.stacktrace=off swiotlb=128
   mitigations=auto kernel.sched_pelt_multiplier=4 rcupdate.rcu_
   expedited=1 rcu_nocbs=1-3 isolcpus=1-3 irqaffinity=4-7 nohz_
   full=1-3 no-steal-acc vfio_iommu_type1.allow_unsafe_interrupts=1
   "
   ```

2. Run the following commands on the RT Linux kernel:

   ```
   echo 0 > /sys/kernel/tracing/tracing_on
   ```

   ```
   echo E0 > /sys/devices/virtual/workqueue/kgsl-workqueue/
   cpumask
   ```

   ```
   echo E0 > /sys/devices/virtual/workqueue/scsi_tmf_0/
   cpumask
   ```

   ```
   echo E0 > /sys/devices/virtual/workqueue/writeback/
   cpumask
   ```

```
echo 1 > /sys/devices/system/cpu/cpu0/cpuidle/state0/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu0/cpuidle/state1/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu0/cpuidle/state2/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu1/cpuidle/state0/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu1/cpuidle/state1/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu1/cpuidle/state2/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu2/cpuidle/state0/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu2/cpuidle/state1/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu2/cpuidle/state2/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu3/cpuidle/state0/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu3/cpuidle/state1/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu3/cpuidle/state2/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu4/cpuidle/state0/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu4/cpuidle/state1/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu4/cpuidle/state2/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu5/cpuidle/state0/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu5/cpuidle/state1/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu5/cpuidle/state2/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu6/cpuidle/state0/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu6/cpuidle/state1/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu6/cpuidle/state2/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu7/cpuidle/state0/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu7/cpuidle/state1/
disable
```

```
echo 1 > /sys/devices/system/cpu/cpu7/cpuidle/state2/
disable
```

```
echo performance > /sys/devices/system/cpu/cpufreq/
policy7/scaling_governor
```

```
echo performance  > /sys/devices/system/cpu/cpufreq/
policy4/scaling_governor
```

```
echo performance > /sys/devices/system/cpu/cpufreq/
policy0/scaling_governor
```

```
echo +cpuset > /sys/fs/cgroup/cgroup.subtree_control
```

```
mkdir /sys/fs/cgroup/cpuset
```

```
echo +cpuset > /sys/fs/cgroup/cpuset/cgroup.subtree_
control
```

Cyclictest with no-load

To run a cyclictest with no-load, follow these steps:

1. Complete the presetting.
2. Run the following commands to start cyclictest:

   ```
   mkdir /sys/fs/cgroup/cpuset/core1-3/
   ```

   ```
   echo +cpuset > /sys/fs/cgroup/cpuset/core1-3/cgroup.
   subtree_control
   ```

   ```
   echo 1-3 > /sys/fs/cgroup/cpuset/core1-3/cpuset.cpus
   ```

   ```
   echo $$ > /sys/fs/cgroup/cpuset/core1-3/cgroup.procs
   ```

   ```
   cyclictest -a 1-3 -t 3 -m -l 100000000 -i 1000 -p 90 -
   h 800 --mainaffinity 4 --spike 100
   ```

3. Note the latencies.

Cyclictest with stress-ng

To run a cyclictest with stress-ng, follow these steps:

1. Complete the presetting.
2. Open a shell and run the following commands to run stress-ng. In the second example command, the non-RT CPU is loaded with 60% load:

   ```
   mkdir /tmp/temp-path
   ```

   ```
   stress-ng --cpu 5 --cpu-load 60 --temp-path /tmp/
   temp-path --sched fifo --sched-prio 1 -t 2d
   ```

This procedure completes in approximately 48 hours. In this example, CPU 1, 4, and 7 are loaded.

3. Run the following commands to start cyclictest in another terminal to run the cyclictest and stress-ng simultaneously:

```
mkdir /sys/fs/cgroup/cpuset/core1-3/
```

```
echo +cpuset > /sys/fs/cgroup/cpuset/core1-3/
cgroup.subtree_control
```

```
echo 1-3 > /sys/fs/cgroup/cpuset/core1-3/cpuset.cpus
```

```
echo $$ > /sys/fs/cgroup/cpuset/core1-3/cgroup.procs
```

```
cyclictest -a 1-3 -t 3 -m -l 100000000 -i 1000 -p 90 -h
800 --mainaffinity 4 --spike 100
```

4. Use *Ctrl + C* to stop stress-ng.

5. Note the worst-case latencies.

RT Linux kernel KPIs

The following tables describe the cyclictests KPIs:

KPIs for cyclictest with no-load

RT cores	Core1	Core2	Core3
Minimum latencies (in microseconds)	8	9	9
Maximum latencies (in microseconds)	86	29	29

KPIs for cyclictest with stress-ng

RT cores	Core1	Core2	Core3
Minimum latencies (in microseconds)	8	9	9
Maximum latencies (in microseconds)	53	37	28

4 Performance analysis tools

Examine the CPU performance with the analysis tools.

4.1 Function tracer (ftrace)

- This tool provides insights into the kernel's operations
- Function tracer is useful for debugging or analyzing latencies and performance issues
- Function tracer serves as a tracing tool to collect kernel space traces for analysis

To collect ftrace, do the following:

Note: The commands specified in the following steps should be run on the device.

1. To enable trace events, run the following commands from the SSH shell:

   ```
   mount -t debugfs none /sys/kernel/debug
   ```

   ```
   echo 0 > /sys/kernel/tracing/tracing_on
   ```

   ```
   echo 25600 > /sys/kernel/tracing/buffer_size_kb
   ```

   ```
   echo "" > /sys/kernel/tracing/set_event
   ```

   ```
   echo "" > /sys/kernel/tracing/trace
   ```

2. To understand the available events, run the following command:

   ```
   cat /sys/kernel/tracing/available_events
   ```

3. To enable the required events, for example, a scheduler trace, run the following commands:

   ```
   echo 1 > /sys/kernel/tracing/events/sched/sched_wakeup_new/
   enable
   ```

```
echo 1 > /sys/kernel/tracing/events/sched/sched_waking/enable
```

```
echo 1 > /sys/kernel/tracing/events/sched/sched_switch/enable
```

4. To understand the set events, run the following command:

   ```
   cat /sys/kernel/debug/tracing/set_event
   ```

5. To start a trace, run the following command:

   ```
   echo 1 > /sys/kernel/tracing/tracing_on
   ```

6. Run use cases.
7. To stop the trace after running use cases, run the following command:

   ```
   echo 0 > /sys/kernel/tracing/tracing_on
   ```

8. To save the trace, run the following command:

   ```
   cat /sys/kernel/tracing/trace > /opt/ftrace.txt
   ```

9. To pull ftrace from the target to the host, use secure copy protocol (SCP) or a similar tool. Ensure to specify the target IP address in the command. The following is an example command to run on the host machine:

   ```
   scp -r root@10.92.162.185:/opt/ftrace.txt /local/mnt/workspace/
   logs
   ```

For more information, see https://www.kernel.org/doc/html/v4.17/trace/ftrace.html.

4.2 Linux Trace Toolkit next generation (LTTng)

- LTTng is an open-source tracing tool.
- It's used to trace the Linux kernel and user space simultaneously.

For more information, see https://lttng.org/docs/v2.13/.

4.3 Trace Compass by Eclipse

- Trace Compass is an open-source tool that provides a graphical view for analyzing performance issues.
- It can parse several types of traces including ftrace and LTTng, and visualize the following:

  - Kernel resources
  - Kernel control flow
  - User space marker

Trace Compass is a host application that requires a Java virtual machine to work on the host machine. For more information, see https://projects.eclipse.org/projects/tools.tracecompass.

To download Trace Compass, see
https://projects.eclipse.org/projects/tools.tracecompass/downloads

4.4 Qualcomm Profiler

- Qualcomm® Profiler CLI is a command-line interface tool.
- It's used as a performance profiling tool to identify, measure, and optimize application scaling improvement opportunities across Qualcomm system-on-chip (SoC).
- It supports CPU, GPU, process, memory, and I/O profiling metrics.
- Download the Qualcomm Profiler tool on the host machine from the Qualcomm Software Center (QSC).
- To install the QSC, see Build with QSC Launcher.
- For information about how to use the Qualcomm Profiler tool, see Qualcomm Profiler CLI for Linux.

4.5 Perf utility

- The Perf utility is an open-source profiling tool in Linux to read performance monitoring unit (PMU) counters.
- It's used to obtain a function call stack.
- It's part of the Qualcomm® Linux® build.

To run the perf utility, run the following command on the device:

```
perf --help
```

For more information, see https://perf.wiki.kernel.org/index.php/Main_Page.

4.6 systemd-analyze

- Systemd-analyze is a command-line tool used to analyze and debug system boot performance in Linux.
- It measures boot time milestones and analyzes bottlenecks at different layers, such as the kernel, user space platform, and services.
- The Yocto system recipe comes with the systemd-analyze tool by default.

To compile and install the systemd-analyze utility, do the following:

- On host:
  1. To compile `systemd-analyze.ipk`, run the following command in your workspace:

     ```
     bitbake systemd
     ```

  2. The `.ipk` file is generated in the following directory:

     `/build-qcom-wayland/tmp-glibc/deploy/ipk/armv8-2a/ systemd-analyze_250.5-r0_armv8-2a.ipk`

  3. Push the `.ipk` file into the device from the host using SCP or a similar tool. The following is an example command to run on a host machine. Ensure to specify the target IP address in the command:

     ```
     adb shell mount -o remount,rw /usr
     ```

     ```
     scp systemd-analyze_250.5-r0_armv8-2a.ipk root@10.92.162.185:
     /usr/bin
     ```

- On target:

  Install the `.ipk` file by running the following commands:

  ```
  cd /usr/bin
  ```

  ```
  opkg --nodeps install --force-reinstall /usr/bin/systemd-
  analyze_250.5-r0_armv8-2a.ipk
  ```

For more information, see https://www.freedesktop.org/software/systemd/man/latest/systemd-analyze.html.

5 Configure CPU, GPU, and memory

It's essential to tune the basic configuration settings of your device before starting the performance analysis. The settings play a significant role in the performance of the device. You can configure the CPU, GPU, and memory settings.

Caution: Any configuration changes can impact the power and the performance of the device. Ensure that you verify the impact across all relevant use cases before any modifications.

5.1 Configure CPU

You can check and modify the CPU configurations using the commands specified in the following table:

Note: The commands specified in the following table should be run on the device. The unit of the output value for these commands is in KHz.

Note: The commands containing `policy7` aren't supported on QCS5430 FP1.

To configure a CPU for QCS9075 and QCS8275, see the corresponding addendum. The following guides are available to licensed users with authorized access:

- Qualcomm Linux Performance Guide - Addendum for QCS9075
- Qualcomm Linux Performance Guide - Addendum for QCS8275

Table : Commands to configure the CPU

Command	Purpose
`cat /sys/devices/system/cpu/online`	Checks the online CPU cores.

Command	Purpose
`echo 1 > /sys/devices/system/cpu/cpuX/online`	Turns on a CPU core. In cpuX, X represents the number of cores, which ranges from 0 to 7.
`echo 0 > /sys/devices/system/cpu/cpuX/online`	Turns off a CPU core. In cpuX, X represents the number of cores, which ranges from 0 to 7.

`cat /sys/devices/system/cpu/cpufreq/policy0/scaling_cur_freq`
`cat /sys/devices/system/cpu/cpufreq/policy4/scaling_cur_freq`
`cat /sys/devices/system/cpu/cpufreq/policy7/scaling_cur_freq`
	Reads the current frequency of the CPU.

`cat /sys/devices/system/cpu/cpufreq/policy0/scaling_available_frequencies`
`cat /sys/devices/system/cpu/cpufreq/policy4/scaling_available_frequencies`
`cat /sys/devices/system/cpu/cpufreq/policy7/scaling_available_frequencies`
	Reads the supported frequencies of the CPU.

`cat /sys/devices/system/cpu/cpufreq/policy0/scaling_min_freq`
`cat /sys/devices/system/cpu/cpufreq/policy4/scaling_min_freq`
`cat /sys/devices/system/cpu/cpufreq/policy7/scaling_min_freq`
	Reads the minimum frequency of the CPU.

Command	Purpose

`echo <cpu freq in KHz> > /sys/devices/system/cpu/cpufreq/policy0/scaling_min_freq`
`echo <cpu freq in KHz> > /sys/devices/system/cpu/cpufreq/policy4/scaling_min_freq`
`echo <cpu freq in KHz> > /sys/devices/system/cpu/cpufreq/policy7/scaling_min_freq`
	Sets the minimum frequency of the CPU. Replace `<cpu freq in KHz>` with the required frequency and run the commands.

`cat /sys/devices/system/cpu/cpufreq/policy0/scaling_max_freq`
`cat /sys/devices/system/cpu/cpufreq/policy4/scaling_max_freq`
`cat /sys/devices/system/cpu/cpufreq/policy7/scaling_max_freq`
	Reads the maximum frequency of the CPU.

`echo <cpu freq in KHz> > /sys/devices/system/cpu/cpufreq/policy0/scaling_max_freq`
`echo <cpu freq in KHz> > /sys/devices/system/cpu/cpufreq/policy4/scaling_max_freq`
`echo <cpu freq in KHz> > /sys/devices/system/cpu/cpufreq/policy7/scaling_max_freq`
	Sets the maximum frequency of the CPU. Replace `<cpu freq in KHz>` with the required frequency and run the commands.

Command	Purpose

`cat /sys/devices/system/cpu/cpufreq/policy0/stats/trans_table`
`cat /sys/devices/system/cpu/cpufreq/policy4/stats/trans_table`
`cat /sys/devices/system/cpu/cpufreq/policy7/stats/trans_table`
	Checks the CPU residency.

For example, to set the CPU frequency of the Silver core at 1.5 GHz, run the following commands:
`echo 1516800 > /sys/devices/system/cpu/cpufreq/policy0/scaling_min_freq`
`echo 1516800 > /sys/devices/system/cpu/cpufreq/policy0/scaling_max_freq`
	Sets the CPU frequency. Set `scaling_min_freq` and `scaling_max_freq` to the same frequency to keep the CPU frequency at the required level.

5.2 Configure GPU

You can check and modify the GPU configurations on the device using the commands specified in the following table:

Note: The commands specified in the following table should be run on the device. The unit of the output value for these commands is in Hz.

Table : Commands to configure the GPU

Command	Purpose
`cat /sys/class/kgsl/kgsl-3d0/gpuclk`	Reads the current frequency of the GPU.

Command	**Purpose**
`cat /sys/class/kgsl/kgsl-3d0/gpu_available_frequencies`	Reads the supported GPU frequencies.
`cat /sys/class/kgsl/kgsl-3d0/devfreq/min_freq`	Reads the minimum frequency of the GPU.
`echo <GPU freq in Hz> > /sys/class/kgsl/kgsl-3d0/devfreq/min_freq`	Sets the minimum frequency of the GPU. Replace `<GPU freq in Hz>` with the required frequency in Hz and run the command.
`cat /sys/class/kgsl/kgsl-3d0/devfreq/max_freq`	Reads the maximum frequency of the GPU.
`echo <GPU freq in Hz> /sys/class/kgsl/kgsl-3d0/devfreq/max_freq`	Sets the maximum frequency of the GPU. Replace `<GPU freq in Hz>` with the required frequency in Hz and run the command.
`cat /sys/class/kgsl/kgsl-3d0/gpu_busy_percentage`	Reads the GPU percentage usage.

For example, to set the GPU frequency at 600000000 Hz, run the following commands:
`echo 600000000 > /sys/class/kgsl/kgsl-3d0/devfreq/min_freq`
`echo 600000000 > /sys/class/kgsl/kgsl-3d0/devfreq/max_freq`
	Sets the GPU frequency at a required level, both `min_freq` and `max_freq` should be set to the same frequency level.

`cat /sys/class/kgsl/kgsl-3d0/max_pwrlevel`
`cat /sys/class/kgsl/kgsl-3d0/min_pwrlevel`
	Reads the power levels of the GPU.
	Level 0 is mapped to the maximum GPU clock. The higher the level, the lower is the GPU clock.

Command	Purpose
`echo 0 > /sys/class/kgsl/kgsl-3d0/min_pwrlevel`	Sets the GPU frequency to maximum frequency.

5.3 Configure memory

Memory configuration is crucial to conserve memory, make space available for processes, and balance the existing data in RAM when the physical memory reaches its limits.

Set ZRAM disk size

ZRAM is a RAM partition used for swapping memory space. It stores the inactive anonymous pages in a compressed form in RAM to conserve memory.

The ZRAM disk size indicates the maximum amount of memory that can be swapped from RAM to ZRAM. It's recommended to set the ZRAM disk size to half the size of RAM.

For example, to configure the ZRAM disk size to 1 GB for a RAM size of 2 GB, run the following commands on the device:

```
swapoff /dev/zram0
```

```
echo 1 > /sys/block/zram0/reset
```

```
echo 1073741824 > /sys/block/zram0/disksize
```

```
mkswap /dev/zram0
```

```
swapon /dev/zram0
```

```
cat /proc/meminfo | grep -i "swap"
```

The following is the command output:

```
SwapCached: 0 kB
SwapTotal: 1048572 kB
SwapFree: 1048572 kB
```

You can configure the ZRAM disk size at compile time by modifying the `-s` argument value in the `zram-swap-init-update` bash script in the source code at the following path:

`layers/meta-qcom-hwe/dynamic-layers/openembedded-layer/recipes-extended/zram/zram`

To change the size in the script, update the `<size>` parameter:

```
zramctl -a ${ZRAM_ALGORITHM} -s <size> $device
```

The following example shows the ZRAM size set to 1 GB:

```
zramctl -a ${ZRAM_ALGORITHM} -s 1048576KB $device
```

Enable/disable proactive compaction

Compaction is a process that collects all the free memory spaces in the form of fragments into a large memory block. Proactive compaction determines how aggressively compaction happens in the background. It takes a value between 0 and 100, with the default value set to 20.

When the fragmentation score of the node exceeds a high threshold, the corresponding kcompactd thread initiates the proactive compaction process.

The compaction process remains active until the score of the node falls below the low threshold.

The thresholds for proactive compaction are as follows:

- Low = 100 - proactiveness
- High = low + 10%

For instance, if the low threshold is 80, the high threshold would be 90.

To enable proactive compaction, run the following command on the device:

```
echo 20 > /proc/sys/vm/compaction_proactiveness
```

To disable proactive compaction, which also turns off the periodic kcompactd wake-ups, run the following command on the device:

```
echo 0 > /proc/sys/vm/compaction_proactiveness
```

For more information, see Proactive Compaction for the Kernel.

Swappiness

Swap space is a secondary storage that functions similar to the main memory or RAM. The primary parameter of swap space is swappiness.

- This parameter determines how aggressively the kernel swap daemon (kswapd) swaps out anonymous memory relative to the system page cache.
- This parameter can take a value between 0 and 200. Increasing the value augments the amount of anonymous swapping. The configured value is 100, indicating that both anonymous and cache memories are reclaimed equally.

To adjust the swappiness value, run the following command on the device:

```
echo 100 > /proc/sys/vm/swappiness
```

For more information about memory parameters, see https://www.kernel.org/doc/Documentation/admin-guide/sysctl/vm.rst.

6 Customize for performance tuning

Customization is a process that includes fine-tuning various aspects of the system, which can significantly affect the overall performance and power of the system.

The CPU scheduler, CPU frequency governor, dynamic voltage and frequency scaling (DVFS) governor, perflock, and memory can be fine-tuned. It's recommended to undertake any tuning only after gaining a thorough understanding through extensive performance and power analysis.

Caution: Any customization can impact the power and the performance of the device. Therefore, it's crucial to verify the impact across all the relevant use cases before performing any customization.

6.1 Customize CPU scheduler

You can customize features of the CPU scheduler, such as utilization clamping (UCLAMP or util clamp) schedulers.

For more information, see CPU scheduler.

UCLAMP

UCLAMP is a scheduler feature that allows user space to manage the task performance requirements.

It's a hinting mechanism that helps the scheduler understand the performance demands and limitations of tasks, as a result, it assists the scheduler in making informed decisions.

When the `schedutil` CPU frequency governor is used, UCLAMP determines the CPU frequency selection. The UCLAMP value ranges from 0 to 1024.

The following parameters can be customized for UCLAMP:

- `sched_util_clamp_min`: This parameter sets the minimum acceptable performance level for individual tasks and task groups, ensuring that tasks receive sufficient resources to operate effectively, even during periods of low demand.

  Any requested `uclamp.min` value for a task can't exceed `sched_util_clamp_min`.

  - For the scheduler, it acts as a lower bound on the per entity load tracking (PELT) signal, which tracks task usage.
  - For the CPU frequency, it instructs the governor to select a frequency that can meet the performance requirements of the task, thus ensuring responsiveness and efficiency. The Qualcomm-tuned value is 1024. To set this parameter, run the following command on the device:

```
echo 1024 > /proc/sys/kernel/sched_util_clamp_min
```

- `sched_util_clamp_max`: This parameter sets the maximum acceptable performance level for individual tasks and task groups. It ensures that tasks don't consume excessive resources, preventing resource contention and system instability.

  Any requested `uclamp.max` value for a task can't exceed `sched_util_clamp_max`.

  - For the scheduler, it acts as a ceiling on the PELT signal, which tracks the task usage.
  - For the CPU frequency, if the task demands exceed the available frequency, the governor may adjust the frequency to prevent excessive power consumption. The Qualcomm-tuned value is 1024. To set this parameter, run the following command on the device:

```
echo 1024 > /proc/sys/kernel/sched_util_clamp_max
```

- `sched_util_clamp_min_rt_default`: By default, the real-time (RT) tasks always run at the highest frequency and highest CPU capacity. This parameter allows you to change the default behavior of an RT task when UCLAMP is being used.

  It allows tuning the best value for an RT task, offering good performance without pushing it to the maximum performance point. This behavior addresses the system requirement without burning power and running at the maximum performance point all the time. The Qualcomm-tuned value is 128.

  To set this parameter, run the following command on the device:

```
echo 128 > /proc/sys/kernel/sched_util_clamp_min_rt_default
```

For more information, see https://docs.kernel.org/scheduler/sched-util-clamp.html.

6.2 Customize CPU frequency governor

You can configure a CPU frequency governor using the `scaling_governor` node to enhance CPU performance.

Note: The commands containing `policy7` aren't supported on QCS5430 FP1.

To customize a frequency governor for QCS9075 and QCS8275, see the corresponding addendum. The following guides are available to licensed users with authorized access:

- Qualcomm Linux Performance Guide - Addendum for QCS9075
- Qualcomm Linux Performance Guide - Addendum for QCS8275

Note: The commands specified in the following table should be run on the device.

Table : Commands to customize the CPU frequency governor

Command	Purpose

`echo performance > /sys/devices/system/cpu/cpufreq/policy0/scaling_governor`
`echo performance > /sys/devices/system/cpu/cpufreq/policy4/scaling_governor`
`echo performance > /sys/devices/system/cpu/cpufreq/policy7/scaling_governor`
	Sets the CPU governor to enhance the system performance.

`cat /sys/devices/system/cpu/cpufreq/policy0/scaling_governor`
`cat /sys/devices/system/cpu/cpufreq/policy4/scaling_governor`
`cat /sys/devices/system/cpu/cpufreq/policy7/scaling_governor`
	Verifies the CPU frequency governor.

Command	Purpose

`echo schedutil > /sys/devices/system/cpu/cpufreq/policy0/scaling_governor`
`echo schedutil > /sys/devices/system/cpu/cpufreq/policy4/scaling_governor`
`echo schedutil > /sys/devices/system/cpu/cpufreq/policy7/scaling_governor`
	Sets the CPU frequency governor to `schedutil`.

`echo 1000 > /sys/devices/system/cpu/cpufreq/policyX/schedutil/rate_limit_us`	Customizes `rate_limit_us`. The value of `X` in `policyX` corresponds to clusters 0, 4, and 7. This is a `schedutil` governor parameter. It contains the value in microseconds. The governor waits for `rate_limit_us` time to re-evaluate the load. The Qualcomm-tuned value is 1000.

6.3 Customize DVFS governor

You can customize the static map DVFS governor and bandwidth monitoring (BWMON) governor using DTSI files according to both, the power and the performance requirements of your device.

Customize static map DVFS governor

You can customize the mapping between the CPU frequency and the L3/DDR frequency according to both, the power and the performance requirements from the `arch/arm64/boot/dts/qcom/<target>.dtsi` file in the source code.

In the DTSI file, for each CPU node, there is an entry with `operating-points-v2 = <&cpux_opp_table>`, where `cpux_opp_table` holds the static mapping between the CPU frequency, and the L3 and the DDR frequencies.

For example, the following entry indicates that the CPU 0 frequency operates at 300 MHz:

```
cpu0_opp_300mhz: opp-300000000 {
                opp-hz = /bits/ 64 <300000000>;
                pp-peak-kBps = <800000 9600000>;
                };
```

When CPU 0 operates at 300 MHz, it performs the following actions:

- Votes L3 to 9600000, which is 9600000/w (where w = 32) = 300,000, which corresponds to 300 MHz
- Votes DDR to 800000, which is 800000/w (where w = 4) = 200,00, which corresponds to 200 MHz

In this example, w represents the number of bytes you can write in a single cycle.

- For L3, this value is 32, which means one transaction per cycle at 32 bytes per transaction.
- For dual-channel DDR, this value is 4. Each channel can perform two transactions per cycle (because it's the DDR memory), and each transaction is of 2 bytes.

For more information, see Generic OPP (Operating Performance Points) Bindings.

The tables specify the per channel DDR values. This implies that the mapping is between the CPU frequency and the bandwidth of each memory controller channel.

Caution: If you change the DTSI files, it impacts the power and the performance of the device. Ensure that you verify the impact across all the relevant use cases before changing any nodes.

Customize BWMON governor

You can customize `bwmon_opp_table` for last level cache controller (LLCC) and DDR voting according to both, the power and the performance requirements from the `arch/arm64/boot/dts/qcom/<target>.dtsi` file in the source code.

The following code samples contain the DDR frequency level and the LLCC frequency level, and each level translates to LLCC and DDR voting based on the traffic.

The following DTSI entry votes to DDR based on the CPU traffic between LLCC to DDR, which corresponds to 200 MHz. In this example, 800000/w (where w = 4) equals 200000:

```
llcc_bwmon_opp_table: opp-table {
        compatible = "operating-points-v2";
        opp-0 {
                 opp-peak-kBps = <800000>;
         };
```

The following DTSI entry votes to LLCC based on the CPU traffic between CPU and LLCC, which corresponds to 150 MHz. In this example, 2400000/w (where w = 16) equals 150000:

```
cpu_bwmon_opp_table: opp-table {
       compatible = "operating-points-v2";
```

```
        opp-0 {
                  opp-peak-kBps = <2400000>;
          };
```

In these examples, w represents the number of bytes that you can write in a single cycle.

- For DDR, this value is 4. Each channel can perform two transactions per cycle (DDR memory), and each transaction is of 2 bytes.
- For LLCC, this value is 16.

6.4 Customize perflock

You can use perflock APIs, run a use case using different tuning parameters, and debug the perflock issues.

Perflock APIs

To use the perflock APIs from the user space source code, do the following:

1. Include a dynamic linking header in the source file: `#include <dlfcn.h>`.
2. Load `library(libqti-perfd-client.so)` using `dlopen`.
3. Load the `perf_lock_acq()` and `perf_lock_rel()` symbols using `dlsym`.
4. Declare an integer variable to store the returns of `perf_lock_acq()`.
5. Define `perf_lock_opts` with the required resource opcode and value pairs.
6. Acquire a lock with the specified optimizations using `perf_lock_acq()`.
7. Release the lock using `perf_lock_rel()`.
8. Unload the library using `dlclose` on cleanup.

The following sample code shows the usage of perflock APIs and illustrates how to set the minimum frequency of the Silver core to 1.9 GHz:

```
#include <stdlib.h>
#include <stdio.h>
#include <dlfcn.h>
#include <error.h>
#include <unistd.h>

#define PATH_MAX 92

static void *qcopt_handle;
static int (*perf_lock_acq)(int handle, int duration, int list[], int
numArgs);
static int (*perf_lock_rel)(int handle);
static int perf_lock_handle;
char opt_lib_path[PATH_MAX] = "/usr/lib/libqti-perfd-client.so";

int main()
{

    if ((qcopt_handle = dlopen(opt_lib_path, RTLD_NOW)) == NULL)
    {
        printf("dlopen failed with error: NULL\n");
    } else {
        printf("call perflock \n");
        perf_lock_acq = (int (*)(int, int, int *, int))dlsym(qcopt_
handle, "perf_lock_acq");
        if(!perf_lock_acq)
            printf("Unable to get perf_lock_rel function handle.\n");
        perf_lock_rel = (int (*)(int))dlsym(qcopt_handle, "perf_lock_
rel");
    }
    int perf_lock_opts[2] = {0x44008100,1958400};
    perf_lock_handle = perf_lock_acq(perf_lock_handle, 0, perf_lock_
opts, 2);
    printf("perflock acquired %d\n", perf_lock_handle);

   //Code section requiring perflock

   perf_lock_rel(perf_lock_handle);
   printf("releasing perflock\n");
   dlclose(qcopt_handle);
   return 0;
}
```

Perflock native test

The perflock native test is a tool designed to evaluate the effectiveness of specific combinations of perflock opcodes.

This tool is available at `/usr/bin/` on the device.

You can run this utility as a use case to experiment with different tuning parameters for `<resource, opcode>`. When the required performance improvements are confirmed, the perflock code can be integrated into Qualcomm® Linux®.

The following code shows the syntax:

```
perflock_native_test --acq <handle> <duration> <resource opcode,
value, resource opcode, value,...>
```

The following are some examples of how to use the tool. Run these commands on the device:

- To acquire a perflock for the Silver cluster with minimum frequency set to 1958400 KHz for 30 seconds, run the following command:

  ```
  perflock_native_test --acq 0 30000 0x44008100,1958400
  ```

- To verify the minimum frequency of the Silver cluster, run the following command:

  ```
  cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq
  ```

- To verify the perflock logging from `syslog`, run the following command:

  ```
  cat /var/log/syslog
  ```

  The following are the logs:

  ```
  perflock_native_test: ANDR-PERFLOCK-TESTERANDR-PERFLOCK-TESTER:
  initialize() 60: NRP: lib name libqti-perfd-client.so
  perflock_native_test: ANDR-PERFLOCK-TESTERANDR-PERFLOCK-TESTER:
  main() 502: NRP: resource_list[0] = 44008100 //silver core min frequency opcode
  perflock_native_test: ANDR-PERFLOCK-TESTERANDR-PERFLOCK-TESTER:
  main() 502: NRP: resource_list[1] = 7a6 // silver core min freq in hex
  ```

Debug perflock

The following are some examples of logging mechanisms to debug the perflock issues:

Note: The following example commands should be run on the device.

- To verify if the PerfHAL service is available, run the following command:

```
ls /usr/bin | grep perf
```

- To verify the running status of the PerfHAL service, run the following command:

```
systemctl status perf-hal
```

- To enable perflock logs and traces in `/var/log/syslog`, run the following commands:

```
echo debug.trace.perf=1 >> /etc/build.prop
```

```
echo vendor.debug.trace.perf=1 >> /etc/build.prop
```

```
reboot
```

6.5 Customize memory

You can manage virtual memory using Kswapd, regulate system memory using watermark, and define memory regions for specific subsystems through memory carveouts.

Kswapd

Kswapd is a kernel thread that manages the virtual memory. When the system is low on free memory, Kswapd is activated to reclaim memory by identifying less frequently used pages from the file cache and moving them to the swap space (ZRAM).

Kswapd does the following:

- Monitors the memory usage of the system
- Performs swapping activities in the background to maintain an optimal level of free memory

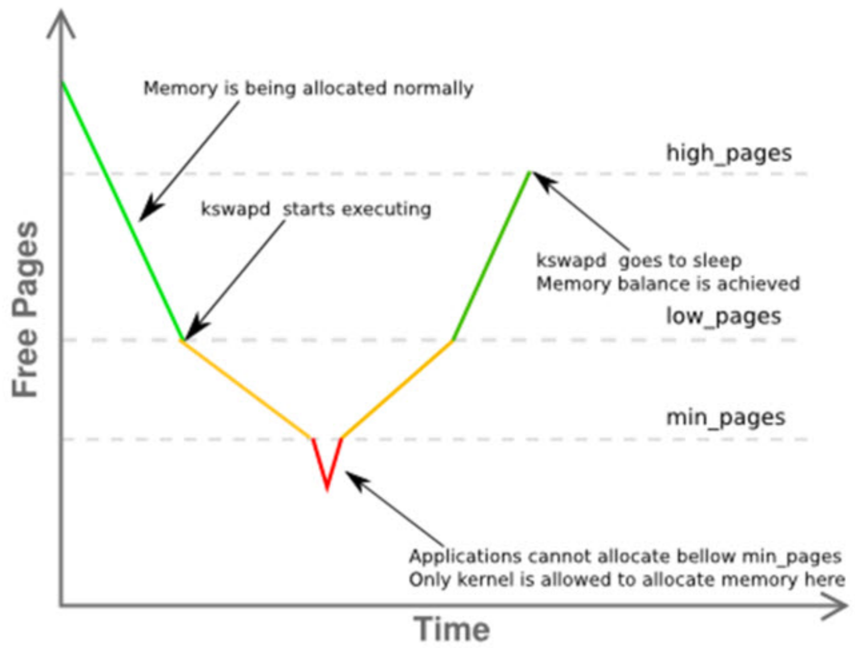


Figure : CPU usage vs. performance using Kswapd

Watermark

A watermark is a threshold or limit set on memory parameters that helps regulate the allocation and usage of system memory.

Table : Watermark parameters

Parameter	Description
`/proc/sys/vm/min_free_kbytes`	• This parameter determines the minimum amount of free memory that the system must maintain across all zones. • It ensures that a certain reserve of memory remains available for critical operations, such as handling atomic allocations, which can't wait for memory reclamation. • The number of reserved free pages in each zone is proportional to its size.
`/proc/sys/vm/watermark_scale_factor`	• This factor controls the aggressiveness of Kswapd. • It defines the amount of memory left in a system before Kswapd is woken up, and the amount of free memory required before Kswapd goes back to sleep. • The unit is in fractions of 10,000. • The default value of 10 means that the distance between the watermarks is 0.1% of the available memory in the node or system. The maximum value is 1000, or 10% of memory.

Parameter	Description
`/proc/sys/vm/watermark_boost_factor`	• This factor is used to optimize memory fragmentation by temporarily providing a high-watermark for the memory management area, which allows Kswapd to reclaim more memory. • The unit is in fractions of 10,000. • The default value of 15,000 means that up to 150% of the high watermark is reclaimed in the event of a pageblock being mixed due to fragmentation. • The number of fragmentation events that have occurred in the recent past determine the level of reclaim. A boost factor of 0 disables the feature.

For more information, see https://www.kernel.org/doc/Documentation/admin-guide/sysctl/vm.rst.

Memory carveout

Memory carveouts refer to specific memory regions that the kernel can't address, known as no-map regions. Specific subsystems such as modem, camera, aDSP, cDSP, and Qualcomm® Trusted Execution Environment (TEE)/TrustZone (TZ) use these regions exclusively and hence those are inaccessible to the Linux kernel.

These carveouts are defined in the source code at `arch/arm64/boot/dts/qcom/<target>.dtsi`.

They fall under the `reserved-memory` node and can be configured.

The following code shows the no-map regions for the cDSP, camera, modem, and QTEE/TrustZone (TZ) subsystems:

```
reserved-memory {
        cdsp_secure_heap_mem: cdsp-secure-heap@81800000 {
            reg = <0x0 0x81800000 0x0 0x1e00000>;
            no-map;
        };

        camera_mem: camera@84300000 {
            reg = <0x0 0x84300000 0x0 0x500000>;
            no-map;
        };
        adsp_mem: adsp@86100000 {
            reg = <0x0 0x86100000 0x0 0x2800000>;
            no-map;
        };
        cdsp_mem: cdsp@88900000 {
            reg = <0x0 0x88900000 0x0 0x1e00000>;
            no-map;
        };
        mpss_mem: mpss@8b800000 {
            reg = <0x0 0x8b800000 0x0 0xf600000>;
            no-map;
        };

        tz_stat_mem: tz-stat@c0000000 {
            reg = <0x0 0xc0000000 0x0 0x100000>;
            no-map;
        };
        tags_mem: tags@c0100000 {
            reg = <0x0 0xc0100000 0x0 0x1200000>;
            no-map;
        };
```

You can adjust the reserved memory carveout regions based on your product requirements. This adjustment can include removing a modem carveout region or reducing the size of QTEE/TrustZone (TZ) application memory carveout regions.

For example, the following code snippet shows how to disable the modem memory carveout for memory optimization:

```
reserved-memory {
        mpss_mem: mpss@8b800000 {
            reg = <0x0 0x8b800000 0x0 0xf600000>;
            no-map;
                    status = "disabled";
        };
```

In this code:

- `0x8b800000` is the base address of the modem region.
- `0xf600000` is the size of the modem region, which is 246 MB.
- Add `status = disabled` to make this region available to the Linux kernel.

The following example snippets show how to reduce the memory carveout size of the QTEE/TrustZone (TZ) applications from 28 MB (`0x1c00000`) to 20 MB (`0x1400000`):

Before:	**After:**

```
reserved memory {
                 trusted_
apps_mem: trusted_
apps@c1800000 {
                 reg = <0x0
0xc1800000 0x0 0x1c00000>;
                 no-map;
   };
```
	```
	         reserved memory {
	              trusted_apps_
	mem: trusted_apps@c1800000 {
	              reg = <0x0
	0xc1800000 0x0 0x1400000>;
	              no-map;
	};
	```

7 Troubleshoot performance issues

To address the performance issues, you can use both basic and advanced troubleshooting methods.

7.1 Basic troubleshooting

Basic troubleshooting involves fundamental techniques at the application level. It's useful when developing applications using the Qualcomm development kits for educational and academic purposes. Basic troubleshooting can be applied to devices with Qualcomm® Linux® that operate without requiring root access.

For more complex issues, see Advanced troubleshooting.

Analyze user space and kernel traces

Tools such as Function tracer (ftrace), Trace Compass, and LTTng are commonly used to analyze traces on Linux for performance issues.

Performance debug tool	Reference
Trace Compass	https://archive.eclipse.org/tracecompass/doc/stable/org.eclipse.tracecompass.doc.user/User-Guide.html
LLTng	https://lttng.org/docs/v2.13/

You can compile your application with `-llttng-ust` and `-g -finstrument-functions` to display the function call stack.

For example, run the following command for compilation:

```
aarch64-qcom-linux-g++ <cpp source file> -o <output file> -llttng-ust
-g -finstrument-functions
```

Qualcomm Linux adds the following GCC and G++ compilers on the device:

- `aarch64-qcom-linux-gcc`
- `aarch64-qcom-linux-g++`

For more information about compiling LTTng, GCC, and G++, see Compile performance tools.

Capture LTTng-UST trace

To capture a trace using LTTng, follow these steps:

1. To display a call stack of the application with `liblttng-ust-cyg-profile.so` create a session named my-session with the following command:

   ```
   lttng create my-session --output=/tmp/my-trace
   ```

   The traces are available at `/tmp/my-trace`.

2. Run the commands in the following sequence to capture the traces:

   ```
   lttng enable-event -u -a
   ```

   ```
   lttng enable-event -k -a
   ```

   ```
   lttng start
   ```

3. Preload the `liblttng-ust-cyg-profile` library when running your program:

   ```
   LD_PRELOAD=/usr/lib/liblttng-ust-cyg-profile.so ./test_
   executable
   ```

   ```
   lttng stop
   ```

   ```
   lttng destroy my-session
   ```

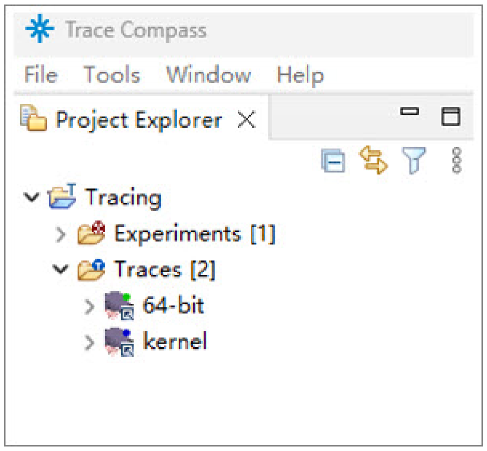
Load LTTng traces

1. To load and visualize the LTTng traces in Trace Compass, use secure copy protocol (SCP) or a similar tool to transfer a trace from the target to the host. Ensure to specify the target IP address in the command. Here is an example command:

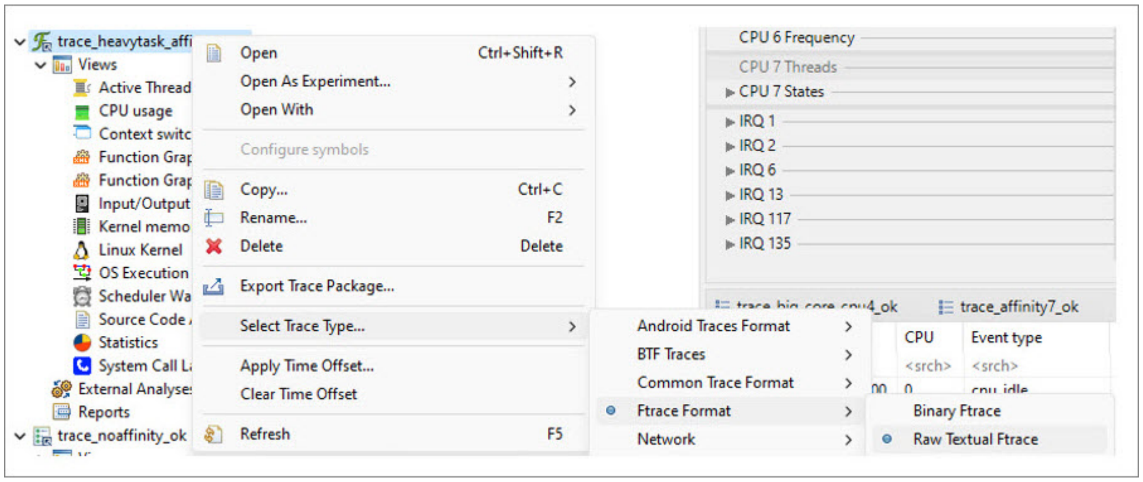
```
scp -r root@10.92.162.185:/home/root/lttng-traces/ <store trace
path>
```

2. Load the LTTng kernel and UST traces with Trace Compass on the host machine. From the Trace Compass tool, use the **File** menu option to open a trace.

Note: The screenshots are provided for reference. The directory structure shown in the screenshots may vary depending on the Trace Compass tool version.



3. To select a trace type, right-click on the trace, and choose **Select Trace Type** > **Ftrace Format** > **Raw Textual Ftrace** as shown in the following figure:



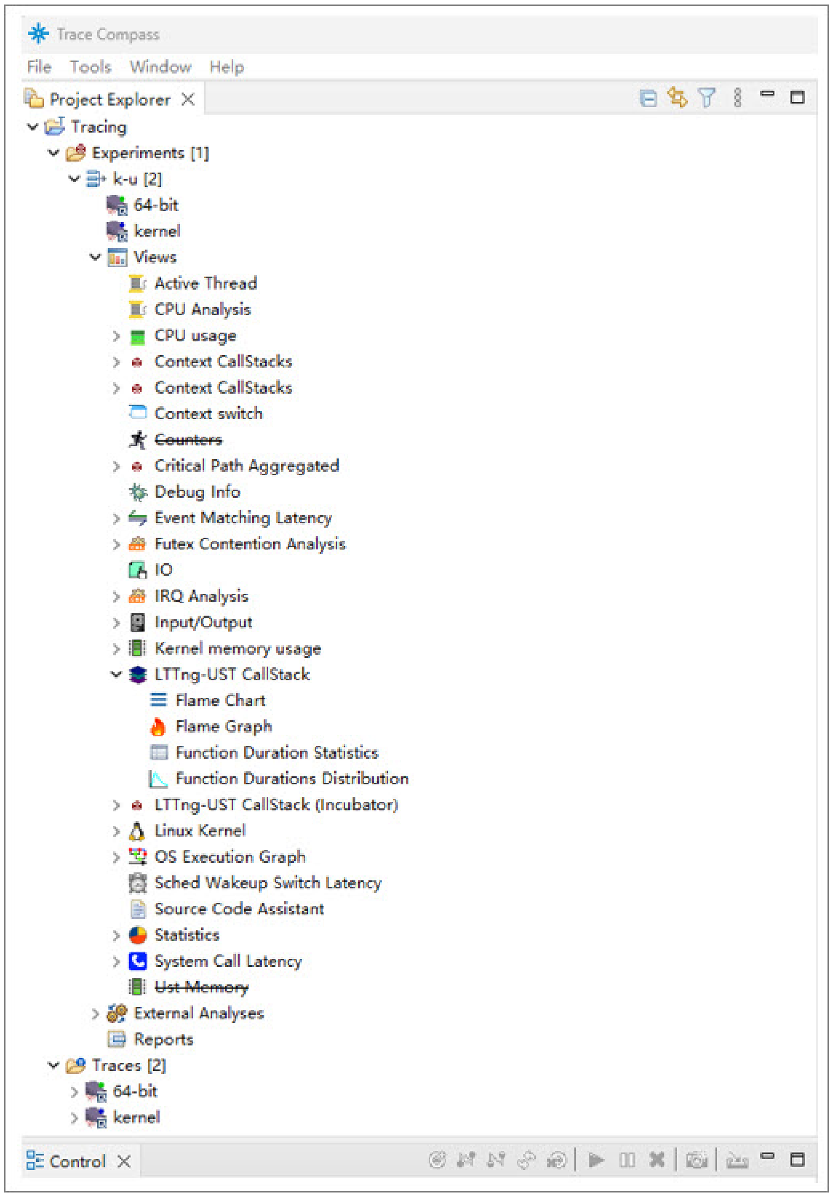
4. Install the required add-ons in Trace Compass for ftrace analysis.

   Navigate to **Menu** > **Tools** > **Add-ons** and select **Trace Compass ftrace**.

   Note: It's recommended to update Trace Compass preferences.

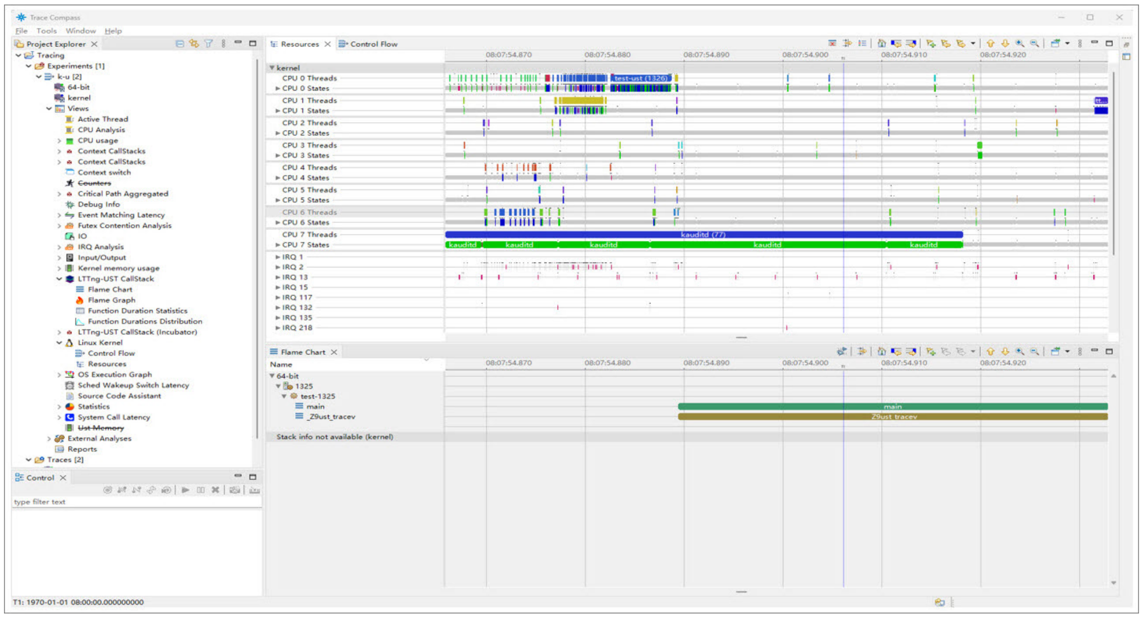
   To print the time that matches the raw ftrace, change **Tracing–Time Format** to **TTT** (seconds in epoch).

5. To display the kernel and UST traces in one view, create Experiments and add two traces.

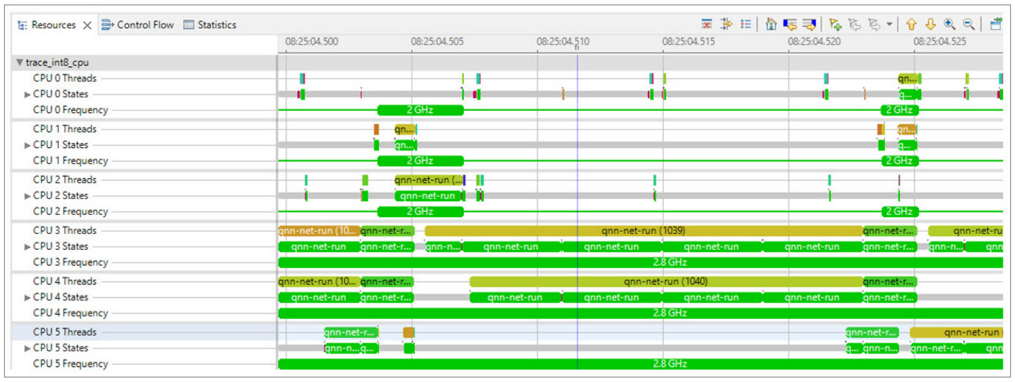


6. Select **Views** > **LTTng-UST-CallStack** > **Flame Chart and Views** > **Linux Kernel** > **Resources**.

   Trace Compass can display kernel resources and user space application function call stack as shown in the following figure:



7. Follow step 6 to open a trace for the CPU frequency. Select the **Resources** panel and the **Timeline** view of the process running on a specified CPU. There is a frequency number in the CPU frequency line. The following figure shows CPU0 to CPU2 running at 2 GHz and CPU3 to CPU5 running at 2.8 GHz.



Monitor CPU consumption of user space application

Several Linux utilities, such as top and htop can be used to monitor the CPU usage.

Top

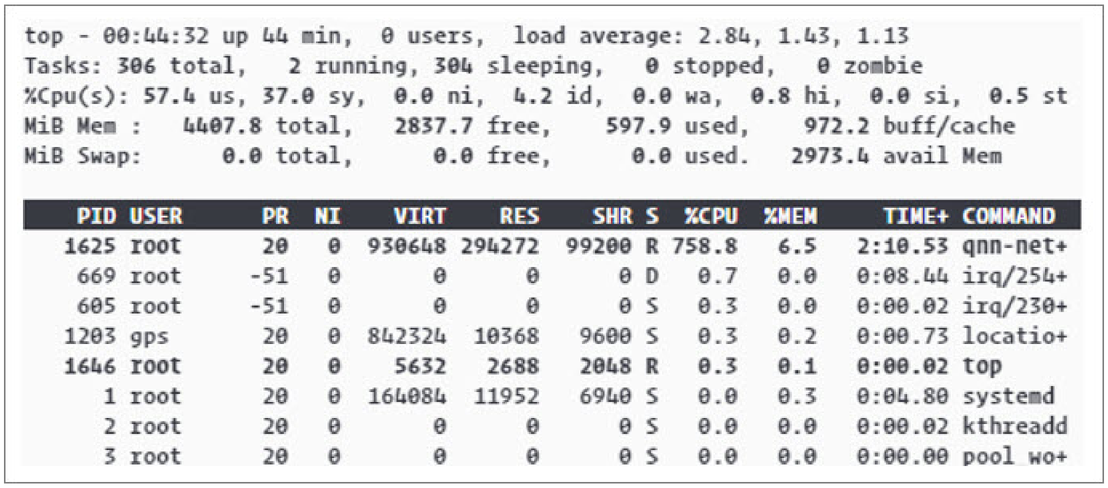
Top is a tool that checks the CPU usage for an application and displays the overall CPU usage. On an octa-core platform, tasks can consume the CPU from 0% to 800%.

To set a terminal environment to run top, run the following commands on the device:

```
export TERM=xterm
```

```
top
```

The following figure shows the CPU usage as an output of the command:



```
top - 00:44:32 up 44 min,  0 users,  load average: 2.84, 1.43, 1.13
Tasks: 306 total,   2 running, 304 sleeping,   0 stopped,   0 zombie
%Cpu(s): 57.4 us, 37.0 sy,  0.0 ni,  4.2 id,  0.0 wa,  0.8 hi,  0.0 si,  0.5 st
MiB Mem :   4407.8 total,   2837.7 free,    597.9 used,    972.2 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   2973.4 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 1625 root      20   0  930648 294272  99200 R 758.8   6.5   2:10.53 qnn-net+
  669 root     -51   0       0      0      0 D   0.7   0.0   0:08.44 irq/254+
  605 root     -51   0       0      0      0 S   0.3   0.0   0:00.02 irq/230+
 1203 gps       20   0  842324  10368   9600 S   0.3   0.2   0:00.73 locatio+
 1646 root      20   0    5632   2688   2048 R   0.3   0.1   0:00.02 top
    1 root      20   0  164084  11952   6940 S   0.0   0.3   0:04.80 systemd
    2 root      20   0       0      0      0 S   0.0   0.0   0:00.02 kthreadd
    3 root      20   0       0      0      0 S   0.0   0.0   0:00.00 pool_wo+
```

htop

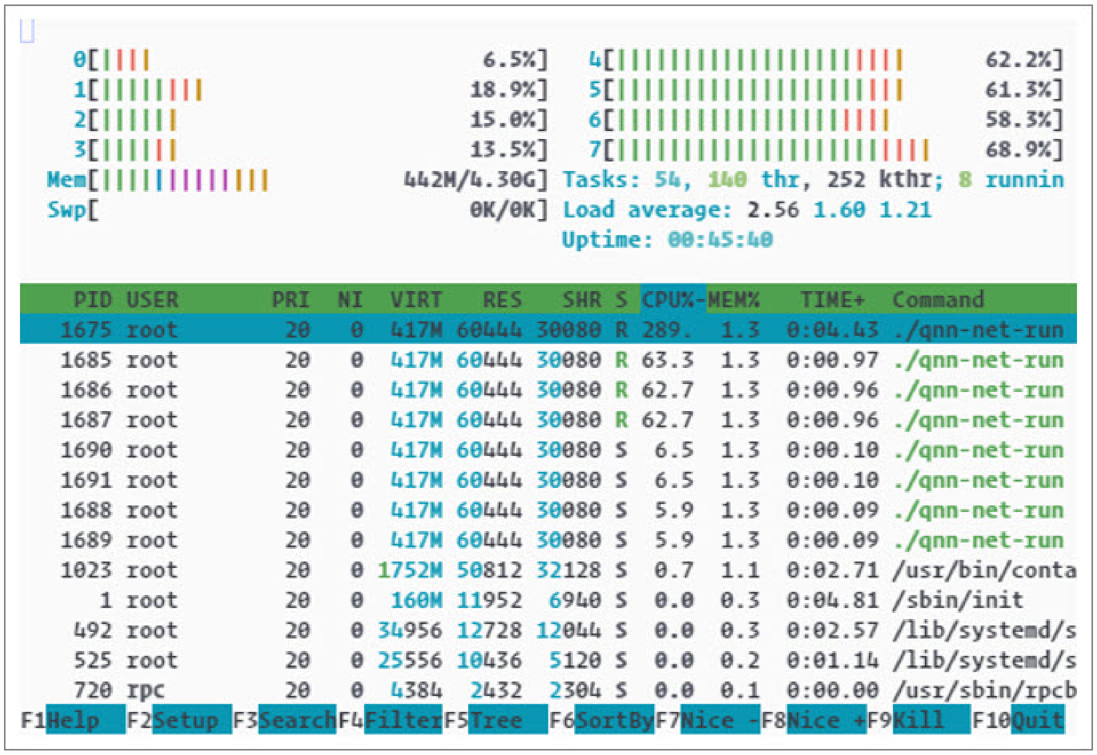
htop displays the per-core CPU usage and overall CPU usage for each process. To compile htop on a build, see Compile performance tools.

To set a terminal environment for htop, run the following commands on the device:

```
export TERM=xterm
```

```
htop
```

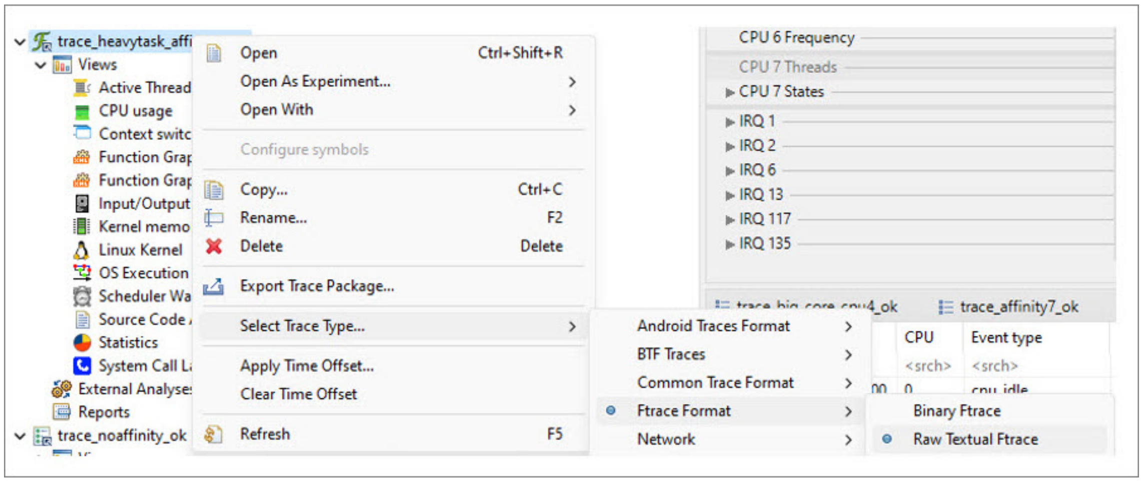
The following figure shows the per core CPU usage as an output of the command:



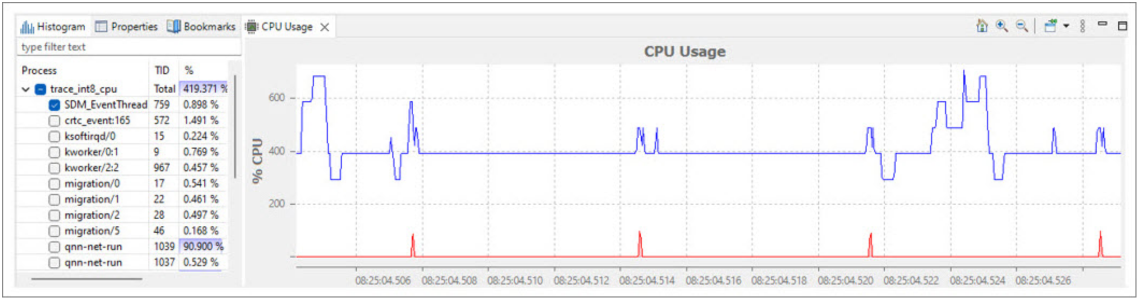
```
    0[||||                         6.5%]   4[||||||||||||||||||||||||      62.2%]
    1[||||||||                    18.9%]   5[||||||||||||||||||||||||      61.3%]
    2[||||||                      15.0%]   6[||||||||||||||||||||||        58.3%]
    3[||||||                      13.5%]   7[||||||||||||||||||||||||||    68.9%]
  Mem[|||||||||||||          442M/4.30G] Tasks: 54, 140 thr, 252 kthr; 8 runnin
  Swp[                            0K/0K] Load average: 2.56 1.60 1.21
                                         Uptime: 00:45:40

    PID USER       PRI  NI  VIRT   RES   SHR S CPU%-MEM%   TIME+  Command
   1675 root        20   0  417M 60444 30080 R 289.  1.3  0:04.43 ./qnn-net-run
   1685 root        20   0  417M 60444 30080 R 63.3  1.3  0:00.97 ./qnn-net-run
   1686 root        20   0  417M 60444 30080 R 62.7  1.3  0:00.96 ./qnn-net-run
   1687 root        20   0  417M 60444 30080 R 62.7  1.3  0:00.96 ./qnn-net-run
   1690 root        20   0  417M 60444 30080 S  6.5  1.3  0:00.10 ./qnn-net-run
   1691 root        20   0  417M 60444 30080 S  6.5  1.3  0:00.10 ./qnn-net-run
   1688 root        20   0  417M 60444 30080 S  5.9  1.3  0:00.09 ./qnn-net-run
   1689 root        20   0  417M 60444 30080 S  5.9  1.3  0:00.09 ./qnn-net-run
   1023 root        20   0 1752M 50812 32128 S  0.7  1.1  0:02.71 /usr/bin/conta
      1 root        20   0  160M 11952  6940 S  0.0  0.3  0:04.81 /sbin/init
    492 root        20   0 34956 12728 12044 S  0.0  0.3  0:02.57 /lib/systemd/s
    525 root        20   0 25556 10436  5120 S  0.0  0.2  0:01.14 /lib/systemd/s
    720 rpc         20   0  4384  2432  2304 S  0.0  0.1  0:00.00 /usr/sbin/rpcb
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

CPU usage in Trace Compass

1. Open the Trace Compass tool on the host machine and load a trace.
2. Right-click on the trace, and choose **Select Trace Type** > **Ftrace Format** > **Raw Textual Ftrace** as shown in the following figure:



3. Right-click on **Raw Textual Ftrace** and select **Open**.
4. Double-click on **CPU usage** to view the system-wide CPU usage. Select a task in the left panel to check the CPU usage per task as shown in the following figure:



Process	TID	%
trace_int8_cpu	Total	419.371 %
SDM_EventThread	759	0.898 %
crtc_event:165	572	1.491 %
ksoftirqd/0	15	0.224 %
kworker/0:1	9	0.769 %
kworker/2:2	967	0.457 %
migration/0	17	0.541 %
migration/1	22	0.461 %
migration/2	28	0.497 %
migration/5	46	0.168 %
qnn-net-run	1039	90.900 %
qnn-net-run	1037	0.529 %

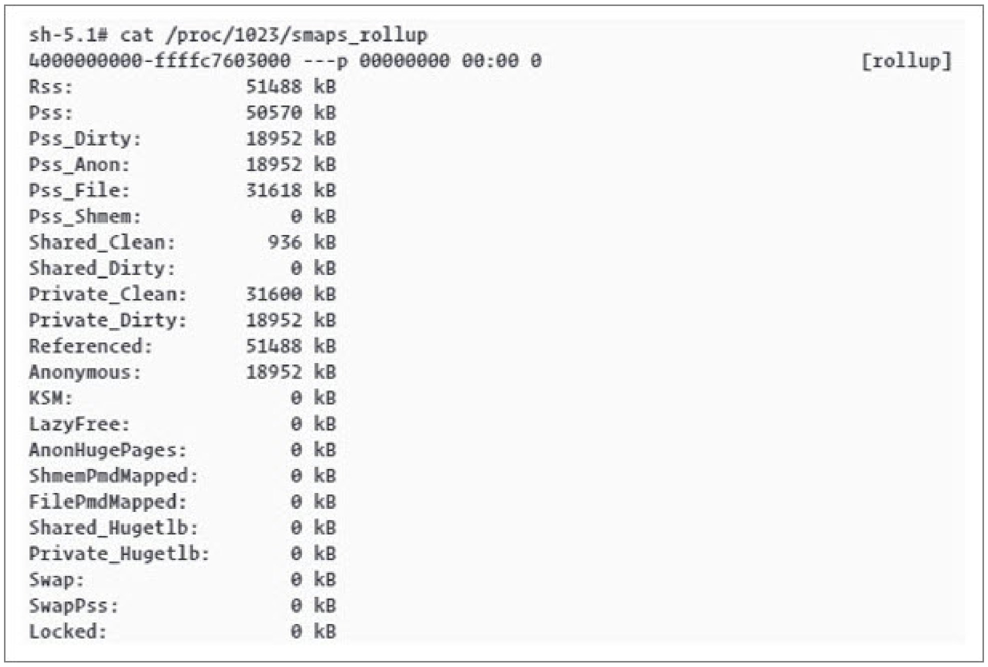
Monitor the memory consumption of user space application

You can check the memory allocation and memory usage for various processes.

To check memory consumption of a process, run the following command on a device:

```
cat /proc/<pid>/smaps_rollup
```

The following figure shows an output of the command:



```
sh-5.1# cat /proc/1023/smaps_rollup
4000000000-ffffc7603000 ---p 00000000 00:00 0                          [rollup]
Rss:               51488 kB
Pss:               50570 kB
Pss_Dirty:         18952 kB
Pss_Anon:          18952 kB
Pss_File:          31618 kB
Pss_Shmem:             0 kB
Shared_Clean:        936 kB
Shared_Dirty:          0 kB
Private_Clean:     31600 kB
Private_Dirty:     18952 kB
Referenced:        51488 kB
Anonymous:         18952 kB
KSM:                   0 kB
LazyFree:              0 kB
AnonHugePages:         0 kB
ShmemPmdMapped:        0 kB
FilePmdMapped:         0 kB
Shared_Hugetlb:        0 kB
Private_Hugetlb:       0 kB
Swap:                  0 kB
SwapPss:               0 kB
Locked:                0 kB
```

Procrank

Procrank is a tool that displays the memory consumption for each process. By default, it shows the following set sizes:

- VSS: Virtual set size
- RSS: Resident set size
- PSS: Proportional set size
- USS: Unique set size

PSS is considered as actual memory consumption by a process.

Build Procrank from source code

Run the following commands on the host machine:

```
sudo apt install -y gcc-aarch64-linux-gnu
```

```
git clone https://github.com/cglmcu/procrank.git
```

```
cd procrank
```

```
export CC=aarch64-linux-gnu-gcc
```

```
aarch64-linux-gnu-gcc *.c -Os -o procrank -I.
```

Use Android Debug Bridge (adb) or a similar tool to transfer the Procrank file into the device from the host. Here are the example commands:

```
adb shell mount -o remount, rw /usr
```

```
adb push procrank /usr/bin
```

```
adb shell chmod a+x /usr/bin/procrank
```

Note: Ensure to specify the target IP address in the command.

Procrank command examples:

- To view the anonymous memory allocated by each process, run the following command on the device:

  ```
  procrank -C
  ```

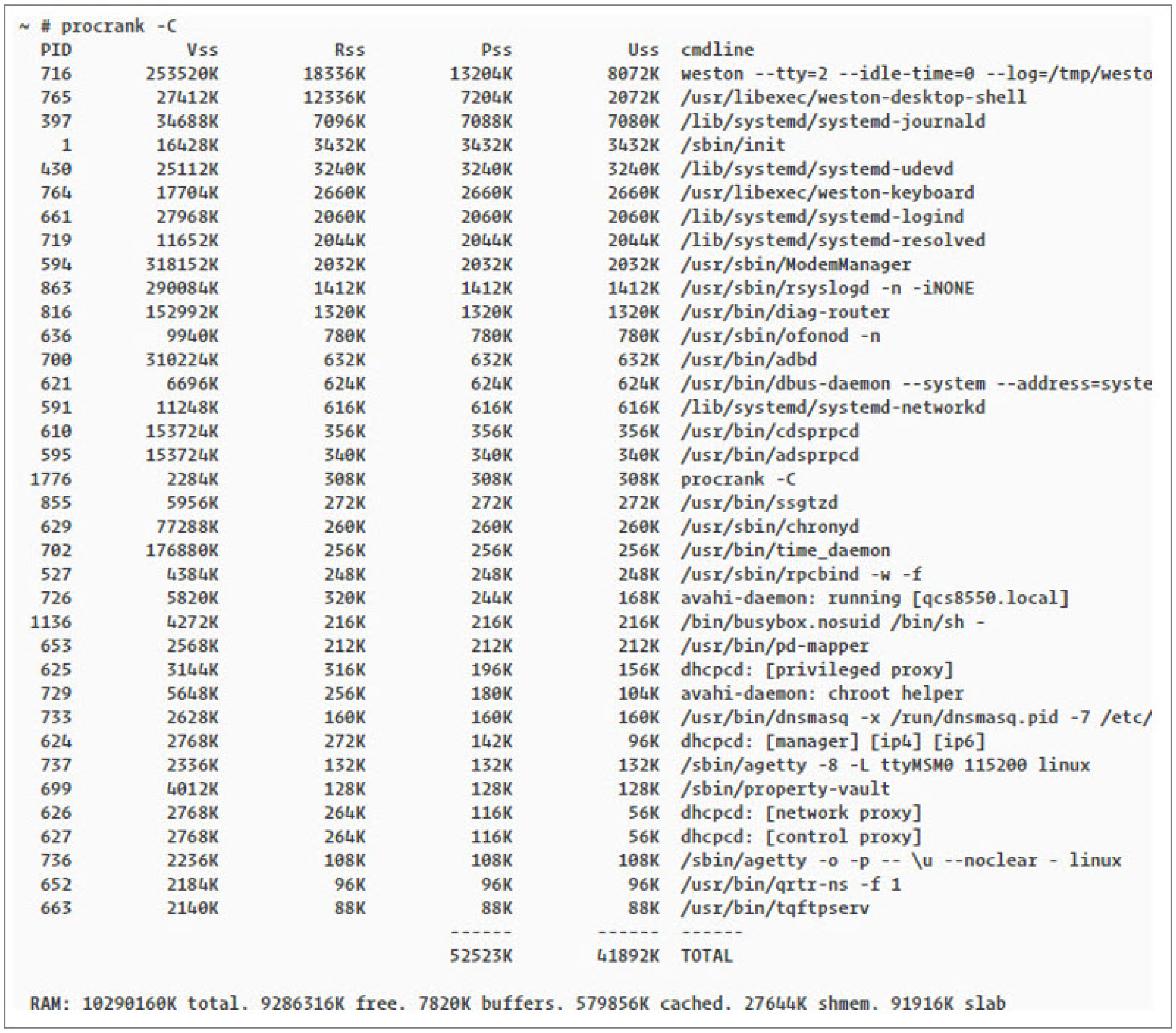
- To show the file cache memory allocated by each process, run the following command on the device:

  ```
  procrank -c
  ```

- To view both the anonymous and file cache memories allocated by each process, run the following command on the device:

  ```
  procrank
  ```

The following figure shows an example output of the `procrank -C` command:



```
~ # procrank -C
  PID          Vss           Rss           Pss           Uss  cmdline
  716      253520K        18336K        13204K         8072K  weston --tty=2 --idle-time=0 --log=/tmp/westo
  765       27412K        12336K         7204K         2072K  /usr/libexec/weston-desktop-shell
  397       34688K         7096K         7088K         7080K  /lib/systemd/systemd-journald
    1       16428K         3432K         3432K         3432K  /sbin/init
  430       25112K         3240K         3240K         3240K  /lib/systemd/systemd-udevd
  764       17704K         2660K         2660K         2660K  /usr/libexec/weston-keyboard
  661       27968K         2060K         2060K         2060K  /lib/systemd/systemd-logind
  719       11652K         2044K         2044K         2044K  /lib/systemd/systemd-resolved
  594      318152K         2032K         2032K         2032K  /usr/sbin/ModemManager
  863      290084K         1412K         1412K         1412K  /usr/sbin/rsyslogd -n -iNONE
  816      152992K         1320K         1320K         1320K  /usr/bin/diag-router
  636        9940K          780K          780K          780K  /usr/sbin/ofonod -n
  700      310224K          632K          632K          632K  /usr/bin/adbd
  621        6696K          624K          624K          624K  /usr/bin/dbus-daemon --system --address=syste
  591       11248K          616K          616K          616K  /lib/systemd/systemd-networkd
  610      153724K          356K          356K          356K  /usr/bin/cdsprpcd
  595      153724K          340K          340K          340K  /usr/bin/adsprpcd
 1776        2284K          308K          308K          308K  procrank -C
  855        5956K          272K          272K          272K  /usr/bin/ssgtzd
  629       77288K          260K          260K          260K  /usr/sbin/chronyd
  702      176880K          256K          256K          256K  /usr/bin/time_daemon
  527        4384K          248K          248K          248K  /usr/sbin/rpcbind -w -f
  726        5820K          320K          244K          168K  avahi-daemon: running [qcs8550.local]
 1136        4272K          216K          216K          216K  /bin/busybox.nosuid /bin/sh -
  653        2568K          212K          212K          212K  /usr/bin/pd-mapper
  625        3144K          316K          196K          156K  dhcpcd: [privileged proxy]
  729        5648K          256K          180K          104K  avahi-daemon: chroot helper
  733        2628K          160K          160K          160K  /usr/bin/dnsmasq -x /run/dnsmasq.pid -7 /etc/
  624        2768K          272K          142K           96K  dhcpcd: [manager] [ip4] [ip6]
  737        2336K          132K          132K          132K  /sbin/agetty -8 -L ttyMSM0 115200 linux
  699        4012K          128K          128K          128K  /sbin/property-vault
  626        2768K          264K          116K           56K  dhcpcd: [network proxy]
  627        2768K          264K          116K           56K  dhcpcd: [control proxy]
  736        2236K          108K          108K          108K  /sbin/agetty -o -p -- \u --noclear - linux
  652        2184K           96K           96K           96K  /usr/bin/qrtr-ns -f 1
  663        2140K           88K           88K           88K  /usr/bin/tqftpserv
                                       ------        ------  ------
                                       52523K        41892K  TOTAL

 RAM: 10290160K total. 9286316K free. 7820K buffers. 579856K cached. 27644K shmem. 91916K slab
```

Check instruction per cycle of the application

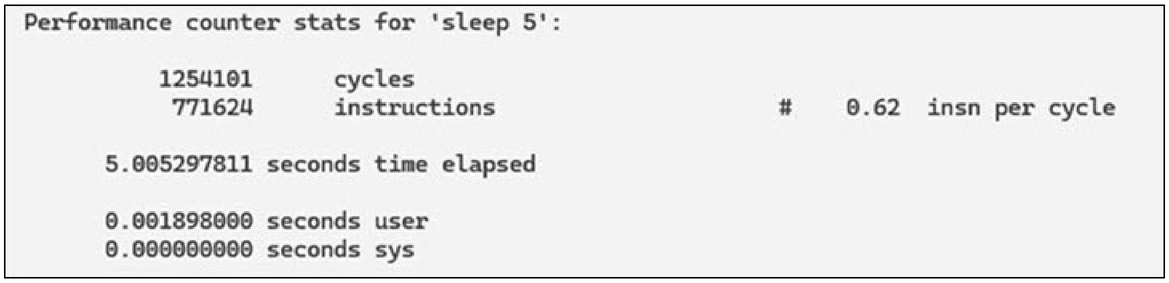
The perf utility calculates instruction per cycle (IPC) for an application using the hardware performance counters.

To compile the perf utility, see Compile performance tools.

To calculate IPC, run the following command on the device:

```
perf stat -e cycles,instructions sleep 5
```

The following figure shows an example output of the command:



```
Performance counter stats for 'sleep 5':

          1254101      cycles
           771624      instructions                     #    0.62  insn per cycle

      5.005297811 seconds time elapsed

      0.001898000 seconds user
      0.000000000 seconds sys
```

- If the IPC is less than 1.0, it's likely that the memory is stalled. In this case, Qualcomm Linux tuning strategies, such as reducing the memory I/O workload, can help improve performance.
- If the IPC is greater than 1.0, it's likely that it's instruction bound. In this case, reducing code execution by eliminating unnecessary work and cache operations can help improve performance.

Check parts of code consuming most CPU

The perf utility tool can generate a flame graph that helps visualize the stack and CPU usage of a thread with all the functions running on the CPU.

To generate a flame graph, do the following:

- On the device:
  1. Collect logs to generate a flame graph. To collect logs using the perf utility tool, run the following commands:

     ```
     perf record -g -o /tmp/perf.data -p <process pid> sleep 5
     ```

     ```
     cd /tmp
     ```

```
perf script > /tmp/perf.script
```

2. Run the following command using SCP or a similar tool and transfer `perf.script` from the target to the host. Ensure that you specify the target IP address in the command. Here is an example command:

```
scp -r root@10.92.162.185:/tmp/perf.script /local/mnt/
workspace/logs
```

- On the host:

1. Run the following command to download the flame graph:

```
git clone https://github.com/brendangregg/
FlameGraph.git
```

Ensure that you install Perl on the host machine.

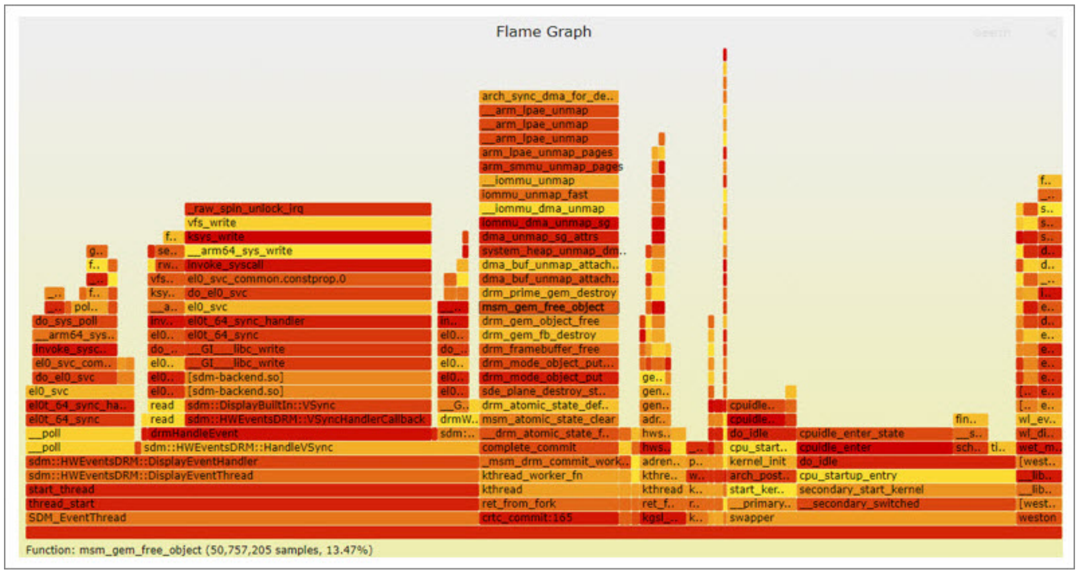
2. Copy `perf.script` in the `FlameGraph` directory:

```
cd FlameGraph
```

```
perl stackcollapse-perf.pl perf.script > out.folded
```

```
perl out.folded > perf.svg
```

3. Open the SVG file in a browser to view the flame graph to know the CPU usage:



Function: msm_gem_free_object (50,757,205 samples, 13.47%)

Check memory consumed by functions in the user space application code

Valgrind, an open-source tool, provides a utility called massif that helps to analyze the memory consumed by each function in a program.

The following is a sample code for memory allocation:

```
    #include <stdlib.h>

void g(void) {
   malloc(4000);
}

void f(void) {
   malloc(2000);
   g();
}

int main(void) {
   int i;
   int* a[10];
   for (i = 0; i < 10; i++) {
      a[i] = malloc(1000);
   }
   f();
   g();
   for (i = 0; i < 10; i++) {
      free(a[i]);
   }
   return 0;
}
```

Compile the source code and run the following Valgrind command on the device:

```
valgrind --tool=massif ./test
```

The following is an output of the sample code:

```

cat massif.out.1587
...
n3: 20000 (heap allocation functions) malloc/new/new[], --alloc-fns,
etc.
n0: 10000 0x10882B: main (in /home/root/valgrind/test)
```

```
n2: 8000 0x1087E7: g (in /home/root/valgrind/test)
n1: 4000 0x108807: f (in /home/root/valgrind/test)
n0: 4000 0x10885B: main (in /home/root/valgrind/test)
n0: 4000 0x10885F: main (in /home/root/valgrind/test)
n1: 2000 0x108803: f (in /home/root/valgrind/test)
n0: 2000 0x10885B: main (in /home/root/valgrind/test)
```

For more information about Valgrind, see https://valgrind.org/docs/manual/ms-manual.html.

Detect memory leaks in the user space application

To detect memory leaks within a process, you can use the Valgrind tool with the leak-check feature enabled.

The following is a sample code where memory has been allocated but not released:

```
    #include <stdlib.h>

void do_alloc() {
    int *x = malloc(10 * sizeof(int)); /* here simulate a leak */
    x[10] = 0; /* here write to invalid memory address */
}

int main() {
    do_alloc();
    return 0;
}
```

To detect memory leaks, compile the sample code and run the following command on the device:

```
valgrind --leak-check=yes ./test
```

The following is an output of the sample code:

```
==1512== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et
al.
==1512== Using Valgrind-3.18.1 and LibVEX; rerun with -h for
copyright info
==1512== Command: ./test
==1512==
==1512== Invalid write of size 4
==1512==    at 0x1087B4: do_alloc (in /home/root/valgrind/test)
==1512==    by 0x1087CF: main (in /home/root/valgrind/test)
==1512==  Address 0x4a36068 is 0 bytes after a block of size 40 alloc
```

```
'd
==1512==    at 0x486551C: malloc (vg_replace_malloc.c:381)
==1512==    by 0x1087A7: do_alloc (in /home/root/valgrind/test)
==1512==    by 0x1087CF: main (in /home/root/valgrind/test)
==1512==
==1512==
==1512== HEAP SUMMARY:
==1512==     in use at exit: 40 bytes in 1 blocks
==1512==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==1512==
==1512== 40 bytes in 1 blocks are definitely lost in loss record 1 of
1
==1512==    at 0x486551C: malloc (vg_replace_malloc.c:381)
==1512==    by 0x1087A7: do_alloc (in /home/root/valgrind/test)
==1512==    by 0x1087CF: main (in /home/root/valgrind/test)
```

7.2 Advanced troubleshooting

Advanced troubleshooting methods are used at the system level. These methods are crucial for building a Qualcomm reference device and integrating Qualcomm Linux across all layers to produce a final product.

For related information, see Basic troubleshooting.

Boot time

The phases of boot time and boot time log markers help in debugging and optimizing the boot process.

The Qualcomm Linux boot chain can be divided into two phases:

- Boot loader initialization and kernel loading: The boot loader is initiated and the kernel is loaded.
- Linux system initialization: The kernel, drivers, and user space services are initialized.

First-phase timelines (Boot loader initialization and kernel loading)

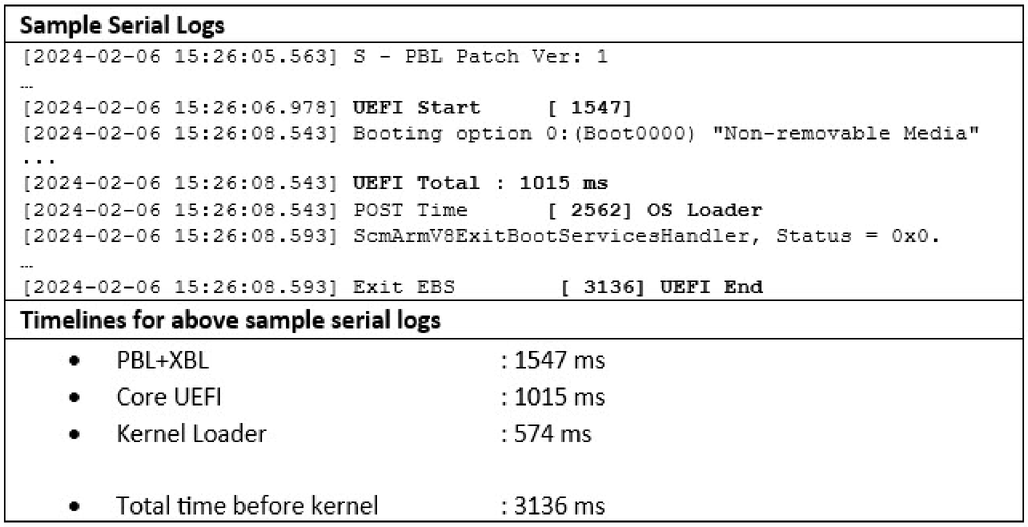
During the device booting sequence, collect the serial logs. Parsing these logs can provide a better understanding of the milestones in this phase.

The time taken across the modules can be measured using the respective timestamps listed in the following table:

Module	**Debug lines printed**
PBL + XBL	“UEFI Start” timestamp
Core UEFI	“UEFI Total” – time consumed is printed in milliseconds
Kernel loading	Difference between “UEFI End” - OS Loader” timestamps

For more information about how to collect serial logs, see Boot time measurement.

The following is an example of the sample serial logs and timelines:



Sample Serial Logs

```
[2024-02-06 15:26:05.563] S - PBL Patch Ver: 1
...
[2024-02-06 15:26:06.978] UEFI Start     [ 1547]
[2024-02-06 15:26:08.543] Booting option 0:(Boot0000) "Non-removable Media"
...
[2024-02-06 15:26:08.543] UEFI Total : 1015 ms
[2024-02-06 15:26:08.543] POST Time      [ 2562] OS Loader
[2024-02-06 15:26:08.593] ScmArmV8ExitBootServicesHandler, Status = 0x0.
...
[2024-02-06 15:26:08.593] Exit EBS        [ 3136] UEFI End
```

Timelines for above sample serial logs

• PBL+XBL	: 1547 ms
• Core UEFI	: 1015 ms
• Kernel Loader	: 574 ms
• Total time before kernel	: 3136 ms

Second phase timelines (Linux system initialization)

To capture performance statistics during system boot, use the systemd-analyze tool.

To install the tool, see Analysis tools.

To analyze the initialization of drivers within the kernel, enable the `initcall_debug` flag in the kernel boot command line. Use the systemd-analyze tool to analyze the initialization details of user space services and applications.

The following are the example commands that you can run on the device for using the systemd-analyze tool:

- To obtain the kernel and user space boot time, run the following command:

```
systemd-analyze time
```

The following is an output of the command:

```
Linux QCS6490 (Linux 6.6.0 #1 SMP PREEMPT Sun Feb 4 18:35:47 UTC
2024) arm64. Startup finished in 4.238s (kernel) + 15.620s
(userspace) = 19.859s multi-user.target reached after 15.594s in
userspace
```

- To obtain the time consumed by each subsystem during boot, run the following command:

```
systemd-analyze blame
```

The following is an output of the command:

```
4.982s android-tools-adbd.service
3.013s dev-disk-byx2dpartlabel-system.device
1.418s systemd-modules-load.service
1.179s sshdgenkeys.service
```

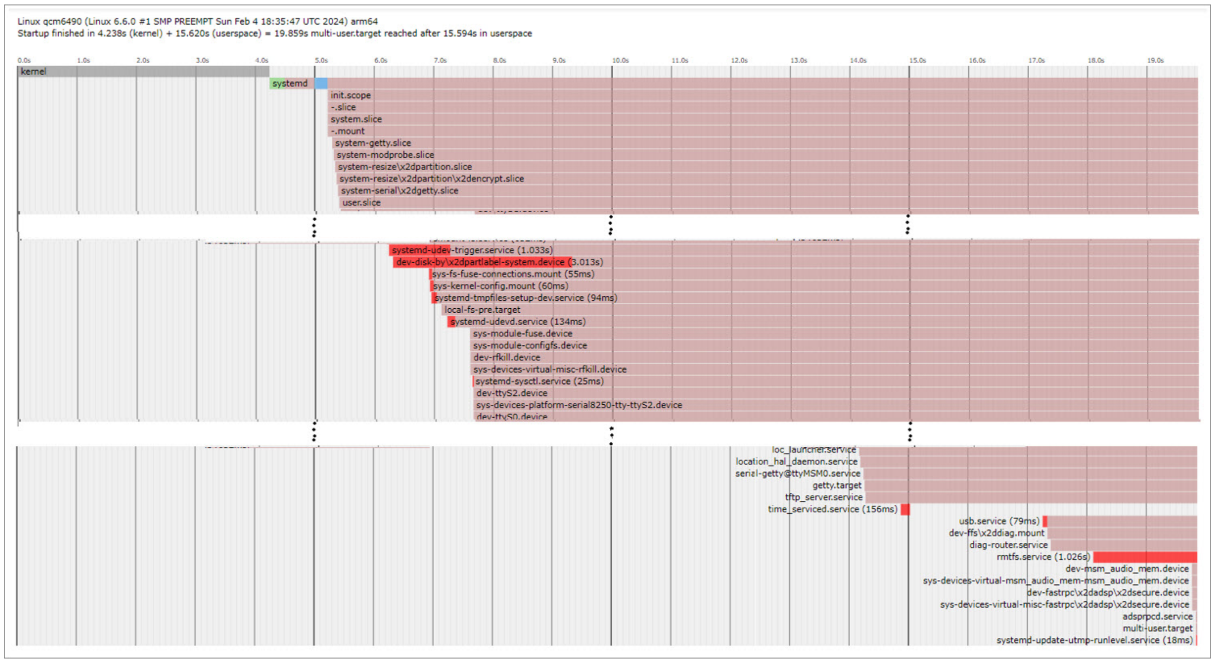
Graphical view of system initialization time

The `systemd-analyze plot` command provides a graphical breakdown of the system services that have started, along with their initialization times.

To obtain a graphical breakdown of the system services, run the following command on the device:

```
systemd-analyze plot > /var/lib/systemd-plot.svg
```

To visualize time consumption across the modules in the system initialization phase and analyze the performance, open the `systemd-plot.svg` file in any web browser. The following figure shows the example graph:



Linux qcm6490 (Linux 6.6.0 #1 SMP PREEMPT Sun Feb 4 18:35:47 UTC 2024) arm64
Startup finished in 4.238s (kernel) + 15.620s (userspace) = 19.859s multi-user.target reached after 15.594s in userspace

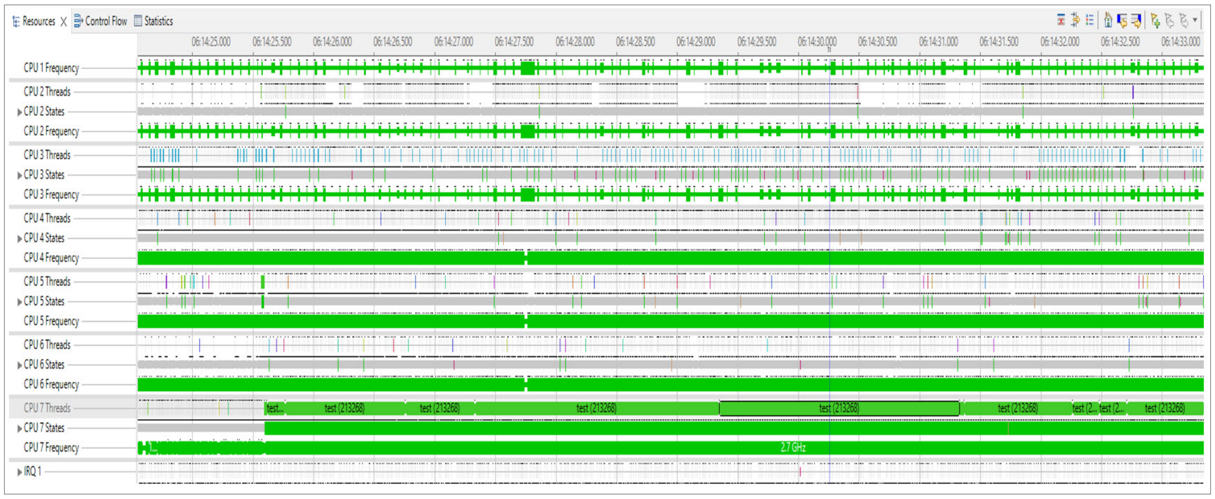
Identify CPU bound use cases

To verify that a task is running on the most capable CPUs at their maximum frequency, capture the scheduler and frequency ftrace.

The following is a sample code that loads the CPU using a `while` loop:

```
    #include <stdlib.h>
#include <unistd.h>

int main() {
     int i = 0;
     while(1)
    {
            i++;
        }
        return 0;
    }
```

You can collect an ftrace for the sample code and use Trace Compass to load the ftrace. It allows you to check if the test thread is running on the Prime core at the maximum CPU frequency of 2.7 GHz as shown in the following figure:



Identify I/O bound use cases

To obtain I/O statistics, use `/proc/diskstats`.

For more information, see kernel.org/doc/Documentation/ABI/testing/procfs-diskstats.

The following is an example of running lmdd on the device for the I/O-bound use case:

- Before running the use case, run the following command:

```
cat /proc/diskstats
```

  The following is an output of the command:

```
8 10 sda10 715 544 15056 250 4394 413 4199944 135729 0 5508
135979 0 0 0 0 0 0
```

  Next, get `pgpgin` and `pgpgout` from vmstat:

```
cat /proc/vmstat
```

The following is an output of the command:

```
pgpgin 348632pgpgout 2100056
```

- To get lmdd working on the build, compile lmbench, see Compile performance tools for more information.

  For the I/O-bound use case, run the following lmdd command:

```
lmdd if=/mnt/overlay/2GB.file of=/mnt/overlay/2GB.file.copy
fsync=1 bs=1M
```

- After running the use case, run the following command:

```
cat /proc/diskstats
```

The following is an output of the command:

```
8 10 sda10 4822 544 4209448 13018 8530 451 8394624 300094 0
11836 313112 0 0 0 0 0 0
```

Next, check `pgpgin` and `pgpgout` again:

```
cat /proc/vmstat
```

The following is an output of the command:

```
pgpgin 2446172pgpgout 4197396
```

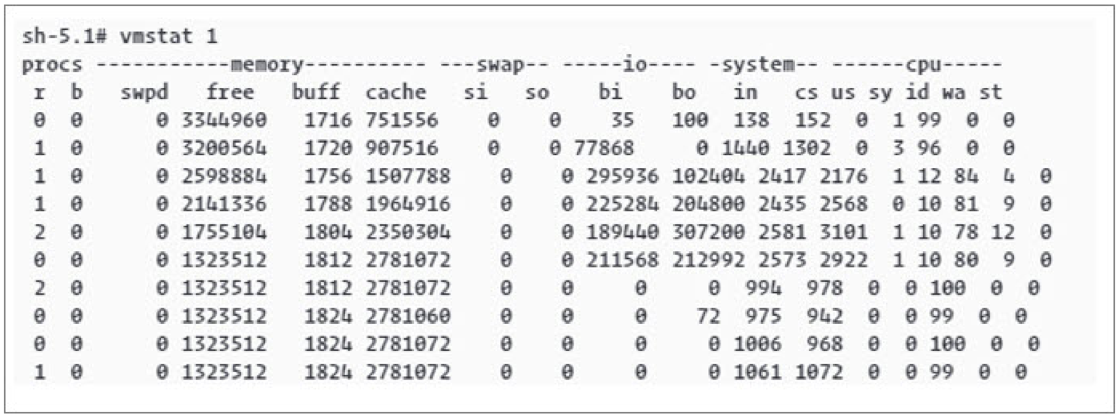
The following is an example of the statistics for an I/O-bound use case:

```
Sectors read = (4209448 - 15056) = 4194392 sectors = 2GB
Time spent reading = (13018 - 250) = 12768 ms
Sectors written = (8394624 - 4199944) = 4194680 sectors = 2GB
Time spent writing = (300094 -135729) = 164365 ms
Time spend IO = (11836 - 5508) = 6328 ms

pgpgin gap = (2446172-348632) = 2GB
pgpgout gap = (4197396 - 2100056) = 2GB
```

For more information about the I/O statistics fields, see
https://www.kernel.org/doc/Documentation/iostats.txt.

Vmstat

`Vmstat` is a Linux command used to gather information about block input (bi) and block output (bo). The following figure shows an example of the vmstat output:



```
sh-5.1# vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 3344960   1716 751556    0    0    35   100  138  152  0  1 99  0  0
 1  0      0 3200564   1720 907516    0    0 77868     0 1440 1302  0  3 96  0  0
 1  0      0 2598884   1756 1507788    0    0 295936 102404 2417 2176  1 12 84  4  0
 1  0      0 2141336   1788 1964916    0    0 225284 204800 2435 2568  0 10 81  9  0
 2  0      0 1755104   1804 2350304    0    0 189440 307200 2581 3101  1 10 78 12  0
 0  0      0 1323512   1812 2781072    0    0 211568 212992 2573 2922  1 10 80  9  0
 2  0      0 1323512   1812 2781072    0    0     0     0  994  978  0  0 100  0  0
 0  0      0 1323512   1824 2781060    0    0     0    72  975  942  0  0 99  0  0
 0  0      0 1323512   1824 2781072    0    0     0     0 1006  968  0  0 100  0  0
 1  0      0 1323512   1824 2781072    0    0     0     0 1061 1072  0  0 99  0  0
```

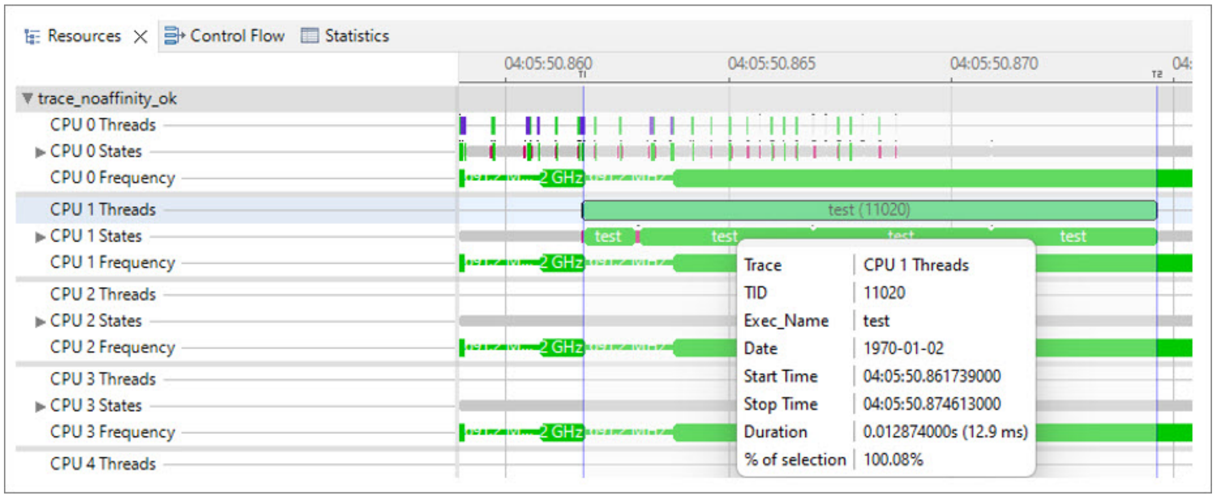
For more information, see https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html.

Use large cores for heavy use cases

When a heavy task runs on the Silver core with a high runtime, it can impact performance. Affine such tasks to the larger (Gold) cores using `sched_setaffinity()`. This task affinity can help to reduce the CPU runtime and enhance performance.

Caution: Any modification made to the nodes can impact the power and the performance of the device. It's important to verify the impact across all relevant use cases before changing the nodes.

The following figure from Trace Compass shows an example of a thread test running for 12.9 milliseconds on CPU0 at a frequency of 1.9 GHz.



Trace	CPU 1 Threads
TID	11020
Exec_Name	test
Date	1970-01-02
Start Time	04:05:50.861739000
Stop Time	04:05:50.874613000
Duration	0.012874000s (12.9 ms)
% of selection	100.08%

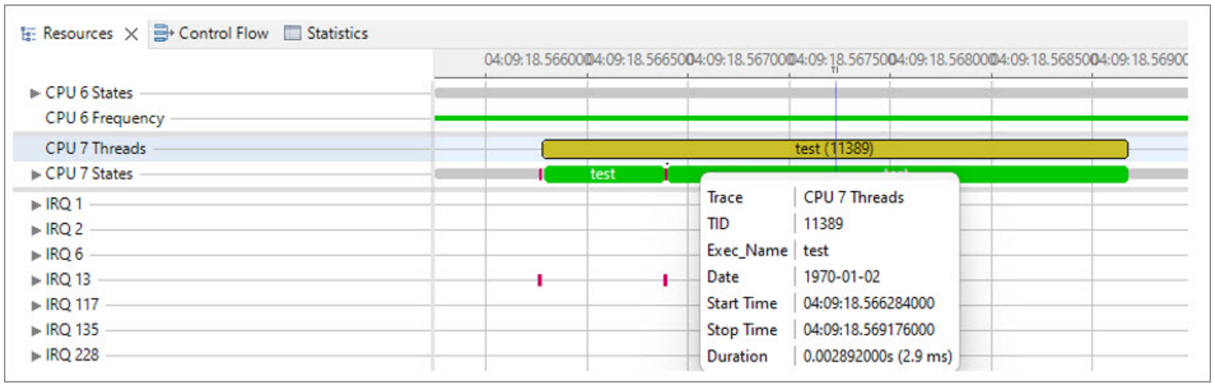
To affine a task to the Gold core using `sched_setaffinity()`, see https://man7.org/linux/man-pages/man2/sched_setaffinity.2.html.

The following is a sample code where a task is affined to Gold core 7:

```
#include <sched.h>
#include <unistd.h>
#include <sys/syscall.h>
cpu_set_t mask;
CPU_ZERO(&mask);
CPU_SET(7, &mask);
pid_t tid = syscall(__NR_gettid);
int result = sched_setaffinity(tid, sizeof(mask), &mask);
```

After the task is affined with `sched_setaffinity()`, it runs on CPU7 and the runtime is reduced from 12.9 milliseconds to 2.9 milliseconds with a CPU frequency of 2.7 GHz.

The following figure shows the reduced time after setting the `sched_setaffinity()` property:



Trace	CPU 7 Threads
TID	11389
Exec_Name	test
Date	1970-01-02
Start Time	04:09:18.566284000
Stop Time	04:09:18.569176000
Duration	0.002892000s (2.9 ms)

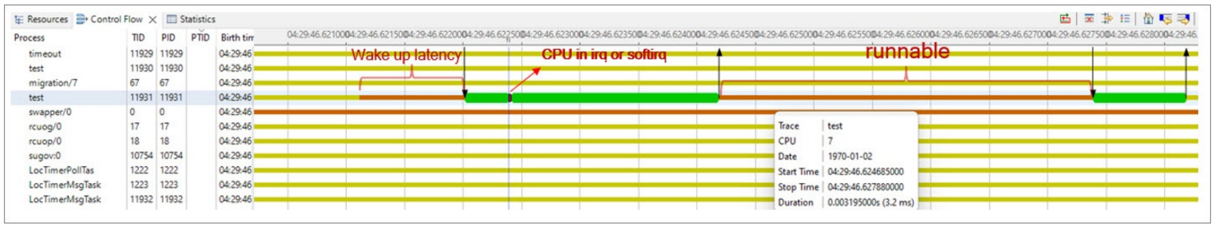
Mitigate impact of runnables on use cases

When a task is ready to run but the CPU is unavailable, the task is considered to be in a runnable state. This state is assigned to tasks when the CPU is under heavy load.

To visualize the status of threads, you can use the Trace Compass **Control Flow** view.

The following figure displays thread statuses represented by different colors:

- Dark red line indicates that the thread is in a runnable state
- Yellow lines represent the sleep state
- Red line indicates that the CPU is busy handling `irq` or `softirq`



Process	TID	PID	PTID	Birth tir
timeout	11929	11929		04:29:46
test	11930	11930		04:29:46
migration/7	67	67		04:29:46
test	11931	11931		04:29:46
swapper/0	0	0		04:29:46
rcuog/0	17	17		04:29:46
rcuop/0	18	18		04:29:46
sugov:0	10754	10754		04:29:46
LocTimerPollTas	1222	1222		04:29:46
LocTimerMsgTask	1223	1223		04:29:46
LocTimerMsgTask	11932	11932		04:29:46

Trace	test
CPU	7
Date	1970-01-02
Start Time	04:29:46.624685000
Stop Time	04:29:46.627880000
Duration	0.003195000s (3.2 ms)

Types of runnables:

- Wake-up latency runnable refers to the time it takes for tasks that are ready to move from a runnable state to actually running on the CPU. This latency can be reduced by tuning a scheduler or disabling the Low-power mode of the CPU.
- Normal runnable occurs when the CPU selects the higher-priority processes to run instead of the current one. Increasing the priority of a task can help reduce the runnables.

The priority of a thread depends on its type:

- The priority of a real-time (RT) thread ranges from 0 to 99, with a higher number indicating a higher priority.

  To change the real-time thread priority, use the `SCHED_FIFO` policy in `sched_setscheduler()`.

- The priority of a normal thread ranges from 100 to 139, with a lower number indicating a higher priority.

  To change the normal thread priority, use the `renice` Linux command and `sched_setscheduler()` with the `SCHED_OTHER` policy. The values in the range –20 to +19 are mapped to the thread priorities in the range 100 to 139.

To reduce the runnable time by changing the thread priority, use `sched_setscheduler()`.

For `sched_setscheduler()`, see sched_setscheduler(2)—Linux manual page.

The following is a sample code that reduces runnable time by changing the thread priority using `sched_setscheduler()`:

```
struct sched_param param = {0};
param.sched_priority = 1;
int ret=0;
ret = sched_setscheduler(0, SCHED_FIFO, &param);
```

The first parameter represents the Task ID. 0 represents the current task. The second parameter represents the scheduler policy. `SCHED_FIFO` is for the RT threads. The `sched_priority` is equal to 1.

```
0--> 99 ( RT class highest priority)
1 --> 99-1 --> 98
2 --> 99-2 --> 97
..
99 --> 99-99 --> 0 (RT least priority)
```

By default, the process priority is 120. It's inherited from the shell. The runnable time is 225 milliseconds and the runtime is 267 milliseconds. By increasing the process priority from 120 to 98 (real-time priority), the runnable duration reduces to less than 2 milliseconds.

Speed up CPU ramp-up time

A delay in transitioning to a higher required CPU frequency can impact performance. You can tune the `sched_util_clamp_min` scheduler node to speed up the CPU frequency ramp-up.

Tune the `sched_util_clamp_min` within a range of 0 to 1024. Higher values can enhance performance but may also increase power consumption.

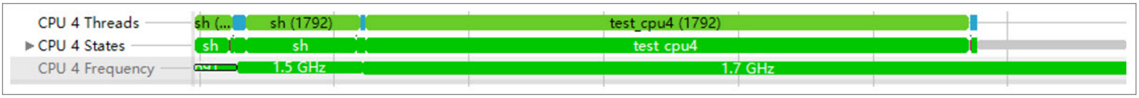
The following are examples of how the test thread performs on core 4:

- When `sched_util_clamp_min` is 0, the CPU frequency ramps up slowly from 691 MHz to 1.5 GHz and then to 1.7 GHz.

  You can set this value by running the following command on the device:

  ```
  echo 0 > /proc/sys/kernel/sched_util_clamp_min
  ```

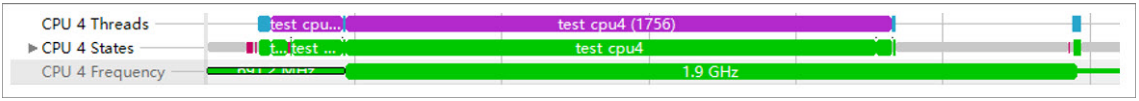
  The following figure from Trace Compass shows the ramping up of the CPU frequency:

- When `sched_util_clamp_min` is 512, the CPU frequency ramps up directly from 691 MHz to 1.9 GHz. You can set this value by running the following command on the device:

  ```
  echo 512 > /proc/sys/kernel/sched_util_clamp_min
  ```

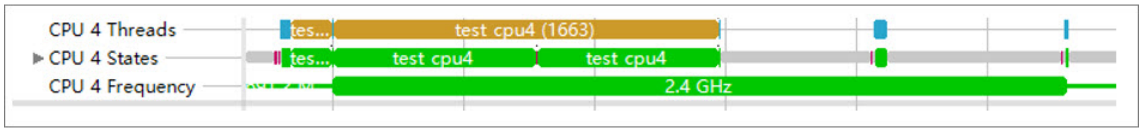
  The following figure shows the ramping up of the CPU frequency to 1.9 GHz:

- When `sched_util_clamp_min` is 1024, the CPU frequency ramps up from 691 MHz directly to the maximum frequency (FMAX) of 2.4 GHz. You can set this value by running the following command on the device:

  ```
  echo 1024 > /proc/sys/kernel/sched_util_clamp_min
  ```

  The following figure shows the ramping up of the CPU frequency directly from 691 MHz to FMAX 2.4 GHz:

Determine cache residency for use cases

The perf utility tool is used to analyze cache misses and cache refill counter statistics. This analysis helps to determine the residency of a use case in a specific cache, such as L2, L3, and last level cache controller (LLCC) DDR residency.

For instructions on how to compile the perf utility, see Compile performance tools.

To check the available cache event for the target, run the following command on the device:

```
perf list | grep cache
```

The following is an example command to obtain the cache residency:

```
perf stat -e l1d_cache_lmiss_rd -e l1i_cache_lmiss -e l2d_cache_
lmiss_rd -e l3d_cache_lmiss_rd -e ll_cache_miss_rd  sleep 5
```

Cache miss counters in the CPU path, from the previous cache levels (L1 → L2 → L3 → LLCC → DDR) indicate the residency of the use case in the subsequent cache.

The following sample code provides cache miss counter statistics:

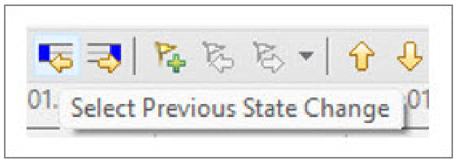
```
Performance counter stats for '5 duration':

           5797      l1d_cache_lmiss_rd
          26699      l1i_cache_lmiss
          16200      l2d_cache_lmiss_rd
           8634      l3d_cache_lmiss_rd
           9710      ll_cache_miss_rd

    5.004388332 seconds time elapsed

    0.001599000 seconds user
    0.000000000 seconds sys
```

Identify lock contention

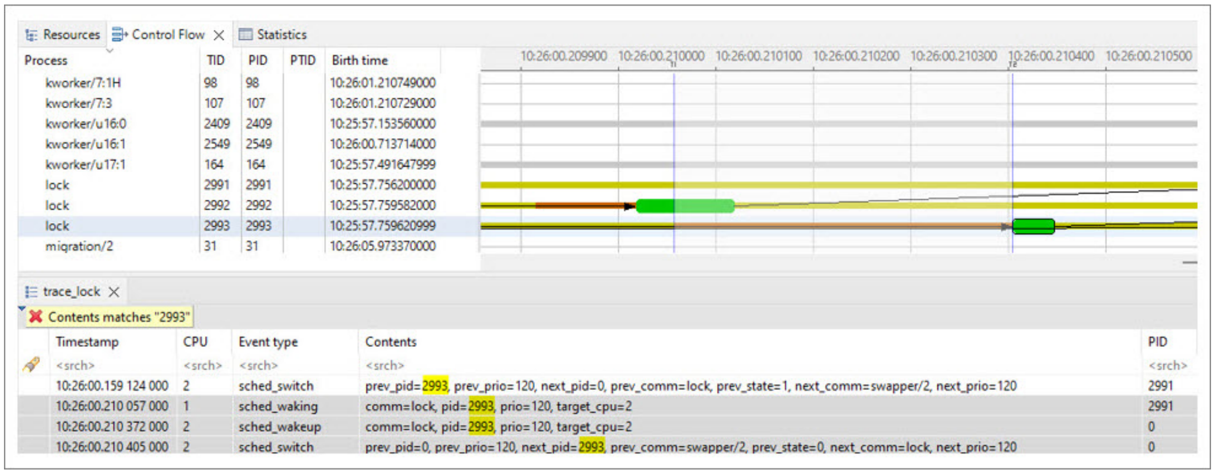
Lock contention occurs when one thread (thread_1) attempts to acquire a Mutex lock that's already held by another thread (thread_2).

In this situation, thread_1 enters the Sleep mode and wakes up when thread_2 releases the Mutex lock.

To resolve this issue, go to **Trace Compass** and select **Select Previous State Change** as shown in the following figure:



The following figure shows an instance where thread 2991 wakes up thread 2993:



Process	TID	PID	PTID	Birth time
kworker/7:1H	98	98		10:26:01.210749000
kworker/7:3	107	107		10:26:01.210729000
kworker/u16:0	2409	2409		10:25:57.153560000
kworker/u16:1	2549	2549		10:26:00.713714000
kworker/u17:1	164	164		10:25:57.491647999
lock	2991	2991		10:25:57.756200000
lock	2992	2992		10:25:57.759582000
lock	2993	2993		10:25:57.759620999
migration/2	31	31		10:26:05.973370000

Timestamp	CPU	Event type	Contents	PID
<srch>	<srch>	<srch>	<srch>	<srch>
10:26:00.159 124 000	2	sched_switch	prev_pid=2993, prev_prio=120, next_pid=0, prev_comm=lock, prev_state=1, next_comm=swapper/2, next_prio=120	2991
10:26:00.210 057 000	1	sched_waking	comm=lock, pid=2993, prio=120, target_cpu=2	2991
10:26:00.210 372 000	2	sched_wakeup	comm=lock, pid=2993, prio=120, target_cpu=2	0
10:26:00.210 405 000	2	sched_switch	prev_pid=0, prev_prio=120, next_pid=2993, prev_comm=swapper/2, prev_state=0, next_comm=lock, next_prio=120	0

Determine duration of pre-emption disabling

The kernel operates on a pre-emptive basis. This means that any kernel process can be paused at any moment to make way for a higher priority process. Therefore, a new task can start running in the same critical region where a previous task was pre-empted.

The following procedure outlines how to record the duration during which pre-emption is disabled:

1. From the kernel configuration, enable `CONFIG_IRQSOFF_TRACER` and `CONFIG_PREEMPT_TRACER` in the source code.
2. To collect a trace, run the following commands:

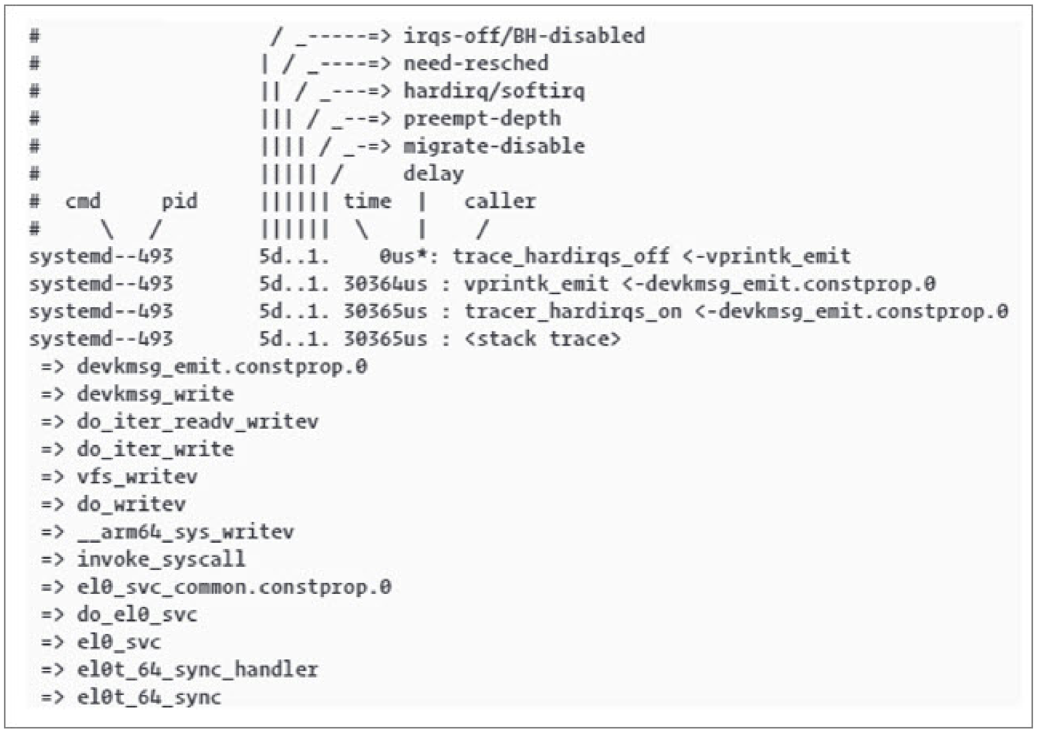
Note: The following commands should be run on the device.

```
echo preemptoff > /sys/kernel/tracing/current_tracer
```

```
echo 1 > /sys/kernel/tracing/tracing_on
```

```
cat /sys/kernel/tracing/trace
```

As shown in the figure, a timestamp is recorded for each instance of pre-emption being disabled, marking the start and end points in the code:



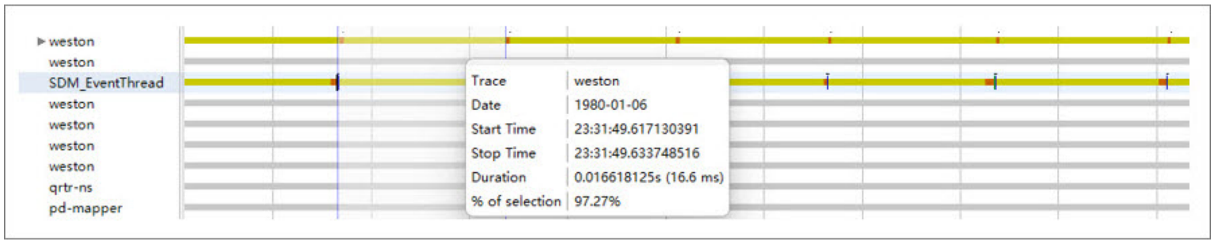
```
#                    / _-----=> irqs-off/BH-disabled
#                   | / _----=> need-resched
#                   || / _---=> hardirq/softirq
#                   ||| / _--=> preempt-depth
#                   |||| / _-=> migrate-disable
#                   ||||| /     delay
#  cmd     pid      |||||| time  |   caller
#     \   /         ||||||  \    |    /
systemd--493       5d..1.    0us*: trace_hardirqs_off <-vprintk_emit
systemd--493       5d..1. 30364us : vprintk_emit <-devkmsg_emit.constprop.0
systemd--493       5d..1. 30365us : tracer_hardirqs_on <-devkmsg_emit.constprop.0
systemd--493       5d..1. 30365us : <stack trace>
 => devkmsg_emit.constprop.0
 => devkmsg_write
 => do_iter_readv_writev
 => do_iter_write
 => vfs_writev
 => do_writev
 => __arm64_sys_writev
 => invoke_syscall
 => el0_svc_common.constprop.0
 => do_el0_svc
 => el0_svc
 => el0t_64_sync_handler
 => el0t_64_sync
```

For more information about function tracer, see kernel.org/doc/Documentation/trace/ftrace.txt.

Debug frame drops

Frame drops can occur due to delays in various subsystems, such as the display or camera. For example, if the display refresh rate is 60 Hz, each frame must be completed within 16.6 milliseconds.

The following figure shows a trace where `Weston` and `SDM_EventThread` run every 16.6 milliseconds. Any application must render periodically and complete its rendering within this 16.6 milliseconds timeframe. If rendering isn't complete before this window expires, the frames are dropped.



Trace	weston
Date	1980-01-06
Start Time	23:31:49.617130391
Stop Time	23:31:49.633748516
Duration	0.016618125s (16.6 ms)
% of selection	97.27%

Identify memory thrashing

Memory thrashing occurs when the system spends a significant amount of time reclaiming memory from RAM and then reloads the same content back into RAM.

This can occur on file cache pages from disk and anonymous pages from ZRAM, leading to substantial performance degradation.

Memory thrashing typically occurs when the available memory is insufficient for the current use case (referred to as the workingset). This causes the system to struggle in finding memory that can be reclaimed.

You can identify memory thrashing from the following information in `/proc/vmstat`:

vmstat nodes	Description
`workingset_refault_anon/workingset_refault_file`	These nodes represent the number of reclaimed pages that are immediately requested after reclaim. The lower these numbers, the better.
`workingset_activate_anon/workingset_activate_file`	These nodes represent the number of reclaimed pages that are immediately activated after reclaim. The lower these numbers, the better.
`pgpgin/pswpin`	These nodes represent the number of pages read from swap and swapped back into the RAM memory.

vmstat nodes	Description
`pgpgout/pswpout`	These nodes represent the number of pages written to swap as part of reclaim. If `pgpg*` and `pswp*` are increasing simultaneously along with `workingset_refaults`, it indicates a memory thrashing situation.
`pgsteal_kswapd/pgsteal_direct`	These nodes represent the number of pages reclaimed by the system.
`pgscan_kswapd/pgscan_direct`	These nodes represent the number of pages that the system has scanned to find reclaimable memory. The ratio of `pgsteal/pgscan` indicates the reclaim efficiency of the system. A higher value indicates better system performance while a lower reclaim efficiency indicates that the system is struggling to find reclaimable memory, indicative of memory thrashing.

To identify memory thrashing, run the following command on the device:

```
cat /proc/vmstat
```

The vmstat fields are as follows:

```
workingset_refault_anon 984111
workingset_refault_file 1838690
workingset_activate_anon 502428
workingset_activate_file 499034
pgpgin 17488312
pgpgout 3398036
pswpin 984141
pswpout 2101230
pgsteal_kswapd 3946686
pgsteal_direct 59226
pgscan_kswapd 4660928
pgscan_direct 73719
```

These counters increase linearly over time.

To detect patterns in memory thrashing, gather data from these counters at regular intervals. Then, plot this data over a specific time period to visualize the patterns.

8 Performance dashboards

You can access performance dashboards for boot time, system benchmarks, memory map, and product segment key performance indicators (KPIs) on the Qualcomm® Linux® reference devices.

The following subsections describe the performance dashboards and measurement procedures for QCS6490 and QCS5430.

For QCS9075 and QCS8275 performance dashboards and measurement procedures, see the corresponding addendum. The following guides provide supplementary information and these guides are available to licensed users with authorized access:

- Qualcomm Linux Performance Guide - Addendum for QCS9075
- Qualcomm Linux Performance Guide - Addendum for QCS8275

QCS6490

8.1 Boot time

The boot time is the duration from device power-on until the initialization of the recorder service.

The following table lists the measured boot time (values in seconds) on QCS6490:

Use case	Score
Boot time (log-based)	8.13

Note: A lower boot time score is better.

For information about the measurement procedure, see Boot time measurement.

System benchmarks

Geekbench is a utility for measuring CPU performance. It provides the following CPU benchmark scores on QCS6490:

Benchmark	Version	Benchmark score
Geekbench ST	6.1.0	1178
Geekbench MT	6.1.0	3048

For information about the measurement procedure, see System benchmark measurement.

Note: A higher system benchmark score is better.

Memory map

The following table lists the memory consumption (values in MB) for each partition, such as non-Linux, kernel static, and applications. It also lists the total free memory available to the system after device boot and during use cases such as 4k resolution encoding at 30 fps.

Memory partitions	After boot	4K30 encode	4K30720p30 encode	4K30 encode sHDRv2	4K30 encode sHDRv3
Total RAM	6144	6144	6144	6144	6144
Non-Linux	620	620	620	620	620
Kernel static	152	152	152	152	152
Applications + framework	566	1292	1356	1570	1915
Total free memory	4806	4080	4016	3802	3457

For information about the measurement procedure, see Memory map measurement.

Note: A higher total free memory value is better for the system performance.

Use case KPIs

The following table lists the camera latency measurement data (values in seconds) for various QCS6490 camera use cases:

Use case	Latency definition	Latency
4K first snapshot latency	First time after boot, time taken from capturing an image to creating a snapshot	0.376
4K subsequent snapshot latency	Time taken from capturing an image to creating a snapshot (and subsequent snapshots)	0.365
4K30 encoding first record latency	First time after boot, time taken from GStreamer Connect to the first video-encoded frame	4.093
4K30 encoding subsequent record latency	Time taken from GStreamer Connect to the subsequent video-encoded frame	0.315
4K30 sHDRv2 encoding first record latency	First time after boot, time taken from GStreamer Connect to the first video-encoded frame for 4K30 sHDRv2	4.483
4K30 sHDRv2 encoding subsequent record latency	Time taken from GStreamer Connect to the subsequent video-encoded frame for 4K30 sHDRv2	0.442
4K30 sHDRv3 encoding first record latency	First time after boot, time taken from GStreamer Connect to the first video-encoded frame for 4K30 sHDRv3	4.395
4K30 sHDRv3 encoding subsequent record latency	Time taken from GStreamer Connect to the subsequent video-encoded frame for 4K30 sHDRv3	0.416
4K30 G2G (Glass to Glass) preview	Time taken from the sensor to output the frame until the frame is previewed on the external display	0.2

Note: A lower camera latency value is better.

For information about the measurement procedure, see Camera recording/snapshot latency measurement.

QCS5430

8.2 Boot time

The boot time is the duration from device power-on until the initialization of the recorder service.

The following table lists the measured boot time (values in seconds) on QCS5430:

Use case	FP 1 score	FP 2 score	FP 2.5 score	FP 3 score
Boot time (log-based)	8.7	8.25	8.33	8.29

Note: A lower boot time score is better.

For information about the measurement procedure, see Boot time measurement.

System benchmarks

Geekbench is a utility for measuring CPU performance. It provides the following CPU benchmark scores on QCS5430:

Benchmark	Version	Benchmark score for FP 1	Benchmark score for FP 2	Benchmark score for FP 2.5	Benchmark score for FP 3
Geekbench ST	6.1.0	955	984	1061	1053
Geekbench MT	6.1.0	2191	2854	2933	2953

For information about the measurement procedure, see System benchmark measurement.

Note: A higher system benchmark score is better.

Memory map

The following table lists the memory consumption (values in MB) for each partition such as non-Linux, kernel static, and applications. It also lists the total free memory available to the system after device boot and during use cases such as 4k resolution encoding at 30 fps.

Memory partitions	**After boot**	**4K30 encode**	**4K30720p30 encode**	**4K30 encode sHDRv2**	**4K30 encode sHDRv3**
Total RAM	6144	6144	6144	6144	6144
Non-Linux	621	621	621	621	621
Kernel static	151	151	151	151	151
Applications + framework	571	1295	1358	1564	1876
Total free memory	4801	4077	4014	3808	3496

For information about the measurement procedure, see Memory map measurement.

Note: A higher total free memory value is better for the system performance.

Use case	**Latency definition**	**Latency for FP 1**	**Latency for FP 2**	**Latency for FP 2.5**	**Latency for FP 3**

Use case KPIs

The following table lists the camera latency measurement data (values in seconds) for various QCS5430 camera use cases:

Use case	**Latency definition**	**Latency for FP 1**	**Latency for FP 2**	**Latency for FP 2.5**	**Latency for FP 3**
4K first snapshot latency	First time after boot, time taken from capturing an image to creating a snapshot	0.375	0.378	0.367	0.361
4K subsequent snapshot latency	Time taken from capturing an image to creating a snapshot (and subsequent snapshots)	0.357	0.363	0.351	0.350
4K30 encoding first record latency	First time after boot, time taken from GStreamer Connect to the first video-encoded frame	3.943	4.045	4.024	4.018
4K30 encoding subsequent record latency	Time taken from GStreamer Connect to the subsequent video-encoded frame	0.353	0.361	0.363	0.398
4K30 sHDRv2 encoding first record latency	First time after boot, time taken from GStreamer Connect to the first video-encoded frame for 4K30 sHDRv2	4.723	4.511	4.512	4.459
4K30 sHDRv3 encoding first record latency	First time after boot, time taken from GStreamer Connect to the first video-encoded frame for 4K30 sHDRv3	4.742	4.414	4.412	4.344
4K30 G2G (Glass to Glass) preview	Time taken from the sensor to output the frame until the frame is previewed on the external display	0.2	0.2	0.2	0.2

Note: A lower camera latency value is better.

For information about the measurement procedure, see Camera recording/snapshot latency measurement.

8.3 Measurement procedures

The measurement procedures include KPIs, such as boot time, system benchmark, record latency, and snapshot latency.

Boot time measurement

The boot time is the duration from device power-on to the initialization of the recorder service.

To measure boot time, do the following:

1. Collect serial logs during the device power-on process, focusing on the boot loader time. To collect the serial logs on a Linux host, do the following:
   a. Connect a serial cable between the device and the Linux host PC.
   b. Connect to the UART terminal to obtain serial logs. To set up the UART terminal, see Connect to a UART shell.
   c. Power-off the device.
   d. Power-on the device.
   e. Save the serial logs from the terminal.
2. After the device boots up and stabilizes, run the following commands on the device to collect logs:

```
journalctl --output=short-monotonic -b --no-pager -l > /
var/lib/journalctl.txt
```

```
systemd-analyze time > /var/lib/systemd-time.txt
```

For more information about the systemd-analyze tool, see Analysis tools.

The following table describes the boot time (values in seconds) measurement procedure with log markers for boot time KPI:

Boot time stages	**Logs**	**Log markers for start and end point**	**Calculation**	**Boot time**
PBL+XBL	Serial logs	`UEFI Start`	`UEFI Start [1818]`	1.818
Core UEFI	Serial logs	`UEFI Total`	`UEFI Total: 1071 milliseconds`	1.071
Kernel loader	Serial logs	`Exit EBS [3469]` `UEFI End`	`Exit EBS - (Core UEFI + PBL+XBL)`	3.469 - (1.071 + 1.818) = 0.580
Kernel total time	systemd-analyze	`Kernel init time`	`Start-up finished in 3.300 seconds (kernel) + 2 min 6.300 seconds (user space) = 2 min 9.600 seconds multi-user.target reached after 2 min 6.280 seconds in user space`	3.300
Kernel init time	journalctl.txt	`enforcing=1 old_enforcing=0`	`1980-01-06T00:00:01.538918+00:00 qcm6490 kernel: [1.854242][T77] audit: type=1404 audit(5.003:2): enforcing=1 old_enforcing=0 auid=4294967295 ses=4294967295 enabled=1 old-enabled=1 lsm=selinux res=1`	1.854

Boot time stages	**Logs**	**Log markers for start and end point**	**Calculation**	**Boot time**
Recorder init time	journalctl.txt	`Recorder time: Spectra camera driver initialized kernel init time: enforcing=1 old_enforcing=0 Recorder init time = recorder time - kernel init time`	`1980-01-06T00:02:00.431978+00:00 qcm6490 kernel: [6.493781][T551] CAM_INFO: CAM-UTIL: camera_ init: 297 Spectra camera driver initialized.`	6.494
Total time until recorder init	–	–	`PBL+XBL + Core UEFI + Kernel Loader + Kernel total Time + (recorder init time - user space)`	11.409

System benchmark measurement

Geekbench is a tool used to measure system performance against the established benchmarks.

To measure system performance using Geekbench, do the following:

1. Download Geekbench for the Linux/ARM architecture from upstream https://www.geekbench.com/preview/.

   Note: The Geekbench 6 for Linux/AArch64 is a preview build. The preview builds require an active Internet connection and automatically upload benchmark results to the Geekbench browser.

2. To measure a CPU benchmark using Geekbench, do the following:

   a. Unzip the Geekbench file and push it from the host into the device, use SCP or a similar tool. Ensure that you specify the target IP address in the first command. The following are the example commands:

   ```
   scp -r Geekbench-6.3.0-LinuxARMPreview root@10.92.174.66:/
   var/cache/
   ```

   ```
   cd /var/cache
   ```

   ```
   chmod 777 Geekbench-6.3.0-LinuxARMPreview/*
   ```

   b. To run Geekbench, run the following commands on the device:

   ```
   cd Geekbench-6.3.0-LinuxARMPreview
   ```

```
./geekbench_aarch64
```

Memory map measurement

A memory map provides information on how memory is allocated for different processes. A memory map measurement allows you to monitor a mapped process and troubleshoot any memory issues.

To calculate the memory map, boot the device and stabilize it. Then, run the following commands to collect logs from the device:

```
cat /proc/meminfo
```

```
cat /proc/iomem
```

```
cat /proc/vmstat
```

Non-Linux memory

Use the following formula to calculate the non-Linux memory:

Non-Linux = Total RAM size Total Linux

Run the following command to calculate the total RAM size: `cat /proc/meminfo | grep -i "MemTotal"`.

MemTotal is 5512456 kB, which corresponds to approximately 6 GB of RAM.

To calculate the total Linux memory from `iomem`, run the following command:

```
cat /proc/iomem | grep System
```

The following is an output of the command:

```
83600000-839fffff : System RAM
9c700000-9d08dfff : System RAM
9d096000-9d0a0fff : System RAM
9d0a9000-9d4ccfff : System RAM
9d4dc000-9d58efff : System RAM
9d598000-9e813fff : System RAM
9e833000-9e87dfff : System RAM
9e887000-9e890fff : System RAM
9e899000-9ed52fff : System RAM
9edcb000-9f7fbfff : System RAM
9f800000-9f9fffff : System RAM
9fc00000-a00cffff : System RAM
e3400000-1ffffffff: System RAM
```

The total Linux memory is the sum of the differences in the system RAM addresses.

For example:

839fffff 83600000 = 4194303 bytes = 3.99 MB

Kernel static

Use the following formula to calculate the kernel static:

Kernel static = Total Linux MemTotal

MemTotal is available in meminfo as follows:

```
MemTotal: 4513944 kB
```

Application + framework memory calculation

Use the following formula to calculate the memory used by the applications and framework:

Application + framework = MemTotal Free memory

Free memory calculation

Use the following formula to calculate the free memory:

Free memory = MemFree + (Cached shmem) + buffer + ION cache

To obtain the free memory details, run the following command:

```
cat /proc/meminfo
```

The following is an output of the command:

```
MemTotal:      4513944 kB
MemFree:       3471436 kB
MemAvailable: 3816716 kB
Buffers:       9216 kB
Cached:        574856 kB
Shmem:         24776 kB
```

To check the vmstat logs for ION cache, run the following command:

```
cat /proc/vmstat
nr_kernel_misc_reclaimable 16217
```

Here, 16217 pages represent approximately 63.3 MB. The calculation is as follows:

16217 pages $\times$ 4 kB/page = 64,868 kB (since 1 kB = 1024 bytes)

To convert to megabytes (MB), calculate the ION cache as follows:

64,868 kB $\div$ 1024 = 63.3 MB.

Camera recording/snapshot latency measurement

This topic describes the camera recording/snapshot latency measurement procedure.

To perform presetting on every reboot, run the following commands on the device:

```
mount -t debugfs none /sys/kernel/debug
```

```
setprop persist.qmmf.kpi.debug 2
```

First record latency

To capture the logs, follow these steps:

1. Boot the device and wait for it to stabilize.
2. In one shell, run the following command:

   ```
   cat /sys/kernel/debug/tracing/trace_pipe > trace.log
   ```

3. In another shell, run the use case.
4. Stop the trace log.
5. See the reference table to measure the record latency.

Subsequent record latency

1. Boot the device.
2. After the device stabilizes, run the use case.
3. Stop the use case.
4. In one shell, run the following command:

   ```
   cat /sys/kernel/debug/tracing/trace_pipe > trace.log
   ```

5. In another shell, run the use case.
6. Stop the trace log.
7. See the reference table to measure the record latency.

First snapshot latency

1. Boot the device.
2. After the device stabilizes, in one shell, run the following command:

```
cat /sys/kernel/debug/tracing/trace_pipe > trace.log
```

3. In another shell, run the use case.
4. Stop the trace log.
5. See the reference table to measure snapshot latency.

Subsequent snapshot latency

1. Boot the device.
2. After the device stabilizes, run the use case in another shell.
3. Stop the use case.
4. In one shell, run the following command:

```
cat /sys/kernel/debug/tracing/trace_pipe > trace.log
```

5. Run the use case.
6. Stop the trace log.
7. See the reference table to measure snapshot latency.

Glass to glass preview latency

1. Start a camera preview.
2. Start a stopwatch timer on any display and face the camera sensor on timer.
3. Connect the external display.
4. Capture a photo of the external display preview and stopwatch timer window in the same capture using an external camera.
5. Note the time difference of a stopwatch timer and external display preview.

Reference table for measurement

Use cases	Log marker	Calculation	Latency
Record latency	`273.082629: tracing_mark_write: B|Connect 273.119992: tracing_mark_write: B|StartCamera 276.722946: tracing_mark_write: B|CreateVideoTrack 276.731158: tracing_mark_write: B|StartVideoTrack 277.306283: tracing_mark_write: E|FirstVidFrame|1`	`Connect to Start Camera = StartCamera Connect StartCamera to CreateVideoTrack = CreateVideoTrack StartCamera CreateVideoTrack to StartVideoTrack = StartVideoTrack CreateVideoTrack StartVideoTrack to FirstVidFrame = FirstVidFrame StartVideoTrack Rec latency = sum of above all`	Connect to Start Camera = 273.120 273.083 = 0.037 StartCamera to CreateVideoTrack = 276.722 273.120 = 3.602 CreateVideoTrack to StartVideoTrack = 276.723 276.722 = 0.001 StartVideoTrack to FirstVidFrame = 277.306 276.731 = 0.575 Record latency = 4.223
Snapshot latency	`303.975067: tracing_mark_write: S|FirstCapImg|0 304.544265: tracing_mark_write: S|SnapShot-Shot|0`	`Snapshot latency = snapshot FirstCapimg`	Snapshot latency = 304.544 303.975 = 569

9 References

9.1 Related documents

Title	Number
Qualcomm Technologies, Inc.	
Qualcomm Linux Build Guide	80-70018-254
Qualcomm Software Center	80-72780-2
Qualcomm Linux Performance Guide - Addendum for QCS9075 Available to licensed users with authorized access.	80-70018-10A
Qualcomm Linux Performance Guide - Addendum for QCS8275 Available to licensed users with authorized access.	80-70018-10B
Resources	
https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt	
https://www.kernel.org/doc/html/v5.0/admin-guide/pm/cpufreq.html	
https://lwn.net/Articles/531853/	
https://lwn.net/ml/linux-kernel/20220531105137.110050-1-krzysztof.kozlowski@linaro.org/	
https://lttng.org/docs/v2.13/	
perf: Linux profiling with performance counters	
https://www.freedesktop.org/software/systemd/man/latest/systemd-analyze.html	

9.2 Acronyms and terms

Acronym or term	Definition
BWMON	Bandwidth monitoring
DTSI	Device tree source include
DVFS	Dynamic voltage and frequency scaling
EEVDF	Earliest eligible virtual deadline first
Ftrace	Function trace
GMEM	GPU memory
Kswapd	Kernel swap daemon
LLCC	Last level cache controller
LTTng	Linux Trace Toolkit next generation
PELT	Per entity load tracking
PerfHAL	Performance abstraction layer
PMU	Performance monitoring unit
QSC	Qualcomm Software Center
QTEE	Qualcomm® Trusted Execution Environment
RT	Real-time (kernel)
SoC	System-on-chip
TZ	TrustZone
UCLAMP	Utilization clamping