

Google Automotive Partner Bootcamp

Performance Analysis / Tuning 101

Google Automotive Partner Bootcamp



Ethan Lee

Software Engineer



Agenda



O2 Getting Started with Perfetto

O3 Anatomy of a Trace

04 Perfetto Pitfalls



•5 Trace AnalysisWalkthrough

o6 SQL Queries

Making DebuggingEasier

08 Performance Tuning

A Scientific Approach to Performance Analysis





What is Performance Analysis?

- Performance issues require a systematic process to uncover their root cause.
- The right tools need to be identified to gather insights into critical parts of complex systems.
- There are a number of techniques which engineers can use to delve deeper into the execution of a system.



What is Performance Analysis?

There are two techniques that are widely used for performance analysis: Tracing and Profiling



Tracing

- Tracing involves collecting highly detailed data about system execution.
- Traces contain enough detail to build a timeline of events.
- Traces give us insight into what a program does over time (e.g. which functions are being run) and context about execution (e.g. function call parameters).



Profiling

- The most common types are memory profiling and CPU profiling.
- Memory profiling surfaces information about heap memory allocation.
- CPU profiling gathers information about the call stack running on a CPU over time.

Profiling involves sampling some usage of a resource by a program.

Why Choose Perfetto?

Profiling and tracing have different use cases:

Why use profiling over tracing?

- Traces, while detailed, are impractical for capturing high-frequency events like every function call due to the sheer volume of data involved.
- Profilers address this limitation through sampling, selectively recording data points to drastically reduce storage requirements.

Why use tracing over profiling?

- Profilers offer valuable insights into where resources are consumed within a program's call stack, but they lack the ability to explain the underlying reasons behind those resource allocations.
- For instance, a profiler might reveal that function foo() called malloc numerous times and allocated X bytes, but it cannot tell us why foo() was making those calls.
- Traces fill this gap by combining application and kernel events, providing in-depth context to understand the root cause of resource consumption.

Perfetto addresses this by supporting the collection, analysis and visualization of both tracing and profiling.

How to Use Perfetto Effectively

How does Perfetto and our performance analysis flow fit into our goals?



This approach allows one to easily compare the delta of a potential regression. To achieve this, one should have an established baseline to compare against.

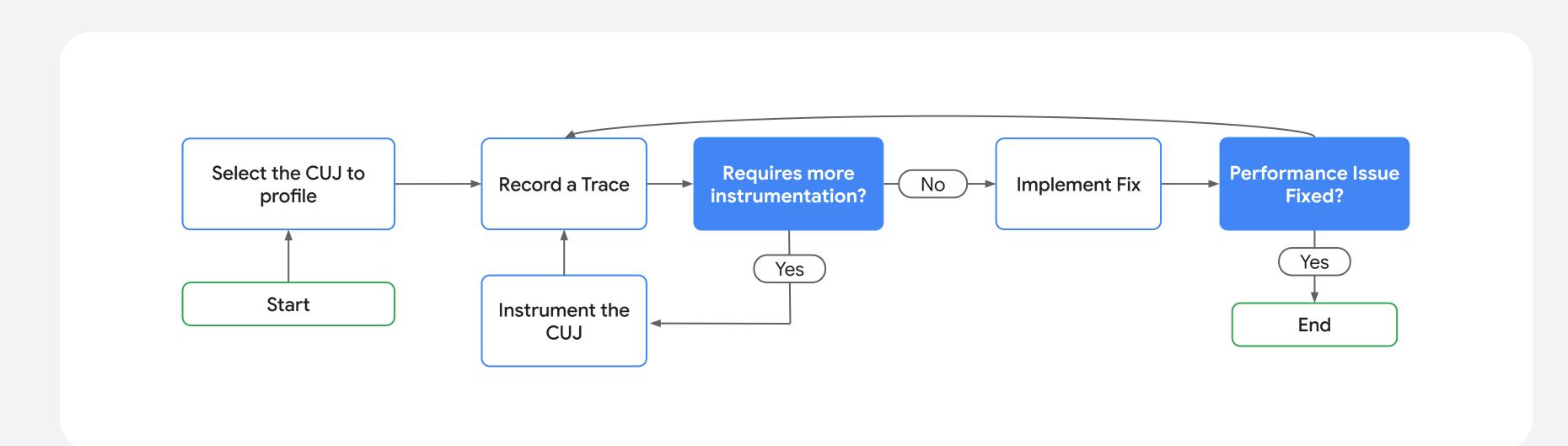
Perfetto will enable one to gather insights beyond just surface level observations. It is imperative that we can translate user-perceptible signals into measurable metrics that can be tested.



Using Perfetto in this approach means that multiple iterations will need to be collected. In order to establish reliable metrics, it is necessary to gather information on a large enough population to capture reproducible issues.

How to Use Perfetto Effectively

How does Perfetto and our performance analysis flow fit into our goals?



Perfetto: A Feature Rich Tool

Ease of Use

Perfetto provides an end-to-end solution to capture Android system traces quickly to identify issues in critical user flows.



Flexibility

Via Perfetto trace configs, users are able to modify tracing behavior via buffers or data sources. For example, one can easily change data sources to capture various ftrace events or atrace events.



Trace Analysis

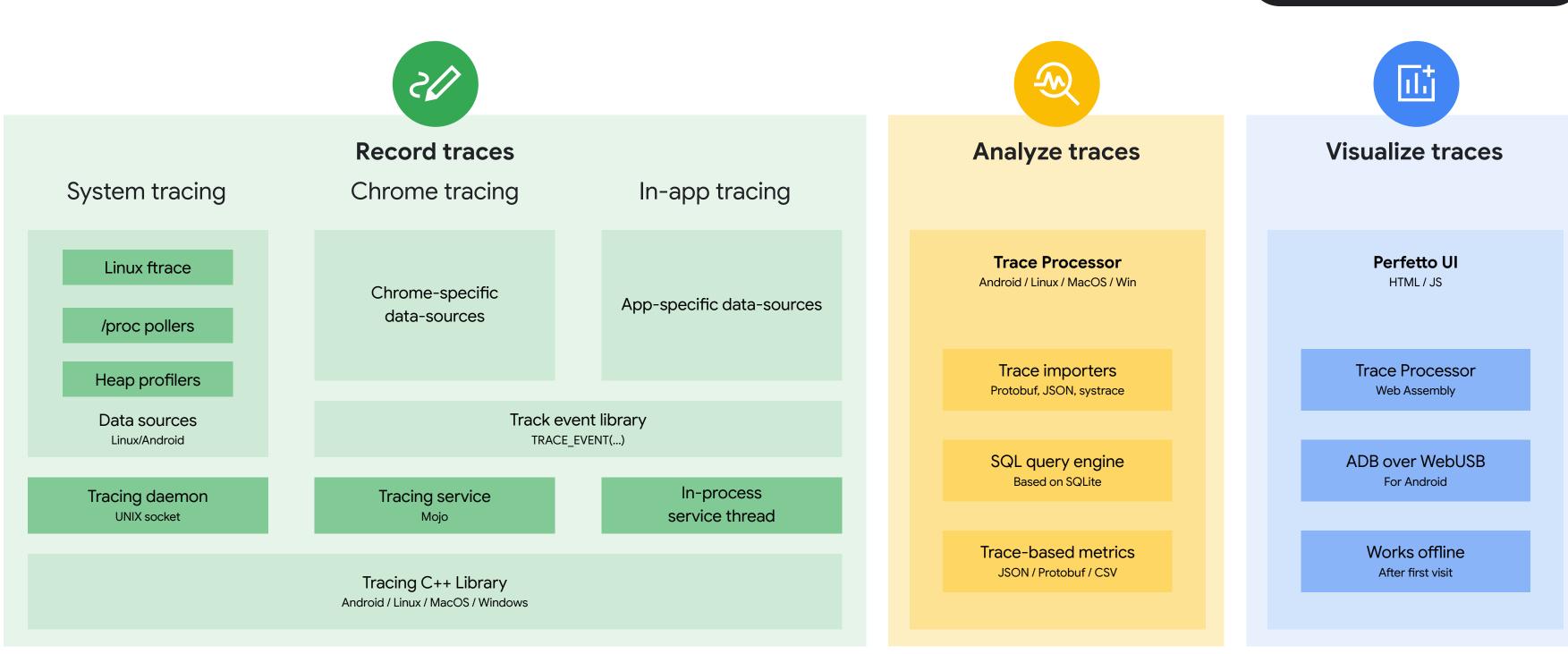
Perfetto provides a comprehensive trace viewer web UI that empowers one to inspect, visualize, and analyze the collected data.



Data Mining

One can leverage SQL-like syntax to query the trace data, making complex analysis easier.

How does Perfetto Work?

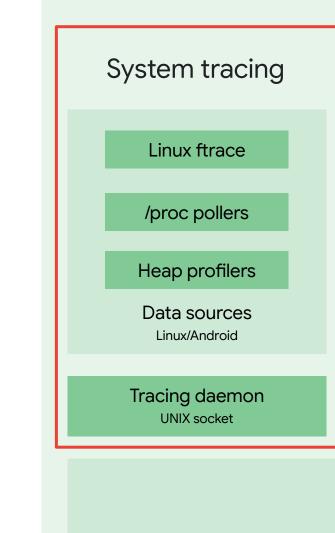


https://perfetto.dev/docs

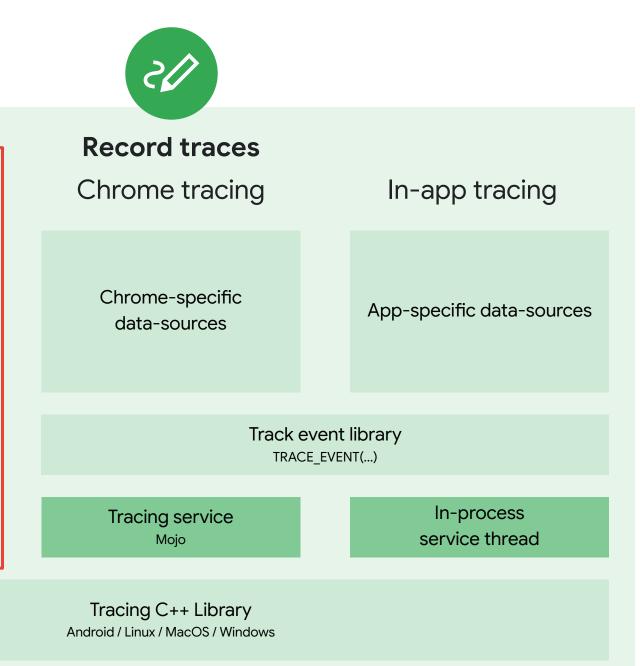
How does Perfetto Work?

System Wide Tracing for Android and Linux

- Kernel tracing is enabled via Linux ftrace, which allows kernel events such as scheduling events and syscalls to be recorded.
- /proc pollers allow the sampling of process-wide cpu and memory counters over a time period.
- Heap profilers also enable capturing information for the Native and Java heap.



<u>https://perfetto.dev/docs</u> \rightarrow

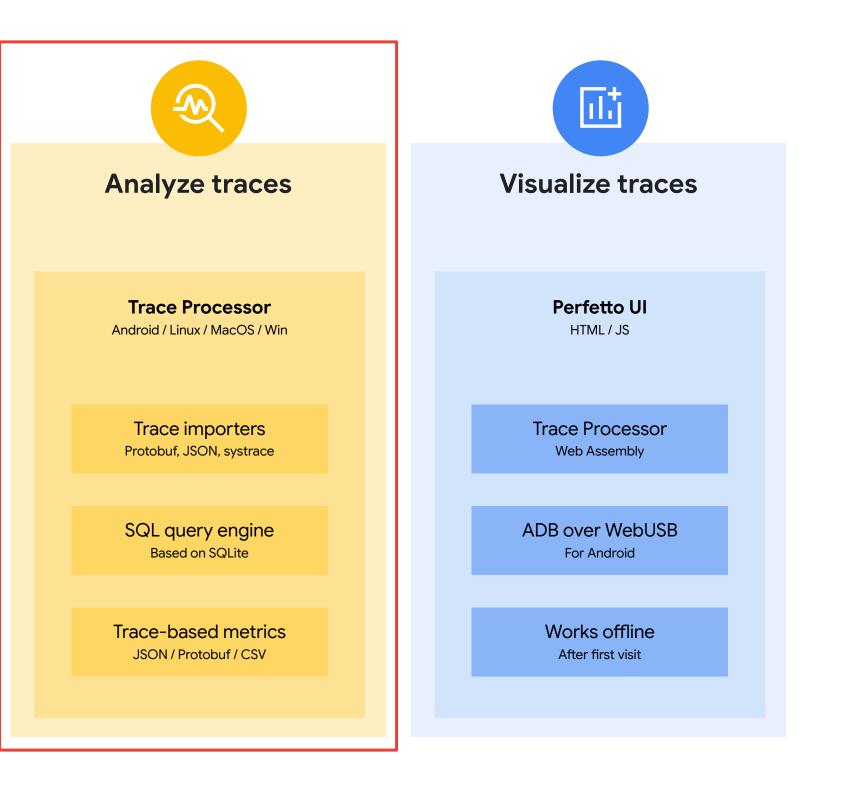


How does Perfetto Work?

Trace Analysis

- The Trace Processor is a C++ library that takes in raw trace data and surfaces it through an SQL interface for straight-forward querying.
- Trace importers allow simple ingestion of multiple formats
- Trace-based metrics creates pre-formatted and extensible queries that provide trace summaries. (e.g. CPU usage at different frequency states).

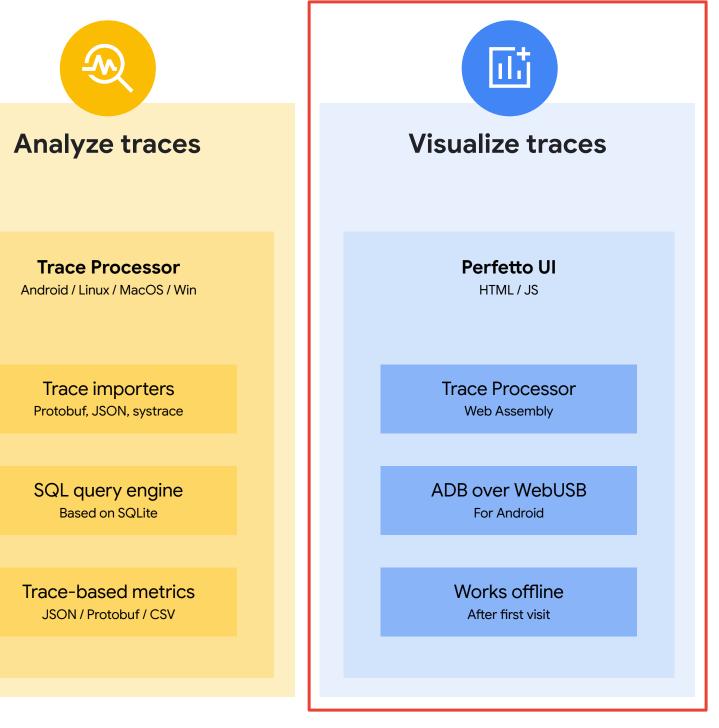
<u>https://perfetto.dev/docs</u> \rightarrow



How does Perfetto Work?

Trace Visualization

- A trace visualizer is instrumental for analysis and is powered by \bullet WebAssembly.
- The Perfetto UI works fully offline after initial opening.



<u>https://perfetto.dev/docs</u> \rightarrow

Google Automotive Partner Bootcamp

Getting Started with Perfetto



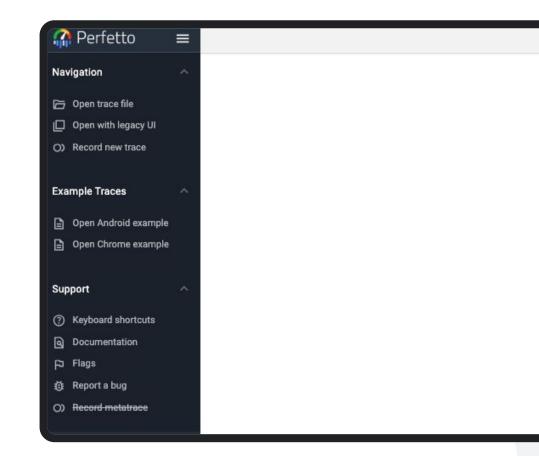
Quick Start: Collecting a Perfetto Trace

After defining an appropriate trace configuration, one can run the trace collection.

- 1. Download the recording script using the below command:
 - \$ curl -0 https://raw.githubusercontent.com/google/perfetto/master/tools/record_android_trace
 - \$ chmod u+x record_android_trace
- Start tracing using: 2.
 - \$./record_android_trace -o <trace-name>.trace -c <previous trace file>
- Run the desired CUJ or experiment 3.
- End the trace using Ctrl+C for the command run in Step 2 4.
- The trace will be automatically be opened in the browser after the collection has completed 5.

Quick Start: Viewing a Trace

If one wants to open an existing trace file, navigate to **ui.perfetto.dev** to open and access a trace:



Q Search or type '>' for commands or ':' for SQL mode

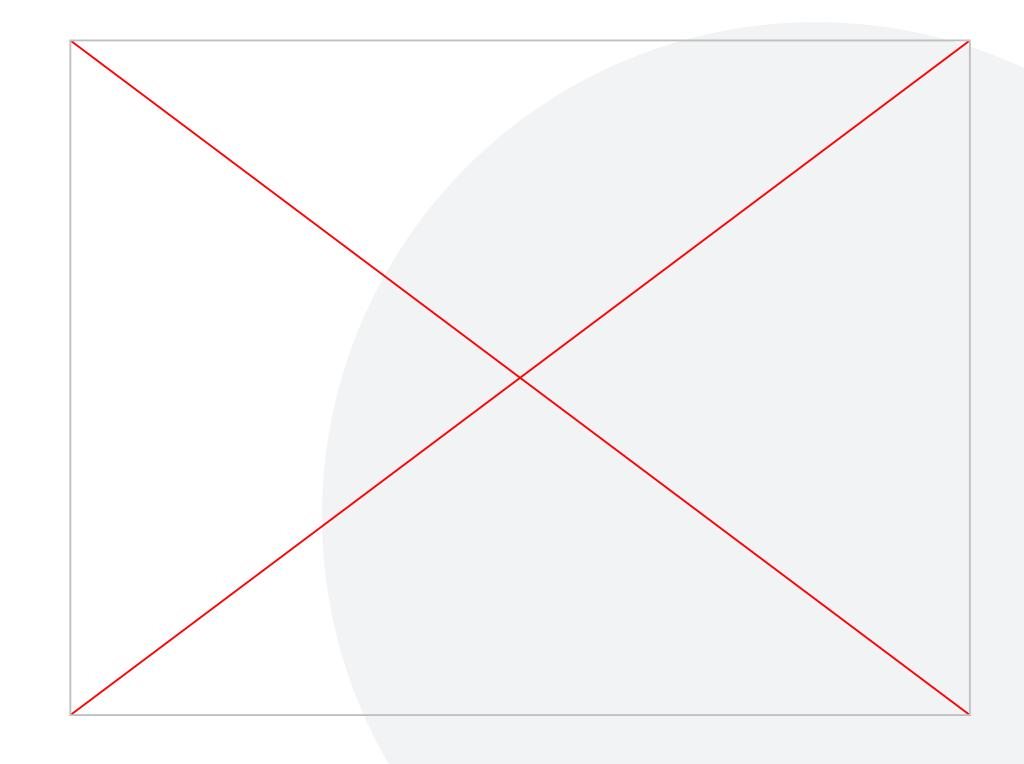
Quick Start: Viewing a Trace

Once the trace is generated, one can also generate a permalink to the existing trace that can be shared:

na Perfetto ≡					/pe '>' for command
Navigation ^		2000400.0	10.00 4000d00.00.00	6000400-00-00	, 100
🗁 Open trace file	UTC 1970-01-01	19755406:52:00	19755d06:52:02 000 000 000	19755d06:52:04 000 000 000	19755d06:52:06 000 000 000
Den with legacy UI	≎ =	0.00,000,000,0	0.00,000,000,000,000,000,000,000,000,00		
O) Record new trace	Cpu 0 (little)				
	Cpu 1 (little)				
Current Trace	Cpu 2 (little)				
boottrace.perfetto-trace (244 MB)	Cpu 3 (little)				
Show timeline	Cpu 4 (mid)				
	Cpu 5 (mid)				
< Share	Cpu 6 (mid)				
🛃 Download	Cpu 7 (big)				
曼 Query (SQL)	Cpu 0 Frequency	2.5 GHz			
🖾 Viz	Cpu 1 Frequency	2.5 GHz			
Metrics	Cpu 2 Frequency	2.5.GHz			
 Info and stats 	Cpu 3 Frequency	2.5 GHz			
() Into and stats	Cpu 4 Frequency	2.5 GHz	We I I I		diage of the
	Cpu 5 Frequency	2.5 GHz	04M THEFT		district of
Convert trace	Cpu 6 Frequency	2.5 GHz	4.14		44.75
	Cpu 7 Frequency	2.5 GHz		and the second second	THE PARTY ST
Switch to legacy UI	Android App Startups	android.	ca com.renault.ca	ar.launcher	
🛃 Convert to .json	Android logs	iter had well in a star	· · · · · · · · · · · · · · · · · · ·		and the state of the
🛃 Convert to .systrace	GPU Memory	A 200M 7			

Demo Video

An example video of a trace being collected from beginning to end.



Special Case: Collect Boot Time Tracing

In Android TM+, the trace can be collected as seen previously. However the following setting must be enabled before the device is restarted:

adb shell setprop persist.debug.perfetto.boottrace 1

In Android SC-, the following steps are required to setup the device:

- 1. Boot tracing in Android SC- requires selinux to be set to permissive.
- The following .rc file on the right must be created. 2.
- adb root && adb remount must be run to remount the device. 3.
- Use the following commands to push the .rc and config file to the device: 4.

adb push perfetto_boot.rc /etc/init/perfetto_boot.rc adb push perfetto_trace_config.textproto /data/misc/perfetto-traces/boottrace.pbtxt

class late_start disabled user shell group nobody oneshot stdio_to_kmsg

EOF

```
cat >> perfetto_boot.rc << 'EOF'</pre>
service perfetto_boot /system/bin/perfetto --txt -c
/data/misc/perfetto-traces/boottrace.pbtxt -o
/data/misc/perfetto-traces/boottrace.perfetto-trace
```

```
seclabel u:object_r:perfetto_exec:s0
capabilities DAC READ SEARCH
```

```
on property:persist.perfetto.boottrace=1
   rm /data/misc/perfetto-traces/boottrace.perfetto-trace
   start perfetto_boot
```

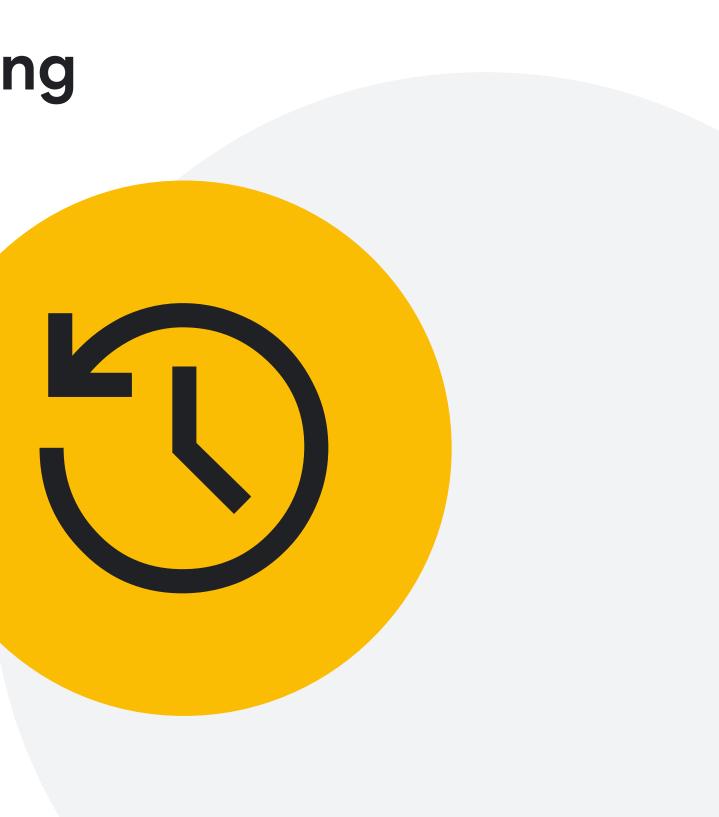
Special Case: Collect Boot Time Tracing

The following steps are required to collect the trace:

- 1. Reboot the device using adb reboot
- 2. Stop perfetto and pull the trace:

adb shell pkill perfetto

adb pull /data/misc/perfetto-traces/boottrace.perfetto-trace



Trace Anatomy

Google Automotive Partner Bootcamp



Trace Config Setup

Selecting the right trace config will allow one to collect the necessary data from the system.

- Perfetto provides granular control over data collection. Unlike always-on logging systems (e.g., Linux's rsyslog, Android's logcat), its tracing data sources start in an idle state.
- The TraceConfig is a protobul message that controls your Perfetto tracing session. It outlines:
- System-wide Settings:
 - Maximum trace duration.
 - Number and size of memory buffers.
 - Maximum output file size.
- Data Source Specifications:
 - For kernel tracing, which ftrace events to enable.
 - For the heap profiler, the target process name and sampling rate.
- Data Routing: Specifies which buffer each data source should write into

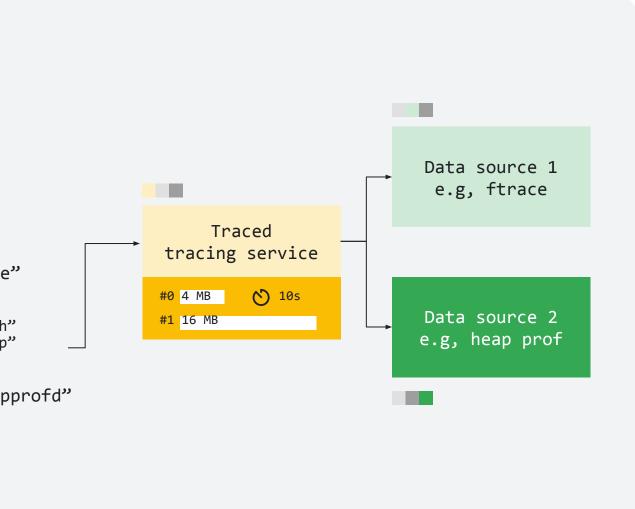
Note: a sample config can be found at perfetto.dev/docs/concepts/config



How the Tracing Service Uses the TraceConfig

- The tracing service (traced) is your config manager. When you start a tracing session, the service:
 - **Reads System Settings:** It determines its behavior based on the TraceConfig's outer section (duration, buffers, etc.).
 - Activates Data Sources: It finds Producers that match the data sources listed in the config. Then, it starts each Producer and provides the relevant DataSourceConfig settings.

Trace config
Duration: 10s
Buffers: #0/: 4MB #1/: 16MB
data source: "linux.ftrac
<pre>Ftrace_config { Ftrace_events: "sched_switch Ftrace_events: "sched_wakeup }</pre>
data source: "android.hea
<pre>heapprofd_config { sampling_interval_bytes: 1 process_cmdline: "adbd" Continuous_dump_config { dump_phase_ms: 10000 Dump_interval_ms: 10000 } }</pre>
-



Defining Buffers:

- This section defines the number, size and policy of in-memory buffers owned by the tracing service.
- Fill Policy:
 - A RING_BUFFER (default) fill policy will wrap over when full and replace the oldest trace data in the buffer.
 - A DISCARD fill policy will stop accepting data once full.

Dynamic Buffer Mapping:

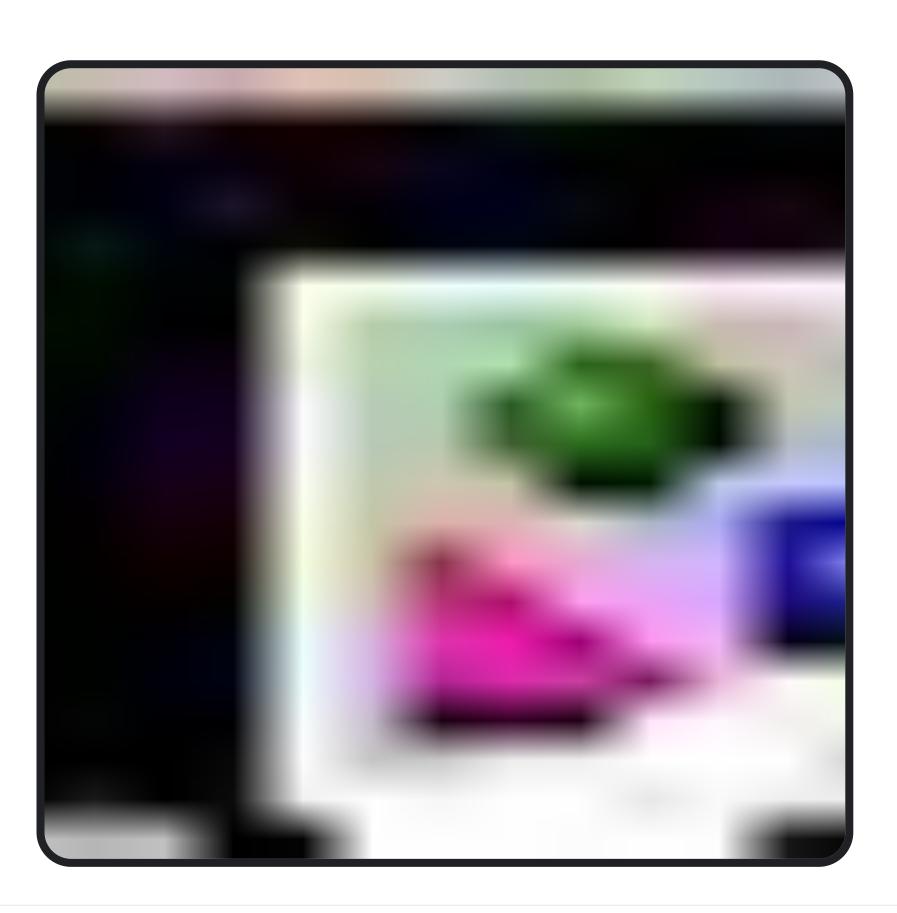
• The target_buffer field can be specified to indicate different buffers for data sources.

Defining several buffers buffers: { size_kb: 4096 fill_policy: RING_BUFFER buffers { size_kb: 4096 fill_policy: RING_BUFFER }

Defining Data Sources (Logcat)

This data source will enable Android logcat messages to be shown:

```
data_sources: {
   config {
      name: "android.log"
      android_log_config {
      }
   }
}
```



Defining Data Sources (CPU Frequency)

Various CPU frequency stats can be collected with the following data sources:

- Enabling the power/cpu_frequency ftrace event Ο
- Setting cpufreq_period_ms > 0 (Note: only works on 0 Android SC-V2 and above)

Cpu 0 Frequency	2.5 GHz
Cpu 1 Frequency	2.5 GHz
Cpu 2 Frequency	2.5 GHz
Cpu 3 Frequency	2.5 GHz
Cpu 4 Frequency	2.5 GHz
Cpu 5 Frequency	2.5 GHz
Cpu 6 Frequency	2.5 GHz
Cou 7 Frequency	2.5 GHz

data_sources: {

}

}

```
config {
   name: "linux.sys_stats"
   target_buffer: 1
   sys_stats_config {
        cpufreq_period_ms: 500
   }
data_sources: {
   config {
        name: "linux.ftrace"
        ftrace_config {
```

```
ftrace_events: "power/cpu_frequency"
ftrace_events: "power/cpu_idle"
ftrace_events: "power/suspend_resume"
```

}

Defining Data Sources (Jankiness)

Jankiness can be examined with the frame timeline data source.

}

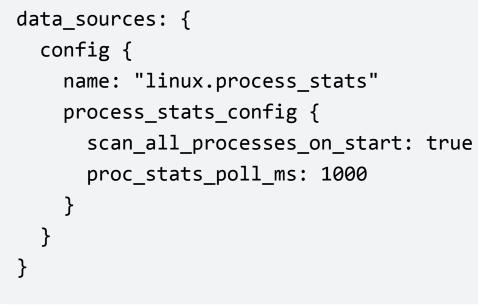
64.2 s +	1.9 s		+17.3 ms	+37.3 ms	+57.3 ms		+77.3 ms	+0	97.3 ms	+1	17.3 ms	+137.3	ms
Expected Timeline	*	4793	4794	4797	4801	4804	4806	4809	4812	4815	4818	4821	48
Actual Timeline	*		4793 4794	4797	4801	4804	4806	4809	4812	4815	4818	4821	

```
data_sources: {
  config {
    name: "android.surfaceflinger.frametimeline"
   target_buffer: 2
```



Defining Data Sources (linux.process_stats)

- The linux.process_stats data source gathers per-process statistics from the /proc/<pid>/status and /proc/<pid>/oom_score_adj files on Linux systems
 - Process memory usage (RSS, VMSize, etc.)
 - Open file descriptors Ο
 - Out-of-memory (OOM) score (indicates how likely the 0 kernel is to terminate the process when memory is low)



▲ system_server 1352	
mem.virt	10 G
mem.rss	0.5 G
mem.rss.anon	75 M
mem.rss.file	0.25 G
mem.rss.shmem	2.5 M
mem.swap	50 M
mem.locked	100 M
mem.rss.watermark	0.5 G
oom_score_adj	1 K

Defining Data Sources (linux.sys_stats)

- The linux.sys stats data source gathers a range of system-level statistics from Linux. The following stat counters can be collected:
 - Stat Counters (proc/stat):
 - STAT CPU TIMES
 - user: Time spent running in user mode
 - nice: Time spent running niced user processes
 - system: Time spent in system (kernel) mode
 - idle: Time the process was idle
 - Mem Info Counters (proc/meminfo): 0
 - Provides information such as free memory, anonymous memory.
 - VM Stat Counters (proc/vmstat):
 - Provides information on virtual memory such as page faults, pages in and out, etc.
 - Note: cpufreq_period_ms is only available above SC-V2. 0
 - The following error will be encountered otherwise:
 - No field named "cpufreq_period_ms" in proto SysStatsConfig.

data_sources: { config { name: "linux.sys_stats" target buffer: 1 sys_stats_config { stat_period_ms: 500 stat counters: STAT CPU TIMES

> meminfo period ms: 1000 meminfo_counters: MEMINFO_ACTIVE_ANON meminfo_counters: MEMINFO_ACTIVE_FILE meminfo_counters: MEMINFO_INACTIVE_ANON meminfo_counters: MEMINFO_INACTIVE_FILE meminfo counters: MEMINFO KERNEL STACK meminfo counters: MEMINFO MLOCKED meminfo counters: MEMINFO SHMEM meminfo counters: MEMINFO SLAB meminfo counters: MEMINFO SLAB UNRECLAIMABLE meminfo_counters: MEMINFO_VMALLOC_USED meminfo_counters: MEMINFO_MEM_FREE meminfo_counters: MEMINFO_SWAP_FREE

vmstat_period_ms: 1000 vmstat_counters: VMSTAT_PGFAULT vmstat counters: VMSTAT PGMAJFAULT vmstat counters: VMSTAT PGFREE vmstat counters: VMSTAT PGPGIN vmstat_counters: VMSTAT_PGPGOUT vmstat_counters: VMSTAT_PSWPIN vmstat_counters: VMSTAT_PSWPOUT vmstat_counters: VMSTAT_PGSCAN_DIRECT vmstat_counters: VMSTAT_PGSTEAL_DIRECT vmstat counters: VMSTAT PGSCAN KSWAPD vmstat counters: VMSTAT PGSTEAL KSWAPD vmstat counters: VMSTAT WORKINGSET REFAULT

Below field not available on < Android SC-V2 releases.</pre> cpufreq_period_ms: 500

}

Defining Data Sources (linux.sys_stats)

 Misc Global Tracks 		
ION allocations (heap: all) metric	~	0.75 G
mem.ion	~	0.75 G
MemAvailable	~	5 G
MemFree	~	5 G
MemTotal	~	7.5 G
nr_active_anon	~	0.5 M
nr_active_file	~	50 K
nr_file_pages	~	0.5 M
nr_free_pages	~	2.5 M
nr_inactive_anon	~	5 K
nr_inactive_file	~	0.5 M
nr_mapped	~	0.5 M
nr_zspages	~	5
num_forks	~	5 K
pgpgin	~	2.5 M
pgpgout	~	25 K
pswpin	~	0
pswpout	~	0

Defining Data Sources (ftrace)

Capturing ftrace events allows developers insights into kernel code. They are useful for analyzing latency or performance issues outside of userspace.

- Memory Events
- Low Memory Killer Events
- Sched Events

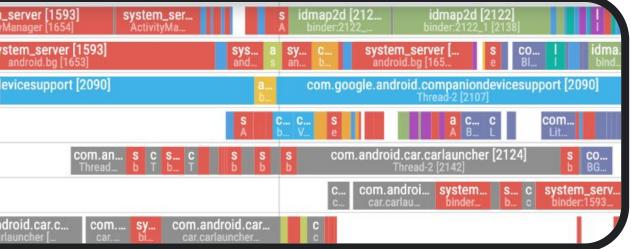
```
data_sources: {
  config {
    name: "linux.ftrace"
    target buffer: 2
    ftrace_config {
      # Memory events
      ftrace_events: "power/suspend_resume"
      ftrace_events: "mm_event/mm_event_record"
      ftrace_events: "kmem/rss_stat"
      ftrace_events: "ion/ion_stat"
      ftrace_events: "dmabuf_heap/dma_heap_stat"
      ftrace_events: "kmem/ion_heap_grow"
      ftrace_events: "kmem/ion_heap_shrink"
      # LMKD events
      ftrace_events: "lowmemorykiller/lowmemory_kill"
      ftrace_events: "oom/oom_score_adj_update"
      ftrace_events: "oom/mark_victim"
      # sched events
      ftrace_events: "sched/sched_process_exit"
      ftrace_events: "sched/sched_process_free"
      ftrace_events: "sched/sched_switch"
      ftrace_events: "sched/sched_wakeup"
      ftrace_events: "sched/sched_wakeup_new"
      ftrace_events: "sched/sched_waking"
   }
```

Defining Data Sources (ftrace)

In order to capture CPU scheduling events, *ftrace_events: "sched/sched_switch"* needs to be added to the *linux.ftrace* data source. With this enabled the following can be captured:

- Threads scheduled per CPU Ο
- Why a thread got de-scheduled (pre-emption, blocked by a mutex) 0
- When a thread becomes runnable 0

Cpu 0 (little)		ndroid.companiondevi c nread-2 [2107]	com se Thre se		S S	system_s ActivityM
Cpu 1 (little)	syste sys tr andro	system_server [1593] android.bg [1653]	CO bi	system_server [1593 android.bg [1653]	3]	syst
Cpu 2 (little)	traced_perf [2075] stack-unwinding [2076]	com c c c system_server [15 Lit B T B ActivityManager [idmap2d [21 binder:2122		com.google.android	.companiondev Thread-2 [2107]
Cpu 3 (little)	comcom.google.android.ca sodsoda_process [31944]	a Sys c c an Act L S bi	C C C b V S	system_server [159 ActivityManager [1	idmap2d [2 binder:212	
Cpu 4 (mid)	C S Sy B A Ac					
Cpu 5 (mid)	sy c tr t Ac B tr t					
Cpu 6 (mid)	com.a com.andro s co car.c	om.android.car s com.an s c car.carlauncher b car.ca b c	S b	com.an co car.ca	com S c	com.andi car.carla



Atrace Categories:

Predefined groups of trace events that make it easier to enable tracing for specific areas of the system.

Fine-grained Process Tracing:

The atrace_apps functionality in Perfetto enables selective tracing of specific applications on Android. It allows you to capture trace data only from the processes of interest.

 com.google.android.gms.persis 502 	stent 3				
.gms.persistent 3502	Uni R	Run. Ru.,	Runnable	e Uninter., Runna., Ru., R	unnable (Preempted) Ru., Run., R R Runnable (
.gms.persistent 3502	ActivityThreadMain binder transaction	ResourcesMa	nager#applyConfigurationToResourc	. binder. /system/frame	work/org.apache.http.lega/syst/system/
			er transaction binder transac	and so it is a second se	AteClassloaderNamespace Open OpenDes G L. GetBes I A IsUse S O. Stat Open o diopen:

```
data_sources: {
  config {
    name: "linux.ftrace"
    target buffer: 2
    ftrace config {
      # Memory events
      atrace_categories: "aidl"
      atrace categories: "am"
      atrace categories: "dalvik"
      atrace_categories: "binder_lock"
      atrace_categories: "binder_driver"
      atrace_categories: "disk"
      atrace categories: "freq"
      atrace categories: "idle"
      atrace_categories: "gfx"
      atrace categories: "hal"
      atrace categories: "pm"
      atrace categories: "power"
      atrace_categories: "rro"
      # atrace apps
      atrace_apps: "lmkd"
      atrace_apps: "system_server"
      atrace apps: "com.android.systemui"
      atrace_apps: "com.google.android.gms"
      atrace_apps: "com.google.android.gms.persistent"
      atrace apps: "android:ui"
      atrace_apps: "com.google.android.apps.maps"
```

Writing to a Trace Output File:

If not recording time limit is specified, one will have to manually terminate the tracing session.

If duration_ms is specified then, the trace will terminate automatically.

If write_into_file is true, then Perfetto will periodically stream results into a trace file.

Flush_period_ms defines the default drain period. A shorter period means a smaller userspace buffer is required. However, this will increase the performance intrusiveness of tracing.

Max_file_size_bytes is used to cap the size of a trace file.

Flush_period_ms is used to periodically issue a Flush() to all data sources, forcing them to commit their data into the tracing service.

No recording time limit (press CTRL+C to stop recording). # Alternatively: uncomment the line below to set a time limit. #duration ms: 1800000 write_into_file: true file_write_period_ms: 5000 max_file_size_bytes: 10000000000 flush period ms: 5000

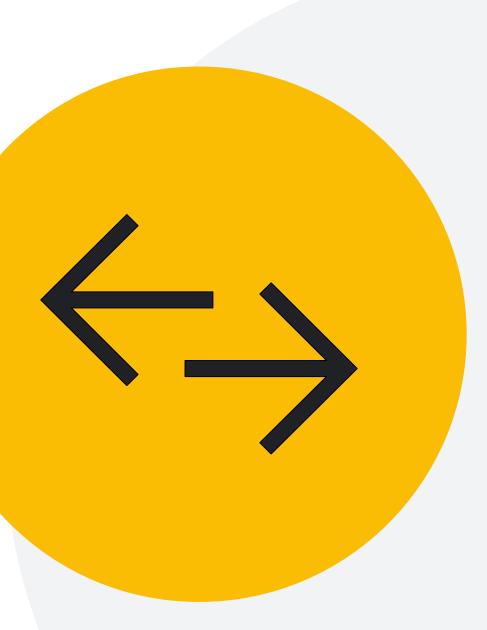
Anatomy of a Trace: Binder Transactions

There are two types of binder transactions:

- 1. Unidirectional: Using the **oneway** keyword in the AIDL language, these transactions do not wait for a reply after sending a parcel.
- Bidirectional: The transmitting end is blocked until it receives a reply. 2.

Note: This is only available in UDC onwards.





Anatomy of a Trace: Bidirectional Transactions

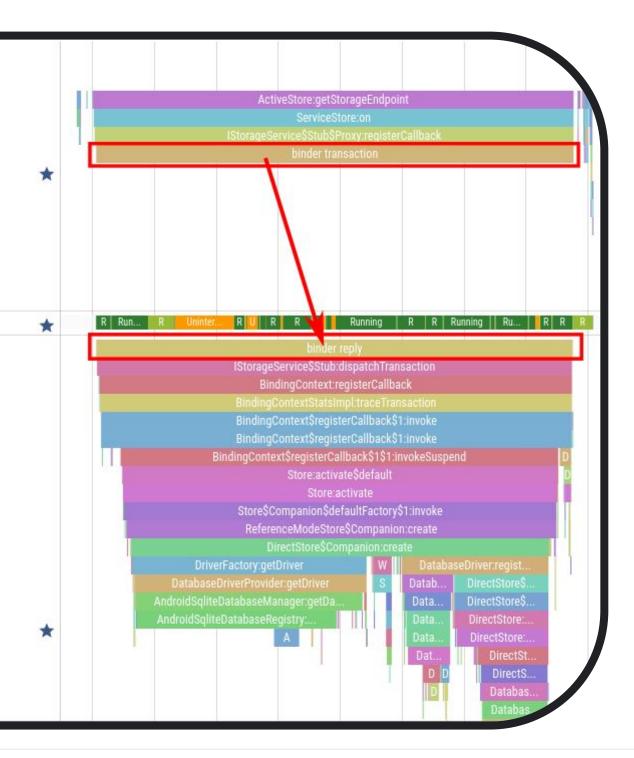
Bidirectional Transaction:

Identified by a corresponding binder transaction and binder reply pair.

pool-4-thread-1 17764

Binder:17769_3 17788

Binder:17769_3 17788



Anatomy of a Trace: Unidirectional Transactions

Unidirectional Transaction:

Indicated by an arrow in a Perfetto trace.

efaultDispatch 17799
Current Selection
Current Selection
Slice Details
Slice Details Name
Slice Details Name Category

	Runnable (Preempted)	Running
	idle\$1\$1:invokeSuspend	Bi
IResultCallback	<\$Stub\$Proxy:onResult	
A		

binder transaction async binder 3s 377ms 694us Os Ox11 his is a one-way call: async, no return; allow replies with file descript

Common Pitfalls

Google Automotive Partner Bootcamp



Common Pitfalls

External Process Interference

One of the reasons that a trace may be empty is that another process is using ftrace. Run either of the following to set the current_tracer to nop: Run the following command to determine if the current_tracer is nop: > adb shell echo "nop > /sys/kernel/tracing/current_tracer" > adb shell cat /sys/kernel/tracing/current_tracer > adb shell echo "0 > /sys/kernel/tracing/tracing_on" > adb shell cat /sys/kernel/debug/tracing/current_tracer # older kernel may still use debugfs

4000-00-00-0 19747d09:07:03 73 976 421 19747d09:07:14 000 006 000 19747d09:07:16 000 000 000 19747d09:07:18 000 000 000 19747d09:07:20 000 000 000 19747d09:07:22 000 000 000 19747d09:07:12 000 000 000 19747d09:07:24 000 000 000 Ξ NAMES OF TAXABLE PARTY. Android logs BRIE LE L' LE MIT TEL 10.000 1 BILL IN 1 () () **() () ()** ----- Chrome Scroll Jank Chrome Scrolls Chrome Scroll Janks Chrome Scroll Input Latencies Misc Global Tracks Kernel threads surfaceflinger 945 Process 1494 Process 4026 Process 2130 Process 4160

12000400-00-00	140	00400:00:00	1 6000400:00:00	
19747d09:07:26 000 000 000	19747d09:07:28 000 000 000	19747d09:07:30 000 000 000	19747d09:07:32 000 000 000	
		_		

Common Pitfalls

Insufficient Buffer Size

Another common pitfall is insufficient data buffer size. Increasing buffer size may alleviate scenarios in which key CUJ data is dropped.

Excessive event collection

If too many events are being collected, there are some that can be dropped to avoid trampling the output trace file. Including sys_enter and sys_exit will lead to all system calls being logged. The below trace demonstrates this, where the tracks do not terminate. data_sources: {
 config {
 name: "linux.ftrace"
 target_buffer: 2
 ftrace_config {
 # Do not include the below:
 ftrace_events: "raw_syscalls/sys_enter"
 ftrace_events: "raw_syscalls/sys_exit"

S	S	S			S		S
				J	sys_epoll_pwait		
			1		Ex	it (Dead)	
				sys_nanosleep	S	ys_exit	
							1
							s
							I.
		Image: Second	Image: second		Image: second	sys_epoll_pwait	sys_epoll_pwait

Trace Analysis

Google Automotive Partner Bootcamp



Trace Analysis Key Steps

Summary:

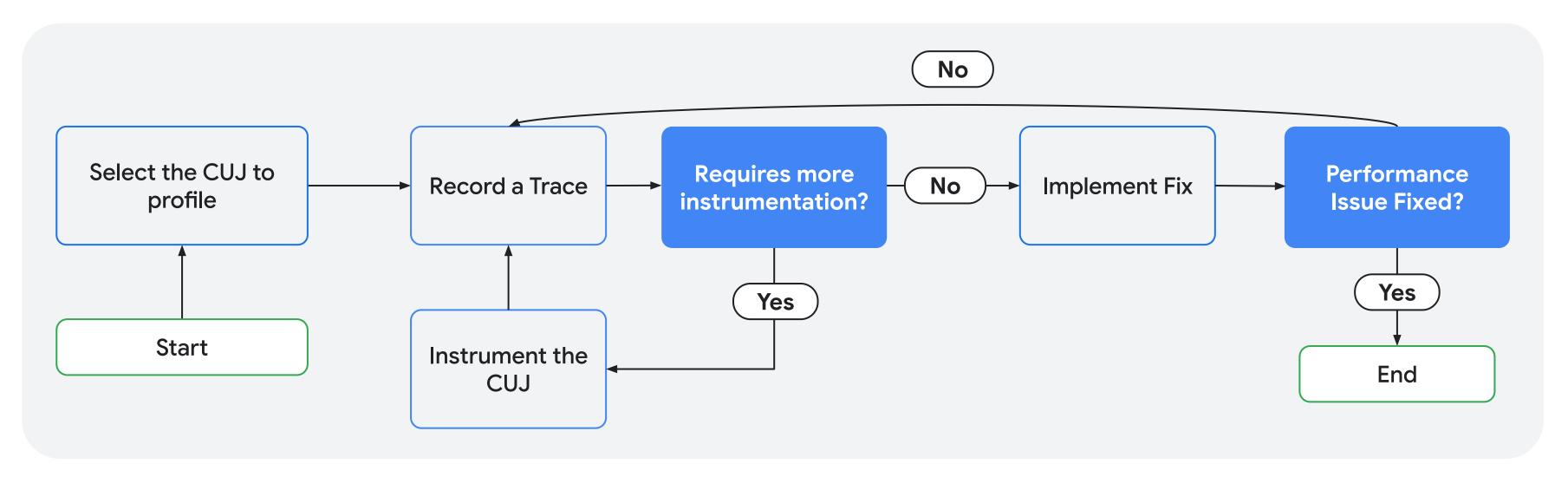
- 1. Narrow the search space: one can achieve this by determining the beginning and ending points via Android system logs or atrace logs.
- 2. Inspect CPU, memory tracks, etc: This will help identify symptoms of a regression so that the analysis window can be tightened.
- 3. Understand context: After capturing a smaller window, it is possible to understand what actions are being performed. (Ex. What is occurring during at this point in the user switch lifecycle?)
- 4. Identify Culprit Process: Given context, it is possible to visualize offending processes in the trace. Adding more logging via atrace will also allow one to trace points in a codepath.
- 5. Analyze thread-level interactions: Looking at markers such as thread state and binder transactions during the window will allow one to make informed hypotheses.



Trace Analysis Summary

Sample flow illustrating:

- 1. CUJ profiling
- 2. Trace recording
- 3. CUJ instrumenting
- 4. Performance Fixes



Looking at a trace can be overwhelming. There are several key steps to help narrow down a problem area to root cause a performance issue. A trace analysis walkthrough will help guide investigations. Initially a trace was collected that captured a user switch from user 10 to user 11.

Narrow the search space: Use Android logs to identify key starting and stopping points. In this case flag UserController.startUser-11-fg-start-mode-1 and 1. onCompletedEventUser 11 act as the stop and start points.

UTC 1970-02-18		19490d21:13:06 000 000 000	19490d21:13:08 000 000 000	19490d21:13:10 000 000 000	19490d21:13:12 000 000 000	19490d21:13:14 000 000 000
≎		User 11 Start				F
Cpu 0 (little)						
Cpu 1 (little)						
Cpu 2 (little)						
Cpu 3 (little)						
Cpu 4 (mid)						
Cpu 5 (mid)						
Cpu 6 (mid)						
Cpu 7 (big)						
Cpu 0 Frequency	2.5 GHz				1 1 1	
Cpu 1 Frequency				A THE REAL PLACE TO THE		
Cpu 2 Frequency	2.5 GHz					
Cpu 3 Frequency	2.5 GHz				1 1 1 11	
Cpu 4 Frequency	2.5 GHz [] [] [] [] [] []					
Cpu 5 Frequency	2.5 GHz				TT T	
Cpu 6 Frequency	2.5 GHz			ULFRENT IL LELL IN ULFRENT I BENE		
Cpu 7 Frequency	2.5 GHz			an an an an that that the second s		Linite of Sector 1
Current Selection Android	Logs					
Android Logs [0, 0] / 2					Log Level Verbose 🗸 Filte	r by tag User-11-fg-start
Timestamp Level	Tag	Process name	Message	9		
19490d21:13:06.2 D	SystemServerTiming	system_server	UserCont	roller.startUser-11-fg-start-mod	e-1	
19490d21:13:06.5 V	SystemServerTiming	system_server	UserCont	roller.startUser-11-fg-start-mod	e-1 took to complete: 367ms	



2. Inspect CPU and memory tracks: In this case, it is apparent that there are big and gold cores being underutilized during the user switch.

UTC 1970-02-18	19490d21:13:06 000 000 000	19490d21:13:08 000 000 000	19490d21:13:10 000 000 000	19490d21:13:12 000 000 000
\$ <i>=</i>	User 11 Sta	n Big a	nd Gold Cores Idle	
Cpu 0 (little)				
Cpu 1 (little)				
Cpu 2 (little)				
Cpu 3 (little)				
Cpu 4 (mid)				
Cpu 5 (mid)				
Cpu 6 (mid)				
Cpu 7 (big)				
Cpu 0 Frequency	2.5 GHz			
Cpu 1 Frequency	2.5 GHz		IN REPORTED AND IN THE PARTY OF THE	T T T
Cpu 2 Frequency	2.5 GHz			1 I II
Cpu 3 Frequency	2.5 GHz			1 1 1 11
Cpu 4 Frequency	12.5 (GHz))())())())() () () () () () () () () () () () () (
Cpu 5 Frequency	2.5 GHz			1.1. 1. 1.
Cpu 6 Frequency	2.5 GHz			TT T T
Cpu 7 Frequency	2.5 GHz			
ODUMemory	0.25.6			
Current Selection Android Logs				
Android Logs [0, 0] / 2				Log Level Verbose 🗸 Filter b
Timestamp Level Tag	Process name	Messa	ge	
<u>19490d21:13:06.2</u> D SystemSer	verTiming system_server	UserCo	ntroller.startUser-11-fg-start-mode-1	
19490d21:13:06.5 V SystemSer	verTiming system_server	UserCo	ntroller.startUser-11-fg-start-mode-1	took to complete: 367ms



After identifying the area of interest, it is necessary to zoom in on events occurring during this underutilization. This can be achieved by zeroing in on log events or processes that have significant activity during that period.

UTC 1970-02-18	19490d21:13:06 000 000 000	19490d21: 000 000 00	13:08 0	19490d21:13:10 000 000 000	19490d21:13:12 000 000 000
\$ <i>=</i>	P Us	ser 11 Start			
Cpu 0 (little)					
Cpu 1 (little)					
Cpu 2 (little)					
Cpu 3 (little)					
Cpu 4 (mid)					
Cpu 5 (mid)					
Cpu 6 (mid)					
Cpu 7 (big)					
Cpu 0 Frequency				AND IL TANK I TO	T
Cpu 1 Frequency	2.5.047	T MARKAN AN AND AN		NUTU JUNETE TET T	1 11
Cpu 2 Frequency	2.5 GHz			NAJALJANY KATONA TA	1. (FU)
Cpu 3 Frequency	2.5 GHz			Marifel Judie a con a c	1 10
Cpu 4 Frequency					
Cpu 5 Frequency	2.5 GHz				1 11
Cpu 6 Frequency	2.5 GHz				I II
Cpu 7 Frequency	2.5 GHz				T II
ODUMemory	0.25.6				
Current Selection Android Logs	Are	ea of Interest			
Android Logs [0, 0] / 2				Log	Level Verbose ~ Filter by ta
Timestamp Level Tag	Process na	ame	Message		
<u>19490d21:13:06.2</u> D SystemSe	rverTiming system_serv	er	UserController.star	tUser-11-fg-start-mode-1	
19490d21:13:06.5 V SystemSe	rverTiming system_serv	er	UserController.start	tUser-11-fg-start-mode-1 took to o	complete: 367ms



3. Understand Context: Here, only 4 out of 8 cores are being utilized, which undoubtedly contribute to a prolonged user switch. This idleness appears early on in the user switch when user 11 is being started. SystemServiceManager is responsible for starting system services during user initialization. SystemServiceManager will wait until all services are created. It is clear that com.android.role.RoleService is the last service to be initialized and also requires the most time.

JTC 1970-02-18			19490d21:13:05 000 000 000	19490d21:13:06 000 000 000	19490d21:13:07 000 000 000	19490d2 000 000 0	1:13:08 000	19490d21:13:09 000 000 000	19490d21:13:10 000 000 000	19490d21:13:11 000 000 000	19490d21:13:12 000 000 000	19490d21:13:13 000 000 000	19490d21:13: 000 000 000	14
\$ <i>=</i>					User 11 Start		RoleServic	ce Done						
Cpu 5 (mid)														
Cpu 6 (mid)														
Cpu 7 (big)														
Cpu 0 Frequency			2.5 GHz							1.0	1 11			
Cpu 1 Frequency			2.5 GHz						LINUT INTEL	1 1				, n
Cpu 2 Frequency			2.5 GHz							1 1	1 UU			
Cpu 3 Frequency			2.5 GHz . 10/10/10/10/10/10/10/10/10/10/10/10/10/1						and and the second	1 I	1.1.1			r n
Cpu 4 Frequency			2:5 GHz]]][] [] [] [] [] [] []							1	1 11	ļ.	n n n	
Cpu 5 Frequency			2.5 GHz							1.1	11 1			
Cpu 6 Frequency			2.5 GHz		I FINDE MET MUNCH. III IN M					1	1 11			1
Cpu 7 Frequency			2.5 GHz							1 1	1 11			
GPU Memory		~	0.25 G											
Cpu 0 cpu-clock		~	25 G											
Cpu 1 cpu-clock		\sim	25 G											
Cpu 2 cpu-clock		\sim	25 G											
Cpu 3 cpu-clock		~	25 G											
Cpu 4 cpu-clock		~	25 G											
Current Selection	Android Log	gs												↑ ~
Android Logs [0,	0]/15							Log Leve	l Verbose 🗸 Sy	stemServiceManager	× Filter by tag	Sear	ch logs	×
Timestamp	Level	Tag		Process	name		Message							
19490d21:13:06.5	I	SystemS	erviceManager	system_se	rver		Calling o	nStartUser 11						
19490d21:13:07.0	W	SystemS	erviceManager	system_set	rver		Service c	om.android.server.or	n.OverlayManagerSe	rvice took 506 ms in	onStartUser-11			
19490d21:13:07.2	W	SystemS	erviceManager	system_set	rver		Service c	om.android.server.St	torageManagerServio	ce\$Lifecycle took 11	7 ms in onStartUser	-11		
19490d21:13:07.8	W	SystemS	erviceManager	system_set	rver		Service c	om.android.server.po	olicy.PermissionPol	licyService took 644	ms in onStartUser-	11		
19490d21:13:08.3	W	SystemS	erviceManager	system_set	rver		Service c	om.android.role.Role	Service took 1298	ms in onStartUser-1				
19490d21:13:08.4	I	SystemS	erviceManager	system_set	rver		Calling o	nSwitchUser 11 (from	n 10)					
19490d21:13:10.2	I	SystemS	erviceManager	system_set	rver			nUnlockingUser 11						
19490d21:13:10.4	I	SystemS	erviceManager	system_set	rver		Calling o	nUnlockedUser 11						
9490d21:13:10.6	I	SystemS	erviceManager	system_se	rver		Calling o	nStopUser 10						

com.android.role.RoleService onUserStarting method returns

4. Identify Culprit Process: Android logs allow us to identify that the com.google.android.permissioncontroller process is largely responsible for starting the RoleService. Zooming in further, it is apparent that the RoleControllerService thread handles majority of the initialization.



5. Look for thread level interactions: Inspecting the com.google.android.permissioncontroller process and its threads may reveal further details about thread state. For example, a long period of uninterruptible sleep could indicate heavy I/O usage. In this case there is nothing that indicates anything out of the ordinary.

ssioncontroller 24864	Running	Running	Running	Running	Ru	Running	Runn. R. Runna	R. R. Runn. R R R.	
ssioncontroller 24864				all.		din .		bindAp	plication
	4		/apex/com	n.android.permissi	on/priv-app/Go	ooglePerm	issionController@MASTER/	/GooglePermissionController.apk	
			OpenDexFilesFromOat(/apex/com	.android.permissio	on/priv-app/Go	oglePermi	ssionController@MASTER/	GooglePermissionController.apk)	
			Open dex file /apex/com.and	roid.permission/p	riv-app/Google	Permissio	nController@MASTER/Goo	glePermissionController.apk	
	Ĩ.		Verify dex file /apex/com.and	roid.permission/p	riv-app/Google	Permissio	nController@MASTER/Goo	glePermissionController.apk	
			and the second second second second second second second	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1				a provide the second second second	



5. Analyzing the RoleControllerService thread reveals that there is an excess of costly binder transactions occurring. It is clear that these inter-process communication transactions are the cause of the slow down.

c 1970-02-18		19490d21:13 500 000 000	3:06	19490d21:13:07 000 000 000	19490d21:13:07 500 000 000		19490d21:13:08 000 000 000		19490d21:13:08 500 000 000	19490d21:13:09 000 000 000	19490d21:15 500 000 000
\$ <i>=</i>			User 11 Start				[Ro	eService Done		
1000 1				ssm.c	onStartUser-11_0	com.android.role.RoleSe	17				
pool-992-thread 24857											
ssioncontroller 24864	Ŧ			bindApp /apex/com OpenDexF Verify d	Dication	11					
RoleControllerS 24887											
✓ Kernel threads											
✓ system_server 1303											
 com.google.android.apps.a ve.templates.host 25028 	automoti										an a
✓ traced_probes 1121	-					947 - TR T	en en ser		a second s		
Current Selection Android Log	gs Flow Events										<u></u> ↑ ~
Slice binder transaction										Contextu	al Options 👻
Details						Following Flows					
CategorybinderStart time19490d21:Absolute Time2023-06-30	Name binder transaction Category binder Start time 19490d21:13:07.915442744					✓ Flow Slice binder reply ↗ Delay 0s Thread binder:1303_14 3899 (system_server 1303)					
✓ Duration <u>354us 115r</u> Runnable (Preempted)		%)				Arguments					
	73us 386ns (20.72%						24887				
	ollerS [24887]					calling tid - code -	24887 0x08 Java Layer Depen	dent			
	e.android.permission	controller [2486	4]			data size -	292	aunt			
SQL ID slice[18934	4] -					destination name -					
						a loge of the second second					

Advanced Topics: SQL Queries

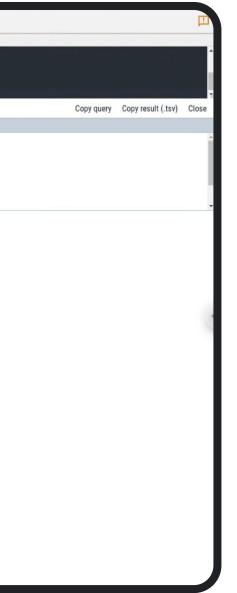


Data Mining Using SQL Queries

Beyond visually inspecting system issues via the Perfetto UI, it is also possible to gain a deeper understanding through SQL queries.

One can access SQL queries via the below interface:

n Perfetto		Q Search or type '>' for commands or ':' for SQL mode
Mandanadara	Enter query and press Cmd/Ctrl + Enter	
Navigation	7 slice.id as slice_id, utid, 8 thread.name as thread.name	
🗁 Open trace file	9 FROM slice	
🔲 Open with legacy Ul	<pre>10 JOIN thread_track ON thread_track.id = slice.track_id 11 JOIN thread USING (utid);</pre>	
O) Record new trace	Query result (0 rows) - 0.8ms DROP VIEW IF EXISTS slice_with_utid; CREATE VIEW slice_with_utid AS SELECT ts, dur, slice.name as slice_name,	slice.id as slice_id, utid, thread.name as thread_name FROM slice JOIN thread_track ON thread_track.id = slice.track_id JOIN thread USING (utid);
	Query history (1 queries)	
Current Trace	DROP VIEW IF EXISTS slice_with_utid; CREATE VIEW slice_with_utid AS	
sample.trace (28 MB)	SELECT	
🚃 Show timeline	ts, dur,	
< Share	slice.name as slice_name, slice.id as slice_id, utid,	
🛃 Download	thread.name as thread_name	
SOLO Ouery (SOL)		
🖄 Viz		
 Metrics 		
 Info and stats 		
Convert trace		
D Switch to legacy UI		
🚽 Convert to json		
Convert to systrace		
Example Traces		
Open Android examp		
Open Chrome example		
Support		
Copport		
⑦ Keyboard shortcuts		
Documentation		
P Flags		
👸 Report a bug		
O) Record metatrace		



Data Mining Using SQL Queries

One common example is collecting the CPU Time for slices. The first step is to build a table that links slices with their thread state.

```
DROP VIEW IF EXISTS slice_with_utid;
CREATE VIEW slice_with_utid AS
SELECT
 ts,
  dur,
  slice.name as slice_name,
  slice.id as slice_id, utid,
 thread.name as thread_name
FROM slice
JOIN thread_track ON thread_track.id = slice.track_id
JOIN thread USING (utid);
DROP TABLE IF EXISTS slice_thread_state_breakdown;
CREATE VIRTUAL TABLE slice_thread_state_breakdown
USING SPAN LEFT JOIN(
  slice_with_utid PARTITIONED utid,
 thread_state PARTITIONED utid
);
```

Data Mining Using SQL Queries

From the previous table, the CPU time for each slide in a Running state can be listed.

SELECT slice_id, slice_name, SUM(dur) AS cpu_time
FROM slice_thread_state_breakdown
WHERE state = 'Running'
GROUP BY slice_id;

nter query and press Cmd/Ctrl + Enter		
SELECT slice_id, slice_name, SUM(dur) AS cpu_time FROM slice_thread_state_breakdown WHERE state = 'Running'		
GROUP BY slice_id;		
uery result (10000 rows) - 686.1ms SELECT slice_id, slice_name, SUM(dur) AS cpu_time FROM slice_thread_stat	te_breakdown WHERE state = 'Running' GROUP BY slice_id;	Copy query Copy result (.tsv) Close
slice_id	slice_name	cpu_time
Contending for pthread mutex		28594
2 sys_epoll_pwait		11146
6 sys_ioctl		17916
7 sys_read		4479
		00000
5 sys_ioctl		39323
5 sys_ioctl 6 binder transaction async		39323

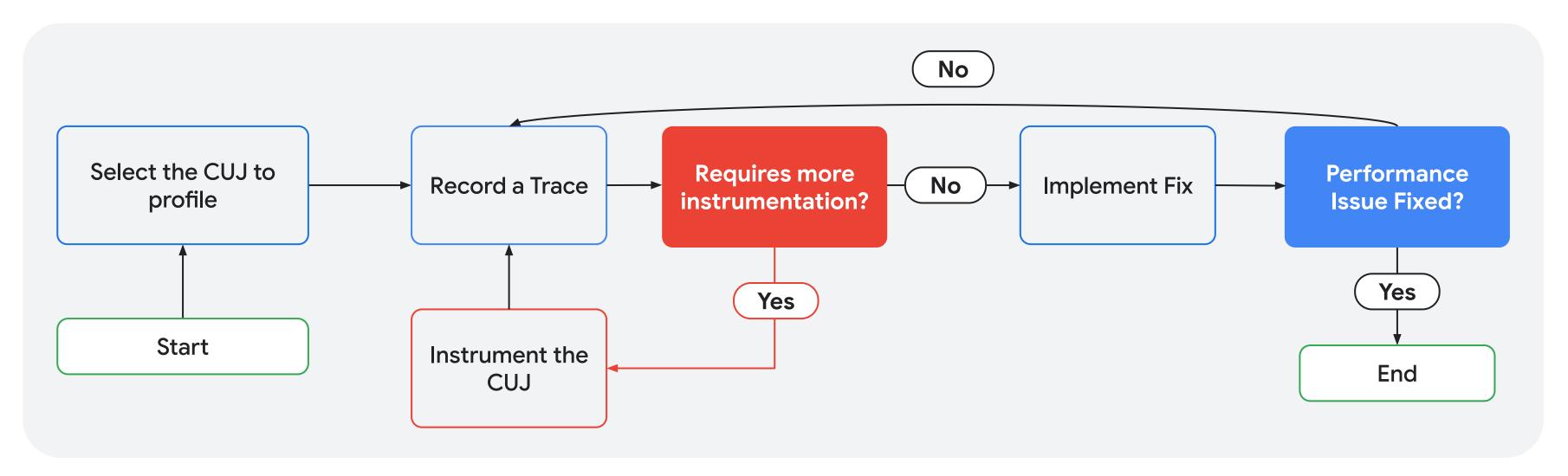
Making Debugging Easier



Make App Debugging Easier

Performance Analysis Flow:

- 1. CUJ profiling
- 2. Trace recording
- 3. CUJ instrumenting
- 4. Performance Fixes



Make App Debugging Easier

A powerful feature that can help with debugging is adding atrace logs that will appear in Perfetto.

Java applications can add trace logs using android.os.Trace.

Native applications can add trace logs using ATrace_beginSection() / ATrace_setCounter() defined in <trace.h>

Trace.traceBegin(TRACE_TAG, "Class#method");

• • •

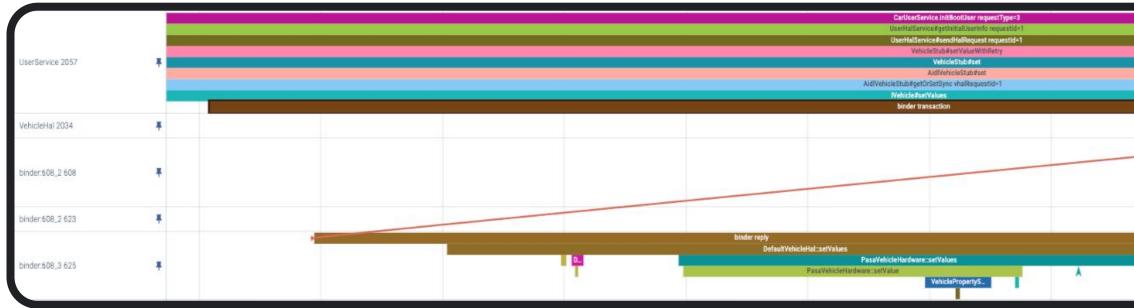
Trace.traceEnd(TRACE_TAG);

Make App Debugging Easier

Before:

UserService 2047	Loom/	VebicleStub#set AidTvebicleStub#getD0:SetSync binder transaction
VehicleHal 2028		
binder:605_2 606		
binder:606_3 626		binder reply
	ariset bioteline	

After:



		Ai_

Performance Tuning

Google Automotive Partner Bootcamp



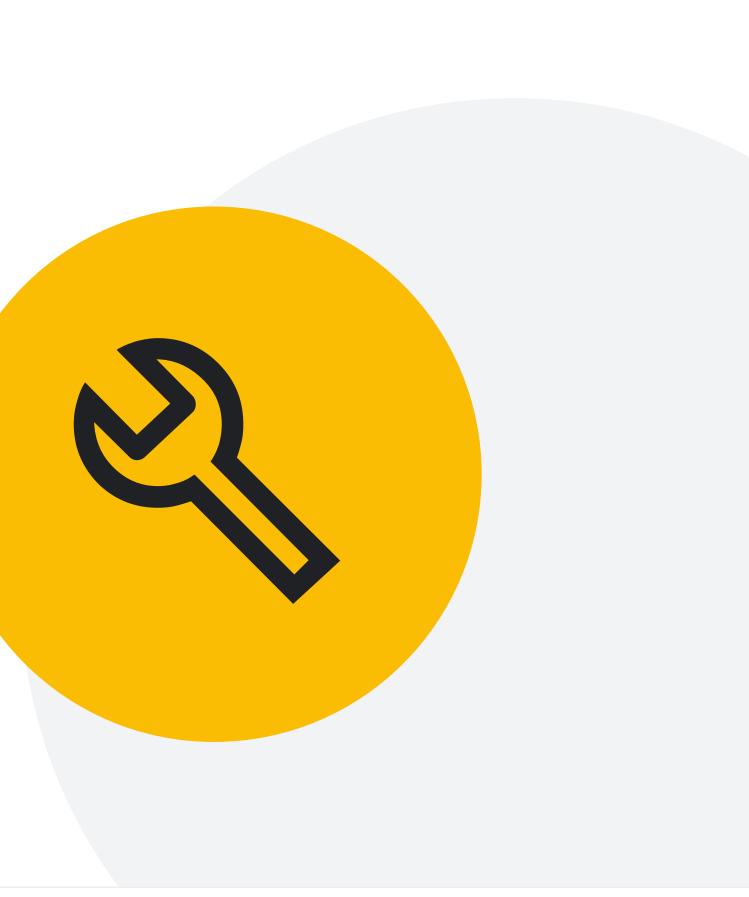
Performance Tuning After Boot

What is Performance Tuning?

With the aid of Perfetto, it is possible to identify performance issues and analyze their root cause. The next step is to implement solutions to solve these issues. One of the ways to achieve this is by iteratively tuning the performance of the system.

Post Boot Tuning

One of the opportunities for performance tuning is during post boot. After boot complete, there is heavy resource contention as multiple applications attempt to perform initialization. Classically, this is known as the Thundering Herd Problem, where lack of system resources leads to degraded performance. There is opportunity to both improve memory and CPU usage during the critical window after boot complete.



Memory Tuning

Google Automotive Partner Bootcamp



Kernel kswapd

•	kswapd is a Kernel task to manage available free memory.	
•	Kernel uses 3 watermarks per memory zone to track pressure	100
	 Min, Low, and High 	100
•	When free memory <= Low and > Min	
	 kswapd performs asynchronous/indirect memory reclaim. 	75
•	When free memory <= Min	
	 kswapd performs synchronous/direct memory reclaim. 	
	 System becomes unstable 	50
•	When free memory >= High	
	 kswapd temporarily enters sleep state. 	25
	 Periodically checks the memory pressure. 	
Re	claim types	
•	Indirect memory reclaim	0
	 Increases kswapd CPU usage. 	
	 May slow down other processes depending on the CPU & memory pressure. 	Total memory Low
•	Direct memory reclaim	High Min
	 All new allocations will be blocked until kswapd frees up memory up to min watermark. 	

Google Automotive Partner Bootcamp

Kswapd watermark levels

Sleep & Check	
 High Watermark - Indirect claim	
 Low Watermark - Aggressive indirect reclaim	
Min Watermark - Direct reclaim	
Min Watermark - Direct reclaim	

Watermark levels

kswapd tuning

Kernel knobs for tuning kswapd behavior:

- **/proc/sys/vm/swappiness** Defines the aggressiveness of swapping out memory pages of inactive processes.
 - **Range:** 0-100 **Default**: 60 Ο
 - High values can cause Kernel to swap out processes even when enough Ο memory is available.
 - Low values can cause Kernel to not swap out processes even when the Ο available memory is low.
 - **Recommendation:** Ο
 - Devices with high physical memory use lower swappiness values.
 - Devices with low physical memory use higher swappiness values.
- /proc/sys/vm/watermark_scale_factor Used to scale the buffer spaces between memory zone watermarks.
 - **Range:** 0 1000 **Default:** 10 Ο
 - 10 means buffer space is 0.1% of available memory.
 - 1000 means buffer space is 10% of available memory.
 - Low values can cause too much direct reclaim or kswapd not freeing up Ο enough memory in a single pass.
 - High values can cause kswapd to free up more memory than needed. Ο

- - Recommended value range: 1% 2% of total system memory. Ο
 - High values can scale up watermark buffer spaces leading to Ο
 - kswapd freeing too much memory than needed.
 - Frequent kswapd invocation causing CPU contention to spike.
 - Low values can cause kswapd to not free up enough memory leading to 0 System slowdown
- - Hangs/crashes

/proc/sys/vm/min_free_kbytes - Amount of free memory kept in reserve at all times. Defines min watermark across all memory zones.

Memory fragmentation

kswapd tuning example

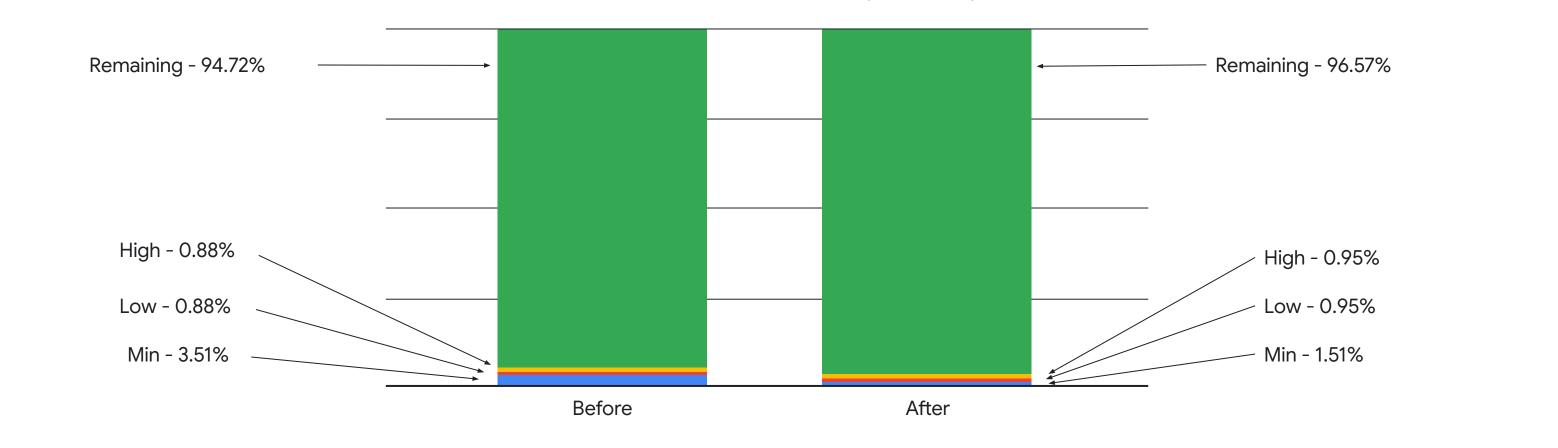
Before

- /proc/sys/vm/watermark_scale_factor 1
- /proc/sys/vm/min_free_kbytes 144 MiB
- /proc/sys/vm/swappiness 60

After

- /proc/sys/vm/swappiness 60

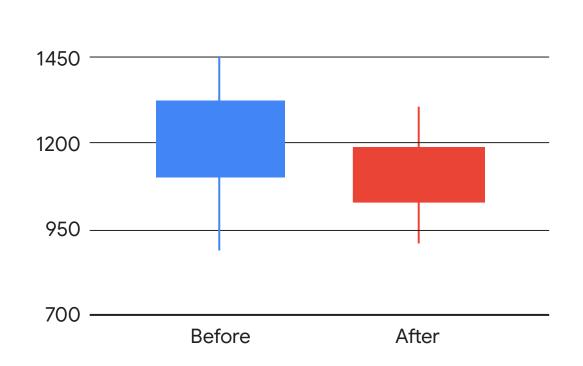
Watermark Level Size Relative to Total Physical Memory



/proc/sys/vm/watermark_scale_factor - 109

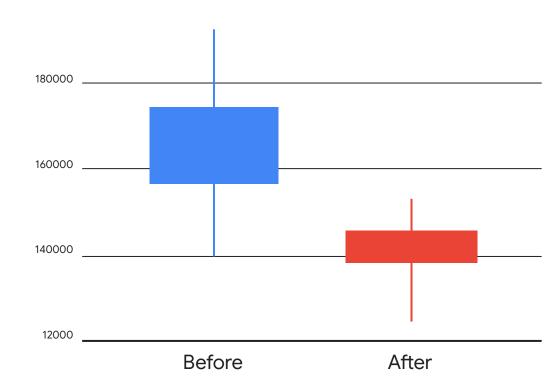
/proc/sys/vm/min_free_kbytes - 60 MiB

kswapd tuning example



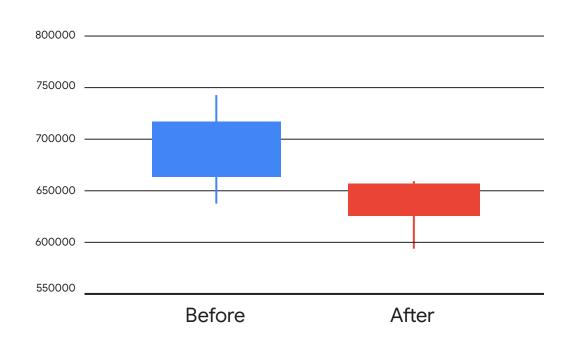
Kswapd cpu time millis

Working set refault file



Google Automotive Partner Bootcamp

Page stolen by kswapd



Dex Optimization

Google Automotive Partner Bootcamp



Dex Optimization

What is Dex Optimization?

By default in Android, apps are executed in interpreted mode. Dex optimization allows for compilation of selected code paths to machine code, which accelerates code execution.

Why is this Important?

Background dex optimization already occurs regularly as users interact with their apps. However, after initial installation, boot time performance may be degraded if all apps run in interpreted mode. There is an opportunity to compile key apps ahead of time to greatly reduce boot times.

Which Apps should be Dex Pre-Optimized?

When apps run in interpreted mode, they will also undergo JIT compilation. Post-boot, a high level of JIT compilation will lead to a high degree of CPU contention, further exacerbating degraded startup times. Identifying processes with high JIT CPU time will indicate apps that can be dex pre-optimized, forgoing JIT entirely.



Dex Optimization

Heavy JIT Compilation in Perfetto

JIT compilation activity can be visualized via Perfetto.

Within a specific process, one can identify the JIT thread pool tracks. In this example, Car Assistant is displaying a large number of JIT compilation events.

 com.google.android.carassistant:sear ch 3267 					
Jit thread pool 3274	R., R Run., R., R., R., R.	R R., Run., Runn., R Runn., S	Runnab. R., R. R. Ru., R., Run., Runn	18. R. Run. Runnable	Ru-12 River R
Jit thread pool 3274	JIT com JIT - J. JIT - J. Com Compil Comp. C Com G C D	J., JI., JIT., J J JIT., J C., Co., Com., C C Com., C (JIT JIT c J JIT compiling t Comp Compi C Compiling basel.	JIT JIT Compi. J C Com. Compiling C	JI., J., J. J., J., J., J., J., J., Co., Co., Co., Co., Co., Co., Co., Co
HeapTaskDaemon 3275	1				11
HeapTaskDaemon 3275					



Dex Optimization

Perfetto Query for Top Processes with the Most JIT CPU Time

Using the query will allow one to obtain a table as shown below:

total_duration	instances	prefix_name	
5936808097	12608	JIT	com.google.android.carassistant
4889113085	9984	JIT	com.google.android.apps.map
4597115883	6785	JIT	com.android.vendin
2342370393	3474	JIT	com.google.android.apps.geo.automotive.adas
1440327723	2422	JIT	com.android.vending
1078105791	2169	JIT	com.google.android.tt

DROP VIEW IF EXISTS interesting_slices_d0; CREATE VIEW interesting slices dO AS select id as slice_id, ts, dur, name, track_id, track_name, thread_name, utid, tid, process_name, upid, pid from _slice_with_thread_and_process_info where depth=0;

DROP TABLE IF EXISTS slice thread state breakdown; CREATE VIRTUAL TABLE slice thread state breakdown USING SPAN_LEFT_JOIN(interesting_slices_d0 PARTITIONED utid, thread_state PARTITIONED utid);

SELECT sum(dur) total_duration, count(*) instances, substr(name, 0, IIF(instr(name, ' ') > 0, instr(name, ' '), IIF(instr(name, ',') > 0, instr(name, ','), length(name)))) as prefix_name, substr(process_name, 0, IIF(instr(process_name, ':') > 0, instr(process_name, ':'), length(process_name))) as process_name_prefix FROM slice_thread_state_breakdown WHERE state = 'Running' and prefix name = "JIT" group by prefix_name, process_name_prefix order by total_duration desc;

INCLUDE PERFETTO MODULE slices.slices;

Dex Optimization Configuration

How to Configure Dex Pre-Optimization

For more details refer to https://source.android.com/docs/core/runtime/configure#build_options.

Add packages to the following makefile configuration:

PRODUCT_DEXPREOPT_SPEED_APPS += \ MapsCarPrebuilt \

Dex Optimization Configuration

How to Verify that an App is Dex Pre-Opted

Run the following ADB command:

\$ adb shell pm art dump com.google.android.apps.maps # Older releases may need to use this command instead: \$ adb shell dumpsys package dexopt | grep -i com.google.android.apps.maps -A 2

The following output indicates Google Maps is executed in interpreted mode:

```
[com.google.android.apps.maps]
path:
/system/product/priv-app/MapsCarPrebuilt/MapsCarPrebuilt.apk
x86_64: [status=verify] [reason=prebuilt]
```

The following output indicates that Google Maps was dex pre-opted:

```
[com.google.android.apps.maps]
path: /product/priv-app/MapsCarPrebuilt/MapsCarPrebuilt.apk
x86_64: [status=speed] [reason=prebuilt] [primary-abi]
```

Further Materials / Important Links

Summary of useful information per section:

Trace Configuration:

https://perfetto.dev/docs/reference/trace-config-proto

How to Collect a Perfetto Trace:

https://perfetto.dev/docs/quickstart/android-tracing

Android Boot Tracing:

https://perfetto.dev/docs/case-studies/android-boot-tracing

CPU Tracks:

https://perfetto.dev/docs/data-sources/cpu-scheduling

Memory Tracks:

https://perfetto.dev/docs/data-sources/memory-counters

Atrace Logging:

https://perfetto.dev/docs/data-sources/atrace



Thank you



Google Automotive Partner Bootcamp

Google Automotive Partner Bootcamp

