



Discussion 12

Profiling

--xuqing

立志成才报国裕民



Content

- Htop
- Btop
- Perf
- Vtune
- sar

Why profiling?

- See how the program works.
- Find the bottleneck of the performance.
- Check cpu usage or memory access pattern.
- Compare different modifications
- Benchmarking

01 htop

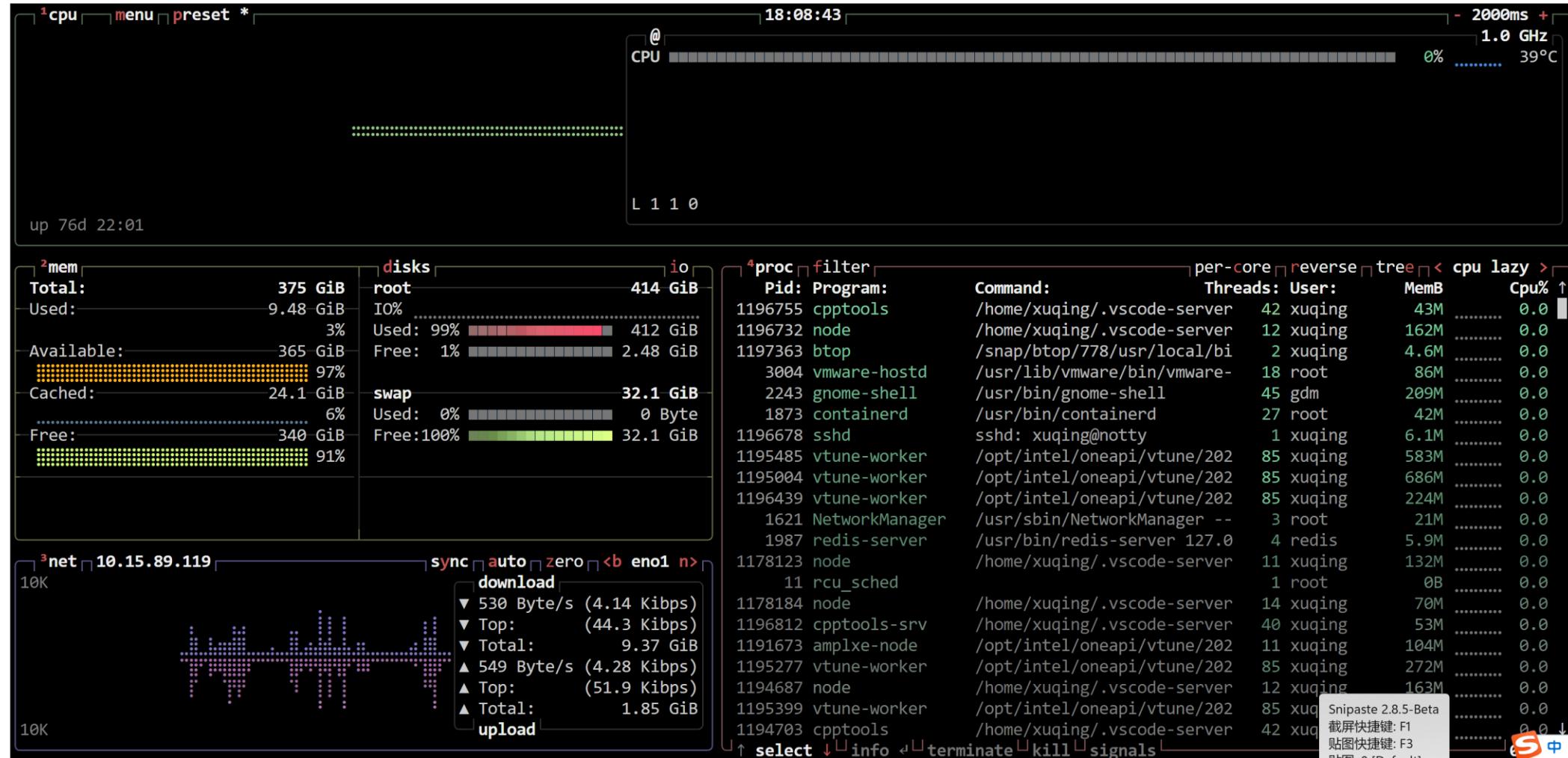


- htop - an interactive process viewer
- You can install it by `apt install htop` on ubuntu.
- Then run by `htop` in your terminal.
- F1 Open help
- F2 Setup (Configuration)
- F9 Kill
- F10 Quit
- Please read the friendly manual for more information.
- If you are interested, try CS130 Operating System.

02 btop



- Resource monitor that shows usage and stats for processor, memory, disks, network and processes.
- H Open help



03 perf



- perf is a Linux performance analysis tool
- Install
 - `sudo apt install linux-tools-common`
 - `perf -v`

03 perf



perf

usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]

The most commonly used perf commands are:

annotate	Read perf.data (created by perf record) and display annotated code
archive	Create archive with object files with build-ids found in perf.data file
bench	General framework for benchmark suites
buildid-cache	Manage build-id cache.
buildid-list	List the buildids in a perf.data file
c2c	Shared Data C2C/HITM Analyzer.
config	Get and set variables in a configuration file.
daemon	Run record sessions on background
data	Data file related processing
diff	Read perf.data files and display the differential profile
evlist	List the event names in a perf.data file
ftrace	simple wrapper for kernel's ftrace functionality
inject	Filter to augment the events stream with additional information
iostat	Show I/O performance metrics
kallsyms	Searches running kernel for symbols
kvm	Tool to trace/measure kvm guest os
list	List all symbolic event types
mem	Profile memory accesses
record	Run a command and record its profile into perf.data
report	Read perf.data (created by perf record) and display the profile
script	Read perf.data (created by perf record) and display trace output
stat	Run a command and gather performance counter statistics
test	Runs sanity tests.
top	System profiling tool.
version	display the version of perf binary
probe	Define new dynamic tracepoints

See 'perf help COMMAND' for more information on a specific command.

03 perf events



Events

The perf tool supports a list of measurable events. The tool and underlying kernel interface can measure events coming from different sources. For instance, some events are pure kernel counters, in this case they are called **software events**. Examples include: context-switches, minor-faults.

Another source of events is the processor itself and its Performance Monitoring Unit (PMU). It provides a list of events to measure micro-architectural events such as the number of cycles, instructions retired, L1 cache misses and so on. Those events are called **PMU hardware events** or **hardware events** for short. They vary with each processor type and model.

The `perf_events` interface also provides a small set of common hardware events monikers. On each processor, those events get mapped onto an actual events provided by the CPU, if they exist, otherwise the event cannot be used. Somewhat confusingly, these are also called **hardware events** and **hardware cache events**.

Finally, there are also **tracepoint events** which are implemented by the kernel ftrace infrastructure. Those are **only** available with the 2.6.3x and newer kernels.

03 perf list



- Use `sudo perf list` to see your supported events.
- display perf events supported by a particular module:
 - `sudo perf list hw/cache/pmu/tracepoint/sw`

```
perf list

List of pre-defined events (to be used in -e):

cpu-cycles OR cycles          [Hardware event]
instructions                   [Hardware event]
cache-references               [Hardware event]
cache-misses                  [Hardware event]
branch-instructions OR branches [Hardware event]
branch-misses                 [Hardware event]
bus-cycles                     [Hardware event]
ref-cycles                     [Hardware event]

cpu-clock                      [Software event]
task-clock                      [Software event]
page-faults OR faults          [Software event]
minor-faults                    [Software event]
major-faults                    [Software event]
context-switches OR cs          [Software event]
cpu-migrations OR migrations   [Software event]
alignment-faults                [Software event]
emulation-faults                [Software event]
bpf-output                      [Software event]
cgroup-switches                 [Software event]
dummy                           [Software event]

L1-dcache-loads                [Hardware cache event]
L1-dcache-load-misses           [Hardware cache event]
L1-dcache-stores                [Hardware cache event]
L1-dcache-store-misses          [Hardware cache event]
L1-dcache-prefetches            [Hardware cache event]
```

03 perf stat



- For any of the supported events, perf can keep a running count during process execution.
- In counting modes, the occurrences of events are simply aggregated and presented on standard output at the end of an application run

```
root@AEP:/home/xuqing# perf stat -h

Usage: perf stat [<options>] [<command>]

-a, --all-cpus          system-wide collection from all CPUs
-A, --no-aggr           disable CPU count aggregation
-B, --big-num            print large numbers with thousands' separators
```

03 perf stat



- perf stat -B dd if=/dev/zero of=/dev/null count=1000000

```
root@AEP:/home/xuqing# perf stat -B dd if=/dev/zero of=/dev/null count=1000000
1000000+0 records in
1000000+0 records out
512000000 bytes (512 MB, 488 MiB) copied, 1.36648 s, 375 MB/s
```

Performance counter stats for 'dd if=/dev/zero of=/dev/null count=1000000':

1,369.07 msec	task-clock	#	0.998 CPUs utilized
3	context-switches	#	0.002 K/sec
0	cpu-migrations	#	0.000 K/sec
82	page-faults	#	0.060 K/sec
5,248,723,740	cycles	#	3.834 GHz
10,131,239,111	instructions	#	1.93 insn per cycle
1,864,773,505	branches	#	1362.069 M/sec
26,839,770	branch-misses	#	1.44% of all branches

1.371657112 seconds time elapsed

0.127940000 seconds user
1.243423000 seconds sys

03 perf stat



- measure one or more events per run of the perf tool
- Events are designated using their symbolic **names** followed by optional **unit masks** and **modifiers**.
 - **perf stat -e cycles:uk dd if=/dev/zero of=/dev/null count=100000**

```
root@AEP:/home/xuqing# perf stat -B -e branch-misses,cache-misses,cache-references dd if=/dev/zero of=/dev/null count=1000000
1000000+0 records in
1000000+0 records out
512000000 bytes (512 MB, 488 MiB) copied, 1.37347 s, 373 MB/s

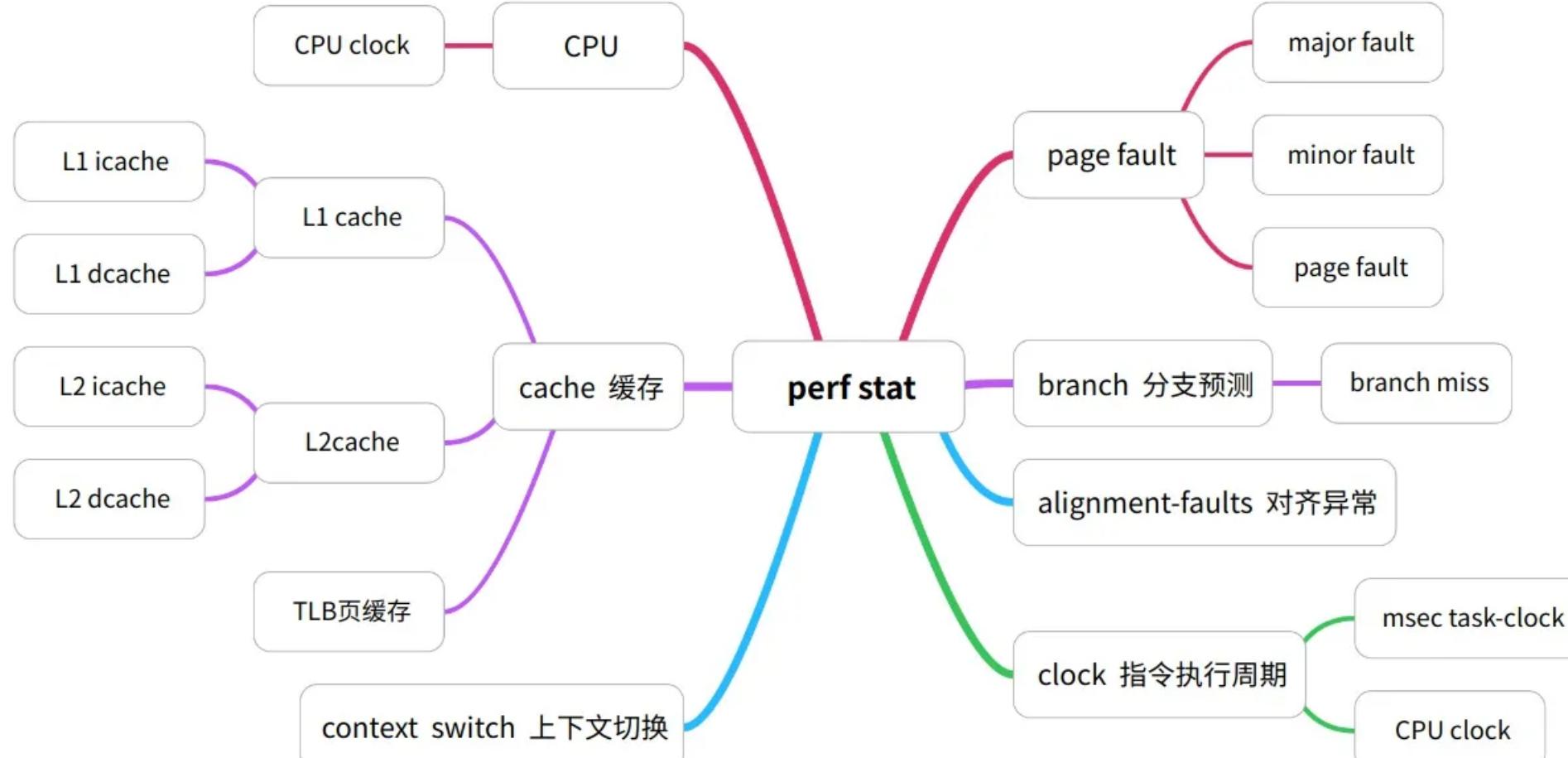
Performance counter stats for 'dd if=/dev/zero of=/dev/null count=1000000':

      27,691,443      branch-misses
          8,272      cache-misses
            34,323      cache-references
                                         #  24.100 % of all cache refs

  1.376950394 seconds time elapsed

  0.116095000 seconds user
  1.261039000 seconds sys
```

03 perf stat



03 perf record



The perf tool can be used to collect profiles on **per-thread**, **per-process** and **per-cpu** basis.

1. collect the samples : **perf record**.

- generates an output file called **perf.data**.
- **perf record -e retired_instructions:u -c 2000 ./noploop 4**

2. analyse file : **perf report** ; **perf annotate**

- possibly on another machine

```
root@AEP:/home/xuqing# perf record -h

Usage: perf record [<options>] [<command>]
      or: perf record [<options>] -- <command> [<options>]

-a, --all-cpus          system-wide collection from all CPUs
-b, --branch-any        sample any taken branches
-B, --no-buildid        do not collect buildids in perf.data
-c, --count <n>         event period to sample
-C, --cpu <cpu>         list of cpus to monitor
-d, --data               Record the sample addresses
-D, --delay <n>          ms to wait before starting measurement after program start
-e, --event <event>     event selector. use 'perf list' to list available events
-F, --freq <freq or 'max'>
-g                      profile at this frequency
-g                      enables call-graph recording
-G, --cgroup <name>     monitor event in cgroup name only
```

03 perf record



```
root@AEP:/home/xuqing/bigTestcase# perf record -g ./pffast pictures/test1.jpg output.jpg 0.8 -3.0 3.0 1801 1701
0.397437
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 11.558 MB perf.data (84687 samples) ]

-rw----- 1 xuqing xuqing 3122668 5月 16 22:33 perf.data
```

report

Samples: 76K of event 'cycles', Event count (approx.): 51352767535

Overhead	Command	Shared Object	Symbol
72.82%	pffast	[kernel.kallsyms]	[k] native_queued_spin_lock_slowpath
12.36%	pffast	pffast	[.] dot_product
7.06%	pffast	libgomp.so.1.0.0	[.] omp_get_num_procs
1.32%	pffast	pffast	[.] gb_.omp_fn.0
0.96%	pffast	[kernel.kallsyms]	[k] function_graph_enter
0.74%	pffast	[kernel.kallsyms]	[k] native_sched_clock
0.49%	pffast	[kernel.kallsyms]	[k] page_counter_cancel
0.43%	pffast	[kernel.kallsyms]	[k] prepare_ftrace_return
0.40%	pffast	[kernel.kallsyms]	[k] ftrace_graph_caller
0.40%	pffast	[kernel.kallsyms]	[k] trace_graph_entry
0.29%	pffast	pffast	[.] from_stbi_.omp_fn.0
0.15%	pffast	[kernel.kallsyms]	[k] native_irq_return_iRET
0.15%	pffast	[kernel.kallsyms]	[k] clear_page_ermS
0.14%	pffast	pffast	[.] stbiw_jpg_processDU
0.12%	pffast	[kernel.kallsyms]	[k] up_read
0.12%	pffast	[kernel.kallsyms]	[k] _raw_spin_lock
0.11%	pffast	[kernel.kallsyms]	[k] native_ticks_update

annotate

```
Samples: 76K of event 'cycles', 4000 Hz, Event count (approx.): 51352767535
native_queued_spin_lock_slowpath /proc/kcore [Percent: local period]

Percent          add    $0x30c00,%rax
                  add    -0x63be8680(%rsi,8),%rax
                  mov    %rdx,(%rax)
                  mov    0x8(%rdx),%eax
                  test   %eax,%eax
                  ↓ jne   158
14f: pause
                  mov    0x8(%rdx),%eax
                  test   %eax,%eax
                  ↑ je    14f
158: mov    (%rdx),%rax
                  test   %rax,%rax
                  ↓ je    1ba
                  mov    %rax,%rsi
                  prefetchw (%rax)
                  ↓ jmp   16a
168: pause
0.90 16a: mov    (%rdi),%eax
0.00 16a: test   %ax,%ax
0.05 16a: ↑ jne   168
0.01 16a: mov    %eax,%r8d

Press 'h' for help on key bindings
```

03 perf report



```
sudo perf report -f -g > res.txt
```

```
# Total Lost Samples: 0
#
# Samples: 84K of event 'cycles'
# Event count (approx.): 55350461031
#
# Children      Self  Command Shared Object      Symbol
# .....      .....      .....      .....      .....
#
#      94.40%    0.00% pffast  [unknown]      [k] 0000000000000000
|--94.40%--0
|   |--90.91%--omp_in_final
|   |   |--78.83%--gb._omp_fn.0
|   |   |   |--77.63%--page_fault
|   |   |   |   |--77.47%--do_page_fault
|   |   |   |   |   |--77.45%--__do_page_fault
|   |   |   |   |   |   |--77.32%--do_user_addr_fault
|   |   |   |   |   |   |   |--77.00%--handle_mm_fault
|   |   |   |   |   |   |   |   |--76.88%--__handle_mm_
```



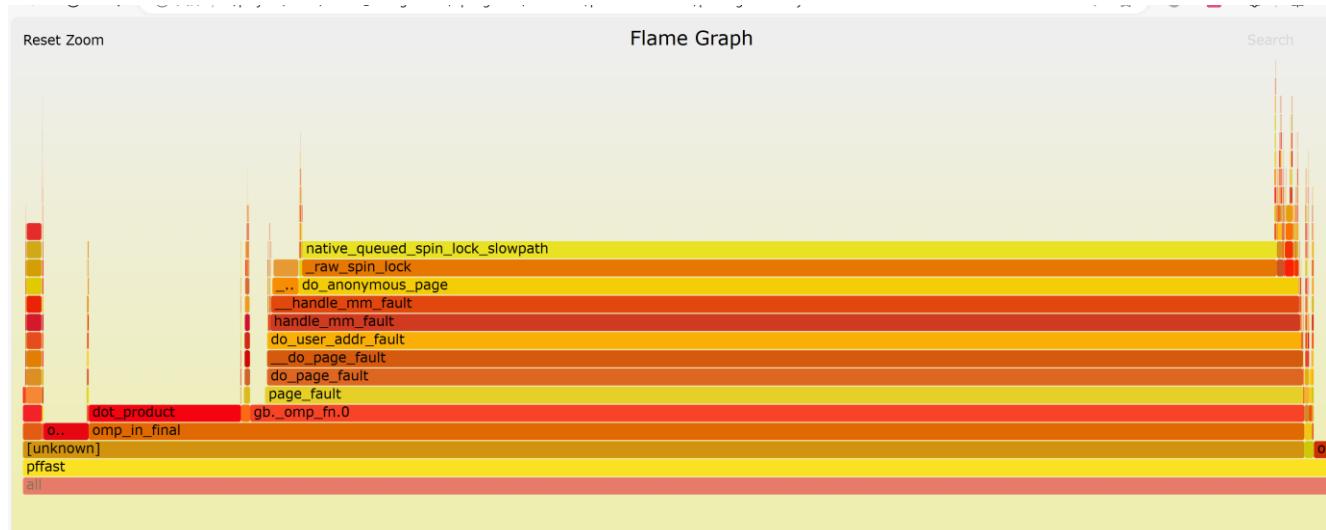
03 perf flame graph



The screenshot shows a browser window with the URL f.wiki.kernel.org/index.php/Tutorial#Commands. The page content is a table of contents for the 'CPU Flame Graph' section:

W Latest Latest 教程 - Tu X CPU Fl dd - 搜索 Lat
2.1.5 Repeated measurement
2.2 Options controlling environment selection
2.2.1 Counting and inheritance
2.2.2 Processor-wide mode
2.2.3 Attaching to a running process
2.3 Options controlling output
2.3.1 Pretty printing large numbers
2.3.2 Machine readable output
3 Sampling with perf record
3.1 Event-based sampling overview
3.1.1 Default event: cycle counting
3.1.2 Period and rate
3.2 Collecting samples
3.3 Processor-wide mode
3.4 Flame Graph
3.5 Firefox Profiler
4 Sample analysis with perf report
4.1 Options controlling output
4.2 Options controlling kernel reporting
4.3 Processor-wide mode
4.4 Overhead calculation
5 Source level analysis with perf annotate
5.1 Using perf annotate on kernel code
6 Live analysis with perf top
7 Benchmarking with perf bench
7.1 sched: Scheduler benchmarks
7.2 mem: Memory access benchmarks
7.3 numa: NUMA scheduling and MM benchmarks
7.4 futex: Futex stressing benchmarks

- CPU Flame Graphs (brendangregg.com)
- root@AEP:/home/xuqing/FlameGraph# perf script | ./stackcollapse-perf.pl> out.perf-folded
- root@AEP:/home/xuqing/FlameGraph# ./flamegraph.pl out.perf-folded> perf.svg
- [./perf.svg](#)



04 VTune Basics - Install



- [official website](#)
- It can only run with an Intel cpu, AMD user may use [AMD Prof](#)
- VTune is very simply and powerful

1. Install steps

2. Set Environment Variables:

3. Verify Your Installation: **vtune-self-checker.sh**

```
bin64# echo 0 > /proc/sys/kernel/yama/ptrace_scope  
bin64# echo 0 > /proc/sys/kernel/perf_event_paranoid
```

The system is ready for the following analyses:

- * Performance Snapshot
- * Hotspots and Threading with user-mode sampling
- * Hotspots with HW event-based sampling, HPC Performance Characterization, etc.
- * Microarchitecture Exploration
- * Memory Access
- * Hotspots with HW event-based sampling and call stacks
- * Threading with HW event-based sampling

The following analyses have failed on the system:

- * GPU Compute/Media Hotspots (characterization mode)
- * GPU Compute/Media Hotspots (source analysis mode)

04 VTune Basics – Web GUI



• Web Server Interface

```
^Xuqing@AEP:/opt/intel/oneapi/vtune/latest/bin64$ vtune-backend --allow-remote-access --enable-server-profiling
No TLS certificate was provided as a --tls-certificate command-line argument thus a self-signed certificate is generated to enable transport for the web server: /home/xuqing/.intel/vtune/settings/certificates/middleware.crt.
VTune Profiler GUI is accessible via https://10.15.89.119:33009/?one-time-token=176a0c8a02860371e91440e64631ea78
VTune Profiler GUI is accessible via https://172.16.78.1:33009/?one-time-token=176a0c8a02860371e91440e64631ea78
VTune Profiler GUI is accessible via https://172.16.0.1:33009/?one-time-token=176a0c8a02860371e91440e64631ea78
Warning: VTune Profiler Agent will be connected to https://10.15.89.119:33009/?one-time-token=176a0c8a02860371e91440e64631ea78
Collection has paused.
Collection has stopped.
```

The screenshot shows the VTune Profiler Web GUI interface. On the left, there is a sidebar with a table titled "Profile Applications on This Target". The table has columns for "Type of Collection" (Windows, Linux, FreeBSD, Android) and rows for "Install VTune Profiler on This Host" (Windows, Linux). The "Windows" row shows "Local" collection types for GUI|CLI, N/A, CLI*, and N/A. The "Linux" row shows "Remote" collection types for Use Browser, GUI|CLI, GUI|CLI, and GUI|CLI. The "Linux" row also shows "Local" collection types for N/A, GUI|CLI, CLI*, and N/A. The "Remote" row for Linux shows "Use Browser (View Results Only)" for GUI|CLI, GUI|CLI, and GUI|CLI.

The main area of the interface is the "Project Navigator" which lists a project named "sample (matrix)" containing four items: r000hs, r001ue, r002ps, and r003ps. A "Welcome" tab is also visible. At the bottom right, there is a large blue button labeled "Configure Analysis..." and a link "New Project...".

04 VTune Basics - procedure



1. Prepare your target application for analysis:

Recommendations :

- Release mode, with maximum appropriate compiler optimization level.
- generate debug information for your application (-O2 -g)

2. Prepare your target system for analysis:

- Build and install the sampling drivers, if required.

(If the drivers were not built and set up during installation , VTune Profiler provides an error message and enables driverless sampling data collection based on the Linux Perf* tool functionality, which has a limited scope of analysis options.)

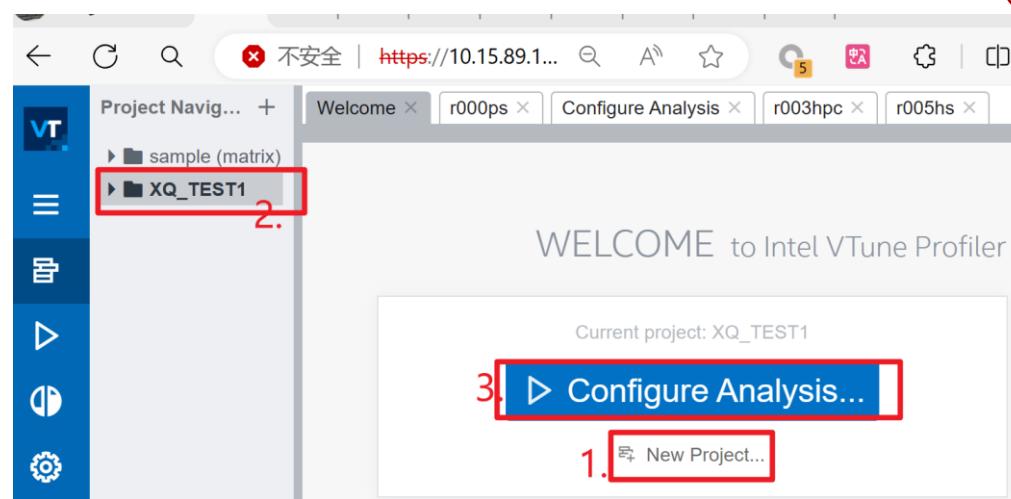
3. Create a VTune Profiler project and run the performance analysis of your choice.

04 VTune Basics - example



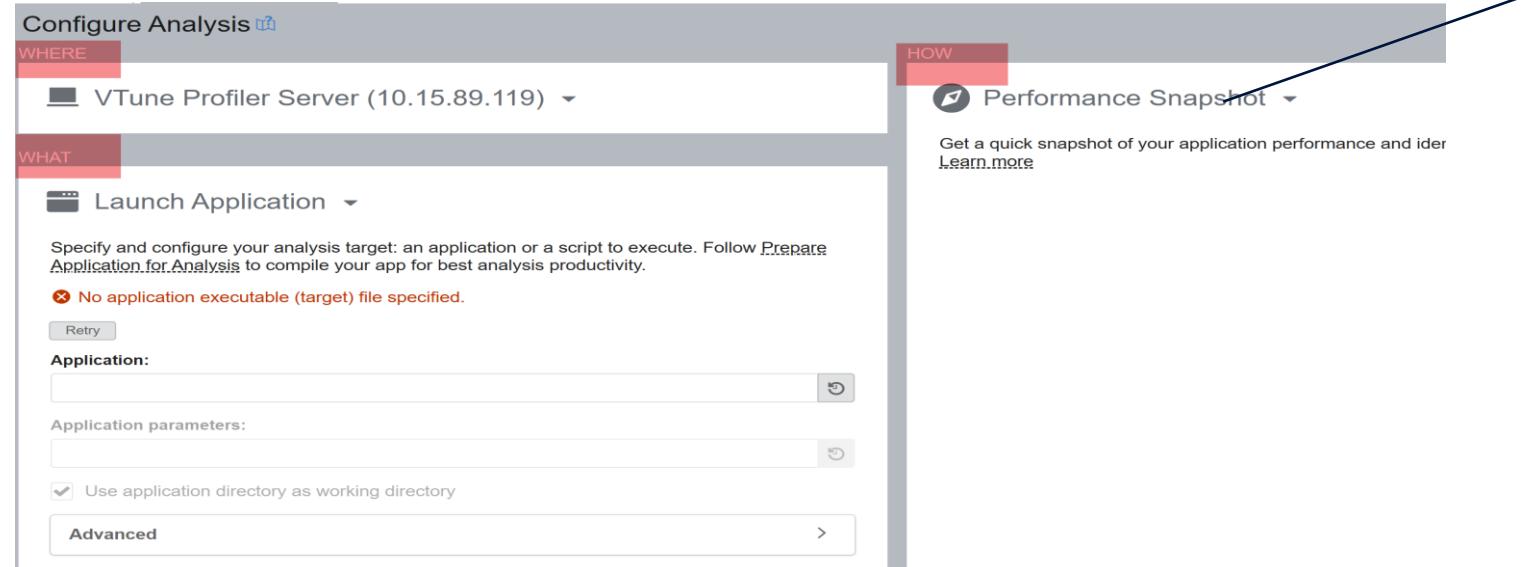
The screenshot shows the Intel VTune Profiler User Guide. The 'Set Up Project' section is highlighted with a red background. Other sections like 'User Guide', 'Introduction', and 'Install Intel® VTune™ Profiler' are also listed.

1.



an overview of issues that affect the performance of application.

2.



04 VTune Basics – performance snapshot example



Launch Application ▾

Specify and configure your analysis target: an application [Application for Analysis](#) to compile your app for best analysis.

Application: /home/xuqing/bigTestcase/gbfloat_base

Application parameters: pictures/test1.jpg output.jpg 0.8 -3.0 3.0 1801 1701

Use application directory as working directory

Performance Snapshot ⓘ ⓘ

Analysis Configuration Collection Log Summary

Choose your next analysis type

Select a highlighted recommendation based on your performance snapshot.

ALGORITHM

- Hotspots
- Anomaly Detection (preview)
- Memory Consumption

PARALLELISM

- Threading 5.5%
- HPC Performance Characterization 0.0%

ACCELERATORS

- GPU Offload
- GPU Compute/Media Hotspots (preview)
- CPU/FPGA Interaction

MICROARCHITECTURE

- Microarchitecture Exploration

I/O

- Input and Output

PLATFORM ANALYSES

- System Overview
- GPU Rendering (preview)

```
xuqing@AEP:~/bigTestcase$ ./gbfloat_base pictures/test1.jpg output.jpg 0.8 -3.0 3.0 1801 1701
8.042599
```

Elapsed Time ⓘ: 8.404s

IPC ⓘ:	2.980
SP GFLOPS ⓘ:	8.350
DP GFLOPS ⓘ:	0.005
x87 GFLOPS ⓘ:	0.000
Average CPU Frequency ⓘ:	3.8 GHz

Logical Core Utilization ⓘ:

next

04 VTune Basics – hpc example



Vectorization^②: 0.0% ↘ of Packed FP Operations

Instruction Mix:

SP FLOPs ②:	17.7%	of uOps
Packed ②:	0.0%	from SP FP
Scalar ②:	100.0% ↘	from SP FP
DP FLOPs ②:	0.0%	of uOps
x87 FLOPs ②:	0.0%	of uOps
Non-FP ②:	82.3%	of uOps

FP Arith/Mem Rd Instr. Ratio ②: 1.439

FP Arith/Mem Wr Instr. Ratio ②: 123.052

Top Loops/Functions with FPU Usage by CPU Time

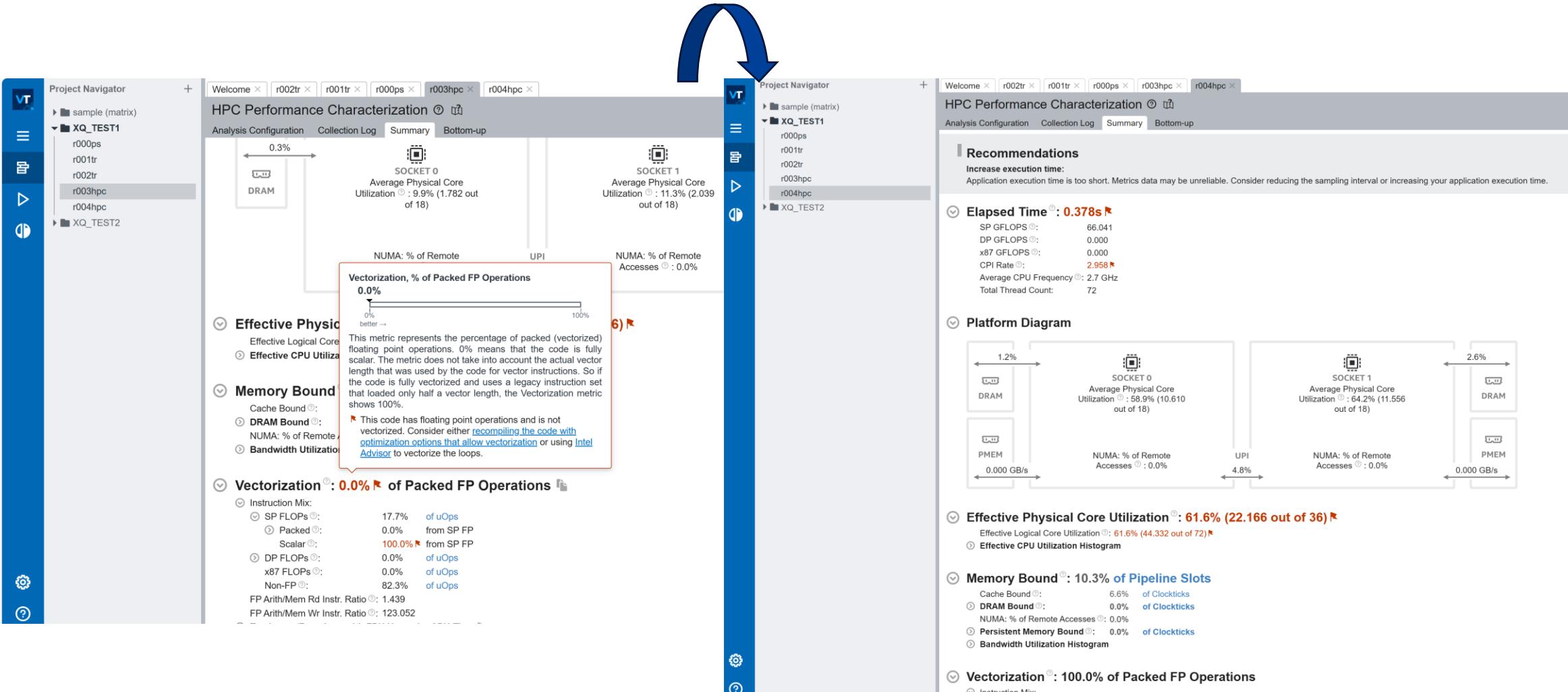
This section provides information for the most time con

Can be optimized

Function	CPU Time ②	% of FP Ops ②	FP Ops: Packed ②	FP Ops: Scalar ②	Vector Instruction Set ②	Loop Type ②
[Loop@0x39d8 in gb_v_omp_fn.0]	17.210s	17.5%	0.0%	100.0% ↘		
[Loop@0x36e0 in gb_h_omp_fn.0]	14.410s	18.6%	0.0%	100.0% ↘		

*N/A is applied to non-summable metrics.

04 VTune Basics – hpc example



04 VTune Basics – hotspot example



Call stacks and cpu time

Call Stacks

Call Stack	CPU Time (%)
gbfloat_base ! gb_v_omp_fn.0	75.4% (12.928s of 17.152s)
libgomp.so.1 ! func@0x1a734+0x139	
libpthread.so.0 ! start_thread+0xd8 - pthread_create.c:477	
libc.so.6 ! clone+0x42 - clone.S:95	

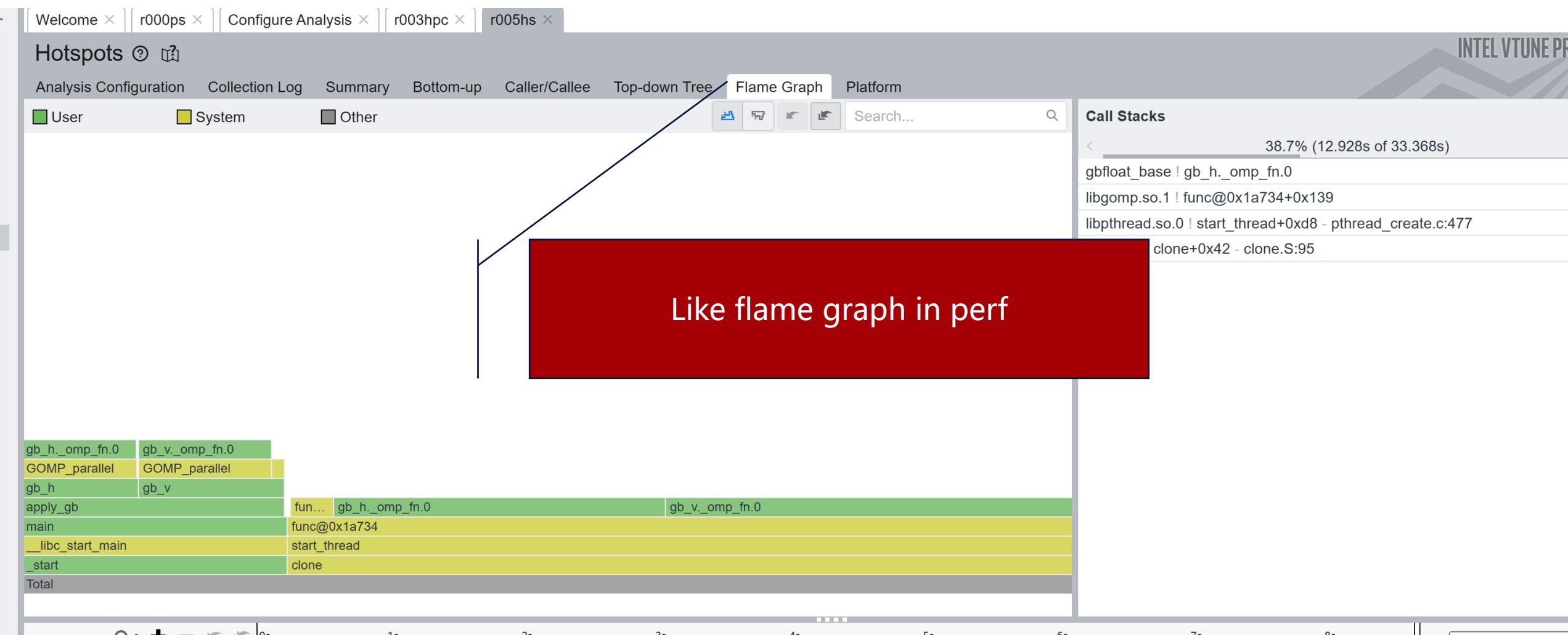
Function / Call Stack

Function / Call Stack	CPU Time	Module	Function
gb_v_omp_fn.0	17.152s	gbfloat_base	gb_v_omp_fn.0
func@0x1a734 ← start_thread ← clone	12.928s	libgomp.so.1	func@0x1a734
GOMP_parallel ← gb_v ← apply_gb ← main ← __libc_start_main ← _start	4.224s	libgomp.so.1	GOMP_parallel
gb_h_omp_fn.0	14.155s	gbfloat_base	gb_h_omp_fn.0
func@0x1a734 ← start_thread ← clone	10.535s	libgomp.so.1	func@0x1a734
GOMP_parallel ← gb_h ← apply_gb ← main ← __libc_start_main ← _start	3.620s	libgomp.so.1	GOMP_parallel
func@0x1d374	1.368s	libgomp.so.1	func@0x1d374
func@0x1bd0e	0.459s	libgomp.so.1	func@0x1bd0e
func@0x1d1d0	0.096s	libgomp.so.1	func@0x1d1d0
stbiw_jpg_processDU	0.020s	gbfloat_base	stbiw_jpg_processDU
stbi_loadf_main	0.020s	gbfloat_base	stbi_loadf_main
GOMP_parallel	0.016s	libgomp.so.1	GOMP_parallel
munmap	0.013s	libc.so.6	munmap
stbi_idct_simd	0.012s	gbfloat_base	stbi_idct_simd
func@0x1a734	0.012s	libgomp.so.1	func@0x1a734
stbiw_jpg_DCT	0.012s	gbfloat_base	stbiw_jpg_DCT
stbi_write_jpg_core.part.0	0.008s	gbfloat_base	stbi_write_jpg_core.part.0

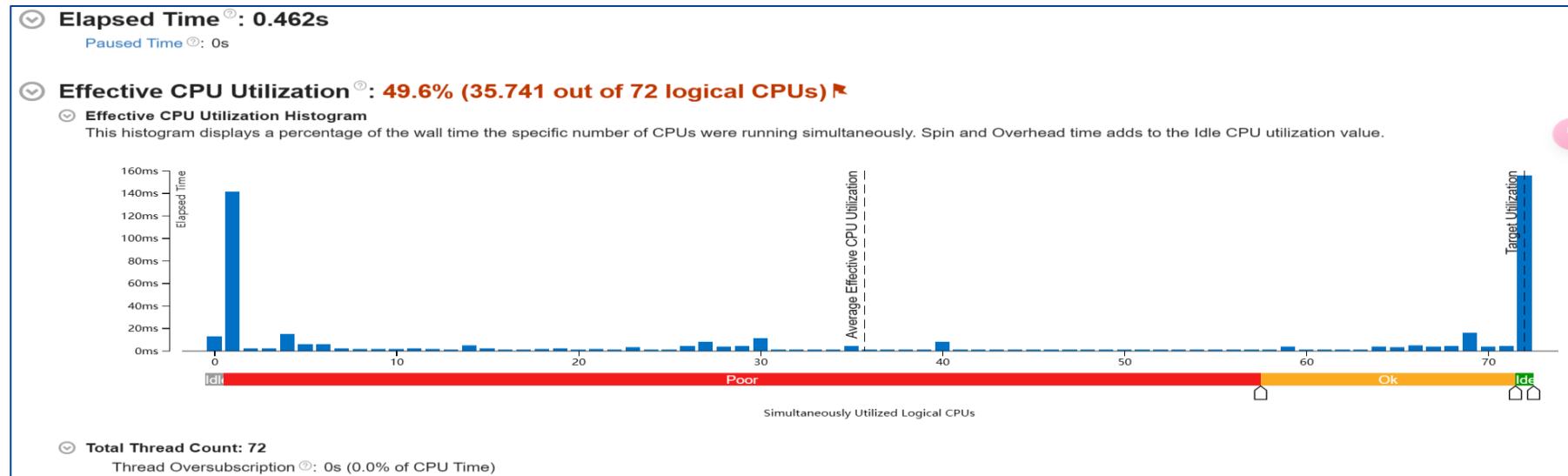
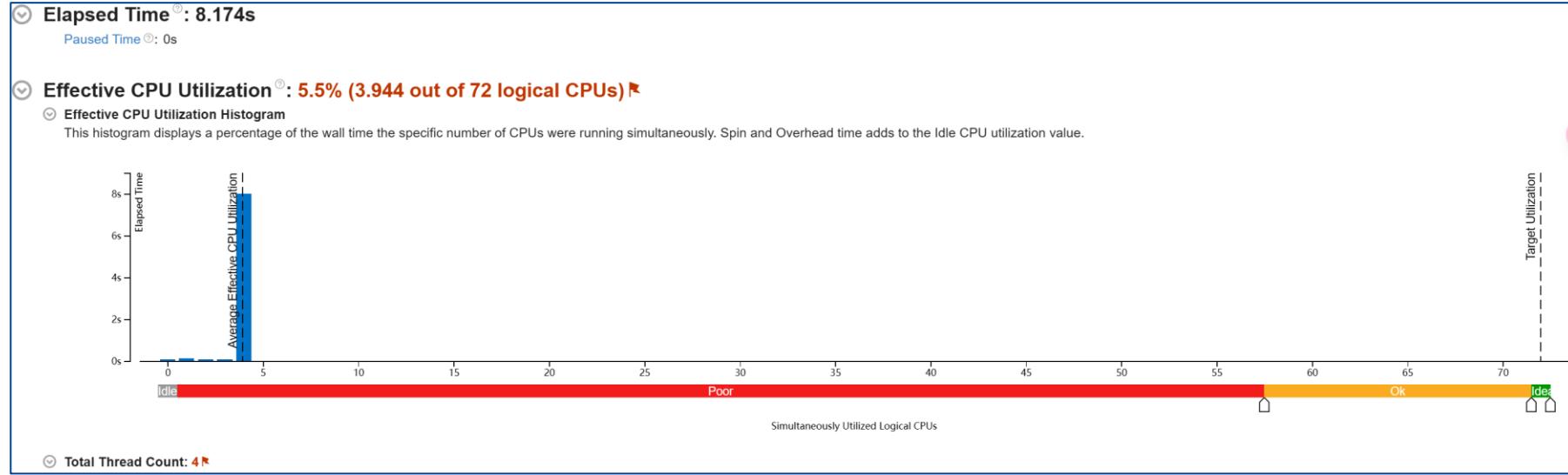
04 VTune Basics – hotspot example



上海科技大学
ShanghaiTech University



04 VTune Basics – threading example



04 VTune Basics



The screenshot shows the Intel VTune Profiler interface with several numbered callouts:

- 1**: Project Navigator pane on the left, showing a tree structure with nodes like sample (matrix) and Test.
- 2**: Filter and search icons in the Project Navigator pane.
- 3**: Top navigation bar with tabs: Welcome, r000hs, r004hs, Hotspots (selected), Collection Log, Summary, Bottom-up, Caller/Callee, Top-down Tree, Platform.
- 4**: Analysis Configuration toolbar with various icons.
- 5**: Hotspots table header: Function / Call Stack, CPU Time, Instructions Retired, Microarchitecture Usage, Module, Function (Fu).
- 6**: Hotspots table data rows, showing functions like func@0x1401bb230, func@0x101e8890, etc., with their respective CPU times, instruction counts, and module names.
- Bottom Panel:** Shows a timeline from 0s to 10s for multiple threads (TID: 0, 12500, 7608, 2920, 28564, 11340, 15008). A legend on the right defines the colors: green for Running, blue for CPU Time, red for Spin and Overhead, and grey for Clocktick Sample. It also lists CPU Time metrics.

05 sar



- sar - Collect, report, or save system activity information.
- Report selected cumulative activity counters.
- sar –help
- Usage: sar [options] [<interval> [<count>]]
 - interval == zero, displays the average statistics for the time since the system was started
 - without the count parameter, reports are generated continuously
- -o output_filename

```
xuqing@AEP:~/bigTestcase$ sar -q 1 3
Linux 5.4.0-169-generic (AEP) 2024年05月17日 _x86_64_ (72 CPU)

19时13分16秒 runq-sz plist-sz ldavg-1 ldavg-5 ldavg-15 blocked
19时13分17秒      2     1804    0.09    0.04    0.00      0
19时13分18秒      0     1806    0.09    0.04    0.00      0
19时13分19秒      0     1803    0.08    0.04    0.00      0
Average:          1     1804    0.09    0.04    0.00      0
```