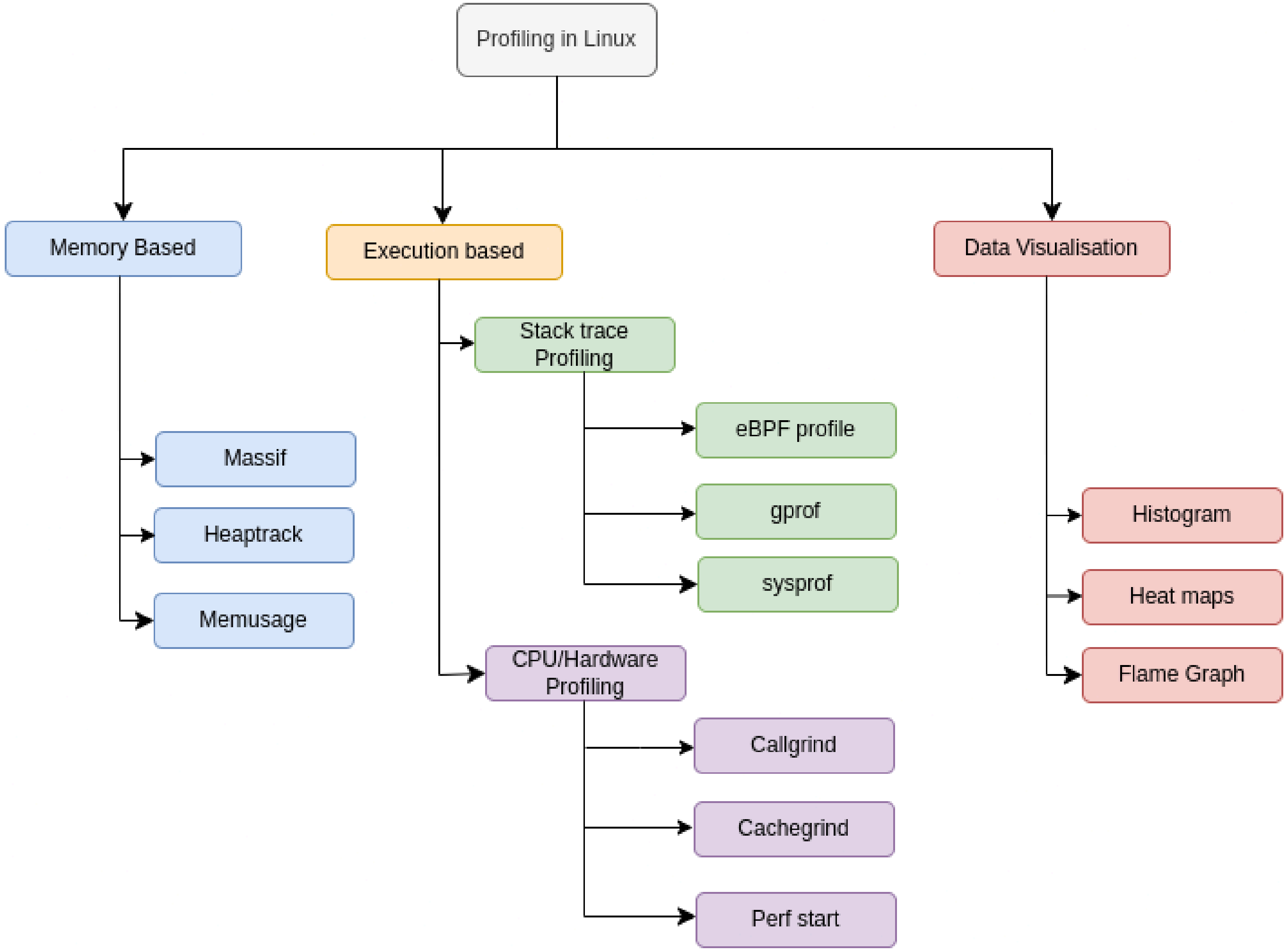
Module 6
Profiling in Linux

Introduction

- Profiling is the process of collecting and analyzing performance data from user applications or system-wide to understand their behavior and identify performance bottlenecks.
- A profiling tool uses sampling techniques to collect data at defined intervals.
- Profiling provides an aggregated overview of system performance to identify bottlenecks, while tracing captures detailed, sequential event data to analyze execution flow and debug specific issues.
- Profiling can be done on different aspects of the system such as:
    - Function exection time
    - Function call count
    - Memory usage
    - Hardware and Software events

Profiling landscape in Linux



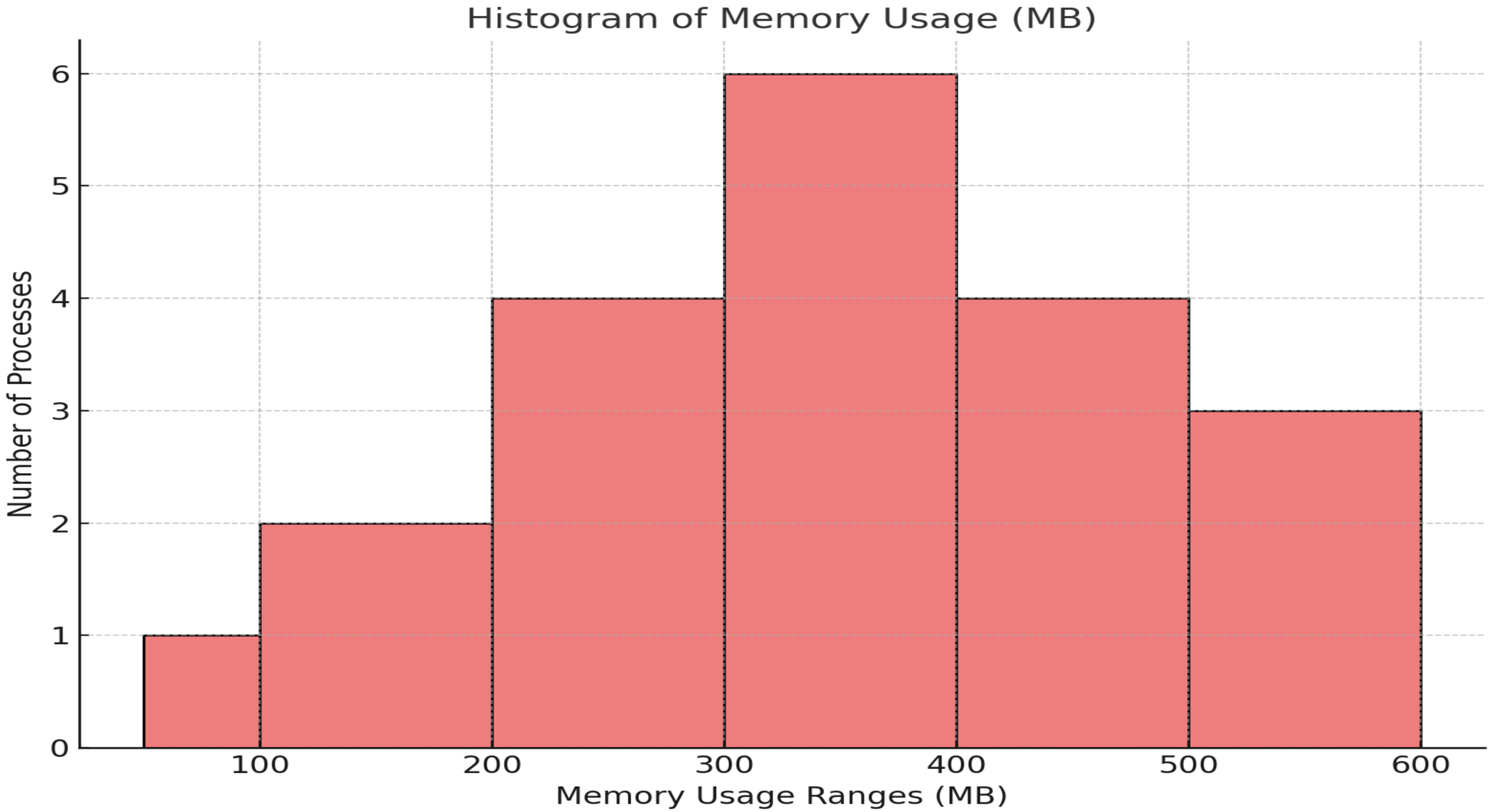
Data Visualization

Histogram

- A histogram is a type of graph that shows how data is distributed by grouping it into intervals or "bins."
- It helps understand the following:
  - Frequency of data points within specific ranges.
  - Understand patterns or trends in data.
  - Identify the shape of the data distribution (e.g., normal, skewed).
  - Spot outliers or unusual data points.
- It can be used to visualize the performance data collected by profiling tools, providing insights into resource utilization and identifying areas for optimization.
- **Example**: Analyzing the memory usage of processes running on a Linux server over a certain period.

```
Memory usage (in MB) for 20 processes:
[50, 120, 180, 200, 210, 230, 250, 300, 320, 330, 350, 370, 390, 410, 430, 450, 470, 500, 520, 550]
```

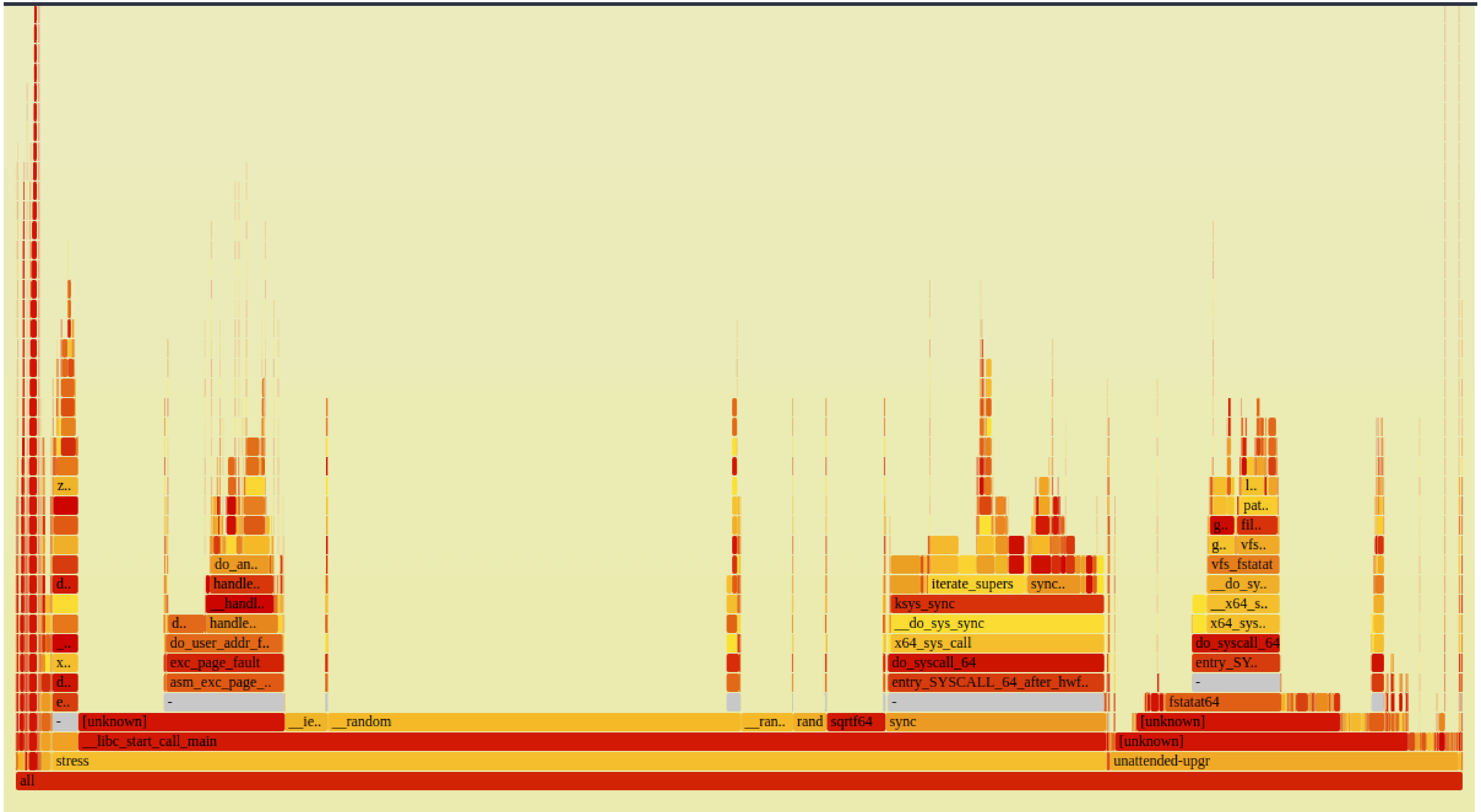
Histogram Visualization



FlameGraph

- Flame graphs are a type of visualization that represents sampled stack traces of a program's execution.
- Flame graphs are commonly used in performance analysis to identify performance bottlenecks, CPU hotspots, and inefficient code paths.
- The colors often represent different functions or function categories, aiding in quick identification.
- The vertical axis (Y-axis) represents the stack depth, showing the sequence of function calls counting from zero at the bottom.
- The horizontal axis (X-axis) represents stack profile population, sorted alphabetically. Each stack frame is represented by a horizontal bar, with wider bars indicating functions that consume more CPU time.
- Please refer to the instructions for flame graph generation.

FlameGraph Visualization



Memory profiling

- Memory profiling tools highlight inefficient memory usage patterns such as excessive allocations, unnecessary copying of data, and redundant memory accesses.
- Helps in finding and fixing memory leaks by tracking all memory allocations and deallocations, ensuring that memory is properly released.
- Enhances cache utilization by promoting better memory locality, allowing data to be accessed more quickly from the CPU cache.
- Reduces the need for frequent page swaps by maintaining a smaller working set in memory, thus minimizing paging and disk I/O.

Massif

- Massif is a memory profiler tool in Valgrind, used to analyze heap allocations, stack usage, and overall memory consumption of a program.
- It provides detailed analysis of heap memory usage, aiding in the identification of memory leaks and inefficient memory usage patterns.
- Massif tool can be invoked using following command line.

```
valgrind --tool=massif --time-unit=B <Application Program>
```

- Upon completion, Valgrind does not print summary statistics; instead, all of Massif's profiling data is written to a file, typically named massif.out.{pid} (filename can be changed using --massif-out-file option).
- You can then use the `ms_print` tool to visualize a heap allocation graph.

```
ms_print massif.out.5464
```

Massif report (1/2)

```
$ ms_print massif.out.5464
--------------------------------------------------------------------------------
Command:            ./massif
Massif arguments:   --detailed-freq=2 --time-unit=B
ms_print arguments: massif.out.5464
--------------------------------------------------------------------------------


    KB
10.58^                                                                   ####
     |                                                                   #
     |                                                                   #   :
     |                                                                   #   :
     |                                                                   #   :
     |                                                                   #   :
     |                                                                   #   :
     |                                                                   #   :
     |                                            :::::::::::::::::::::::#   :
     |                                            :                      #   :
     |                                            :                      #   :
     |                                            :                      #   :
     |                                            :                      #   :
     |                                            :                      #   :
     |                                            :                      #   :
     |                    @@@@@@@@@@@@@@@@@@@@@@@@:                      #   :
     |                    @                       :                      #   :
     |                    @                       :                      #   :
     |       @@           @                       :                      #   :
     |       @ :::::::::::@                       :                      #   :
   0 +----------------------------------------------------------------------->KB
     0                                                                   12.16

Number of snapshots: 10
 Detailed snapshots: [1, 3, 5, 7, 8 (peak)]
 --------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0                0             0            0
  1            408              408              400             8            0
98.04% (400B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->98.04% (400B) 0x1091AD: main (massif.c:15)
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  2          1,216            1,216            1,200            16            0
  3          1,216            1,216            1,200            16            0
98.68% (1,200B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->65.79% (800B) 0x1091BB: main (massif.c:16)
|
->32.89% (400B) 0x1091AD: main (massif.c:15)
```

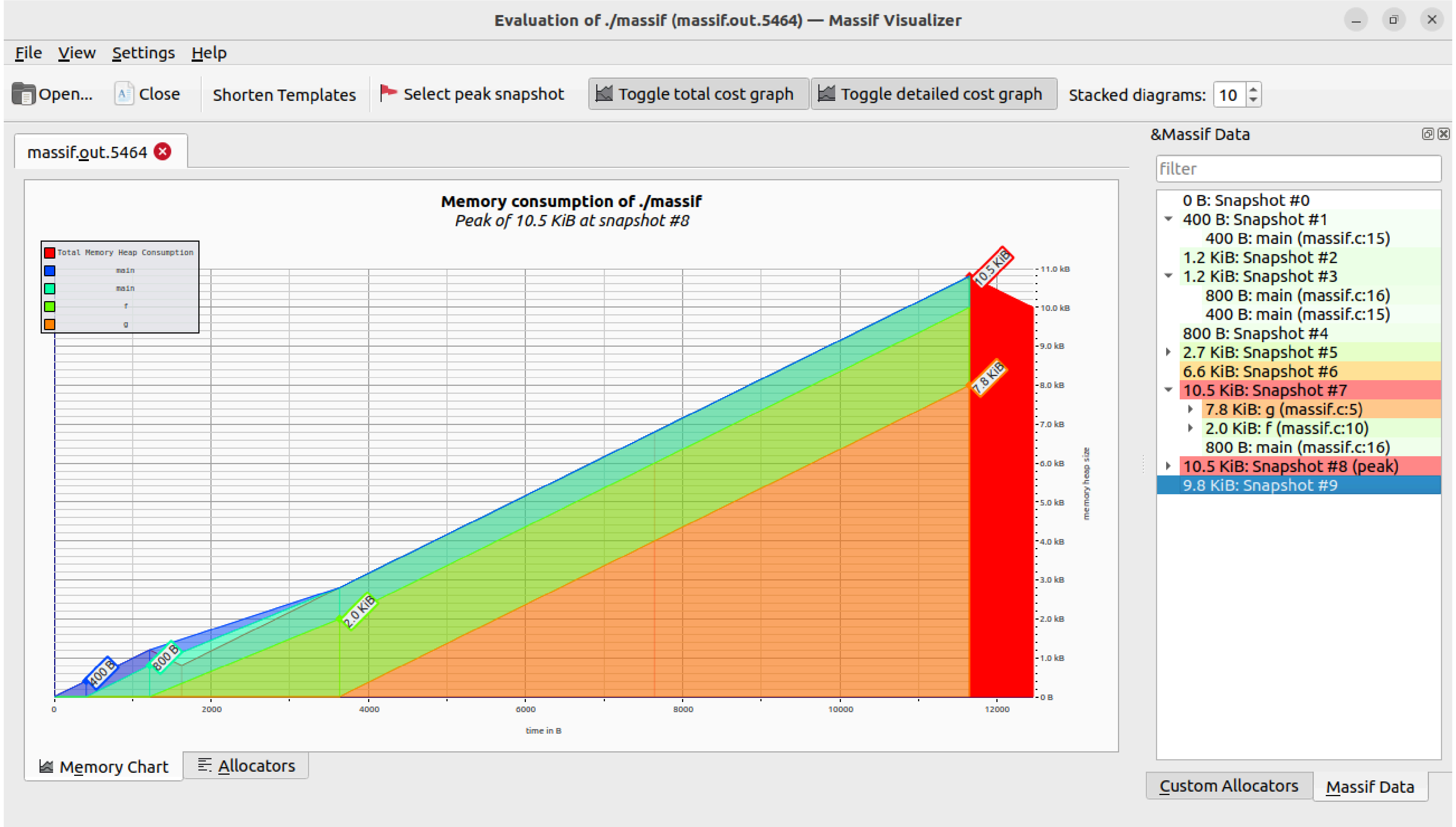
Massif report (2/2)

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  4          1,624              808              800             8            0
  5          3,632            2,816            2,800            16            0
99.43% (2,800B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->71.02% (2,000B) 0x10918F: f (massif.c:10)
| ->71.02% (2,000B) 0x109204: main (massif.c:26)
|
->28.41% (800B) 0x1091BB: main (massif.c:16)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)

--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  6          7,640            6,824            6,800            24            0
  7         11,648           10,832           10,800            32            0
99.70% (10,800B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->73.86% (8,000B) 0x10917A: g (massif.c:5)
| ->36.93% (4,000B) 0x109194: f (massif.c:11)
| | ->36.93% (4,000B) 0x109204: main (massif.c:26)
| |
| ->36.93% (4,000B) 0x109209: main (massif.c:28)
|
->18.46% (2,000B) 0x10918F: f (massif.c:10)
| ->18.46% (2,000B) 0x109204: main (massif.c:26)
|
->07.39% (800B) 0x1091BB: main (massif.c:16)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)

--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  8         11,648           10,832           10,800            32            0
99.70% (10,800B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->73.86% (8,000B) 0x10917A: g (massif.c:5)
| ->36.93% (4,000B) 0x109194: f (massif.c:11)
| | ->36.93% (4,000B) 0x109204: main (massif.c:26)
| |
| ->36.93% (4,000B) 0x109209: main (massif.c:28)
|
->18.46% (2,000B) 0x10918F: f (massif.c:10)
| ->18.46% (2,000B) 0x109204: main (massif.c:26)
|
->07.39% (800B) 0x1091BB: main (massif.c:16)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)

--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  9         12,456           10,024           10,000            24            0
```

Massif result analysis

- We are using massif sample program for this demonstration.
- Each vertical bar represents a snapshot, indicating the memory usage at a specific point in time.
- The columns are further divided into:
    - **Normal snapshots**, which record basic information and are represented using the ':' character.
    - **Detailed snapshots**, denoted by the '@' character, represent information about where allocations occurred for these snapshots.
    - **Peak snapshots**, represented by the '#' character, record the point where memory consumption was greatest.
- The graph is followed by the information for each snapshot.
- Normal snapshots provide basic memory usage counts, while detailed snapshots include an allocation tree indicating code sections responsible for heap memory allocation.

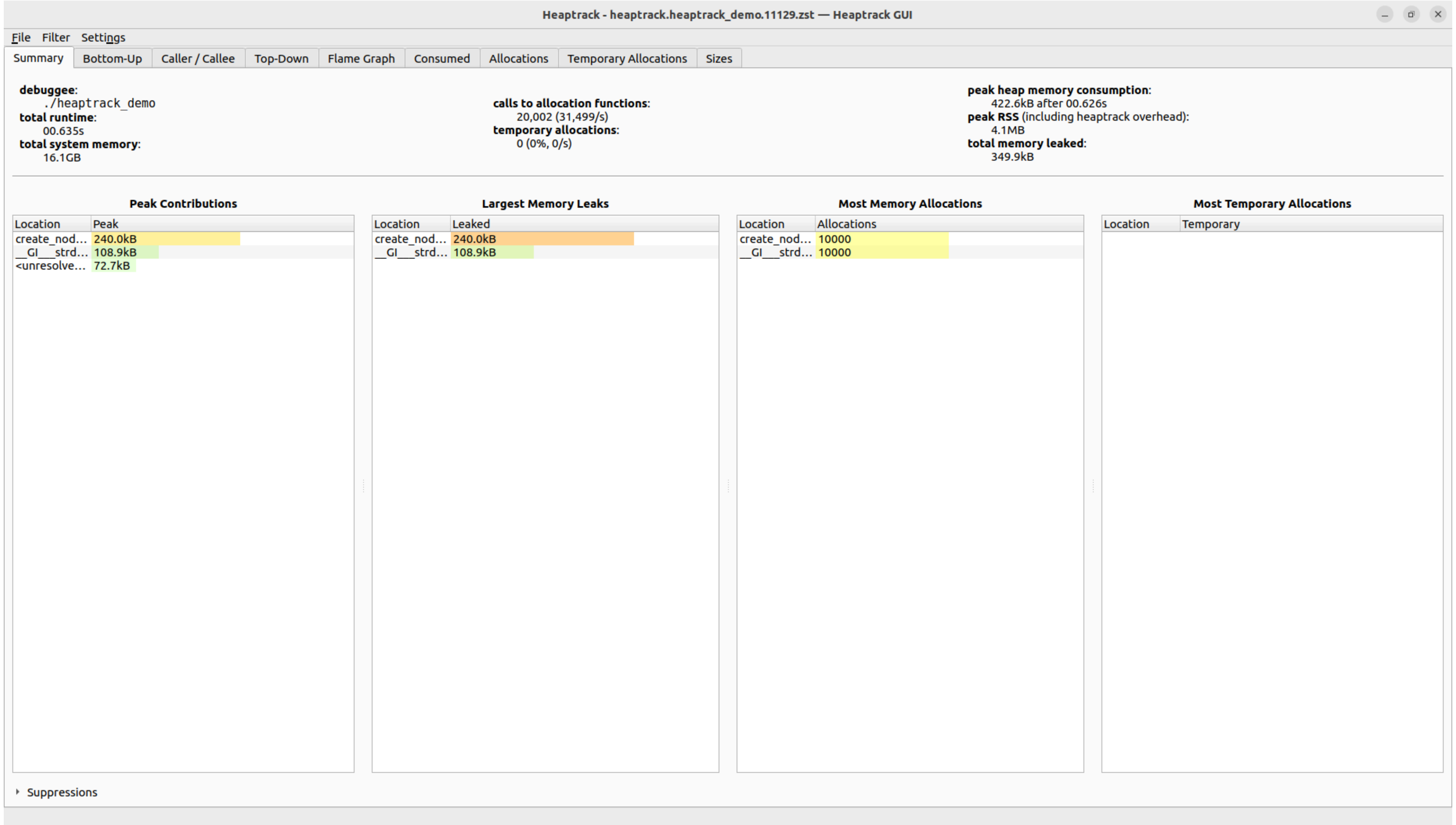
Massif Visualizer



Heaptrack

- Heaptrack traces all memory allocations and annotates them with stack traces.
- Uses `LD_PRELOAD` to intercept memory allocators.
- Analysis tools help identify hotspots for optimization, memory leaks, allocation hotspots, and temporary allocations.
- Heaptrack offers more detailed allocation context compared to Massif and incurs low overhead.
- Execute using `heaptrack {program}` .
- After execution, heaptrack generates a heaptrack.{APP}.{PID}.zst file that can be analyzed using heaptrack_print or heaptrack_gui.
- We are using heaptrack sample program for the demonstration.

Heaptrack GUI



Memusage

- memusage profiles the memory usage of user-space applications.
- It utilizes the `libmemusage.so` library by preloading it into the caller's environment via the `LD_PRELOAD` environment variable.
- libmemusage.so library traces memory allocation by intercepting memory allocators like `malloc` , `calloc` , `free` , `realloc` , `mmap` , `mremap` and `munmap` .
- The output data can be collected in a textual form, or in a PNG file for graphical representation.
- memusage is a part of `libc-devtools` package on Ubuntu.
- Demonstration uses the following memusage example.

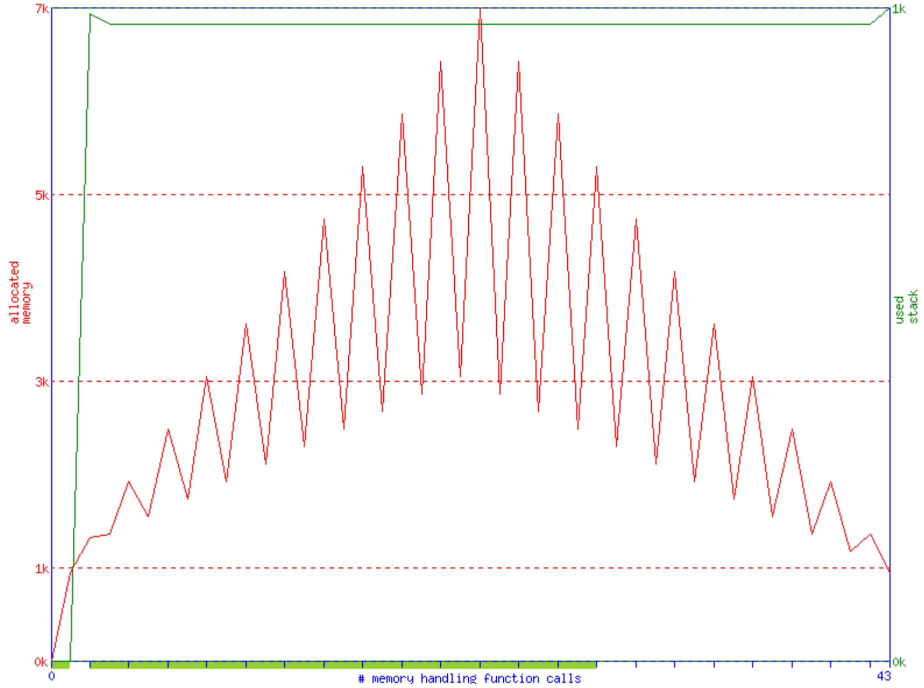
Memusage example

```
manas@sandbox:~/work/memusage$ memusage --data=memusage.dat ./userapp
---
Memory usage summary: heap total: 45464, heap peak: 7464, stack peak: 1968
         total calls   total memory   failed calls
 malloc|          2           1424              0
realloc|         40          44040              0  (nomove:37, dec:19, free:0)
 calloc|          0              0              0
   free|          1            440
Histogram for block sizes:
  240-255             1   2% ============
  400-415             1   2% ============
  432-447             3   7% =====================================
  640-655             2   4% =========================
  832-847             2   4% =========================
 1024-1039            1   2% ============
 1040-1055            4   9% ==================================================
 1232-1247            2   4% =========================
 1440-1455            2   4% =========================
 1632-1647            4   9% ==================================================
 1840-1855            2   4% =========================
 2032-2047            2   4% =========================
 2240-2255            3   7% =====================================
 2832-2847            2   4% =========================
 3440-3455            2   4% =========================
 4032-4047            2   4% =========================
 4640-4655            2   4% =========================
 5232-5247            2   4% =========================
 5840-5855            2   4% =========================
 6432-6447            1   2% ============
```

Memusage graphical representation

- Convert the collected data into a PNG format

```bash
manas@sandbox:~/work/memusage$ memusagestat memusage.dat memusage.png
```



Execution based profiling

eBPF profiler

- profile-bpfcc is an eBPF-based CPU profiler that captures stack trace samples at regular intervals.
- It helps analyze CPU usage by identifying which code is executing and measuring its impact, covering both user-space and kernel execution.
- Compared to other profilers, eBPF is efficient and low-overhead. It counts stack trace frequencies in the kernel and passes only unique stacks and their counts to user space, reducing kernel-to-user transfers.
- By default, it samples at 49 Hz across all CPUs which is adjustable via a command-line option. Frequencies like 49 Hz or 99 Hz help avoid lock-step sampling issues compared to 50 or 100 Hz.
- The profiler is a part of `bpfcc-tools` package.

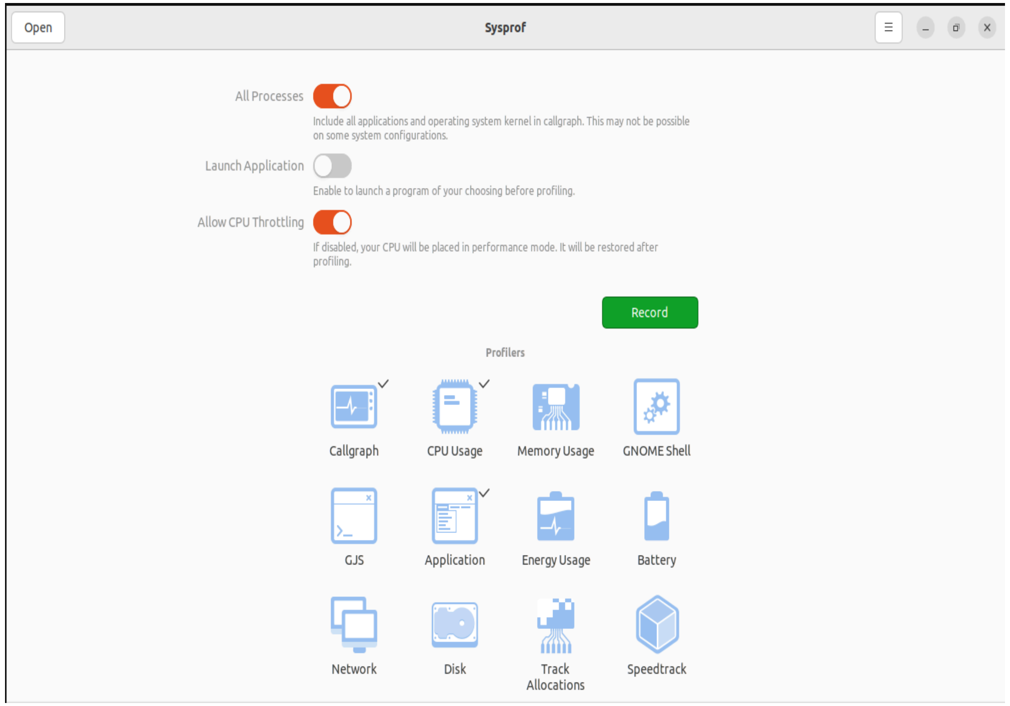
eBPF profiler usage

- The profiler can provide output in folded format directly. We can further pipe the output to `flamegraph.pl` directly to skip the intermediate file.
- The command samples at 49 Hz (-F 49), includes kernel annotations (-a), separates user and kernel stacks with a delimiter (-d), outputs in folded format (-f), and runs for 30 seconds.

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

# sudo apt-get install bpfcc-tools

# cd FlameGraph

# Profiler output piped into flamegraph.pl
# profile-bpfcc -F 49 -adf 30 | ./flamegraph.pl > profile.svg

# firefox profile.svg
```

Sysprof (1/2)

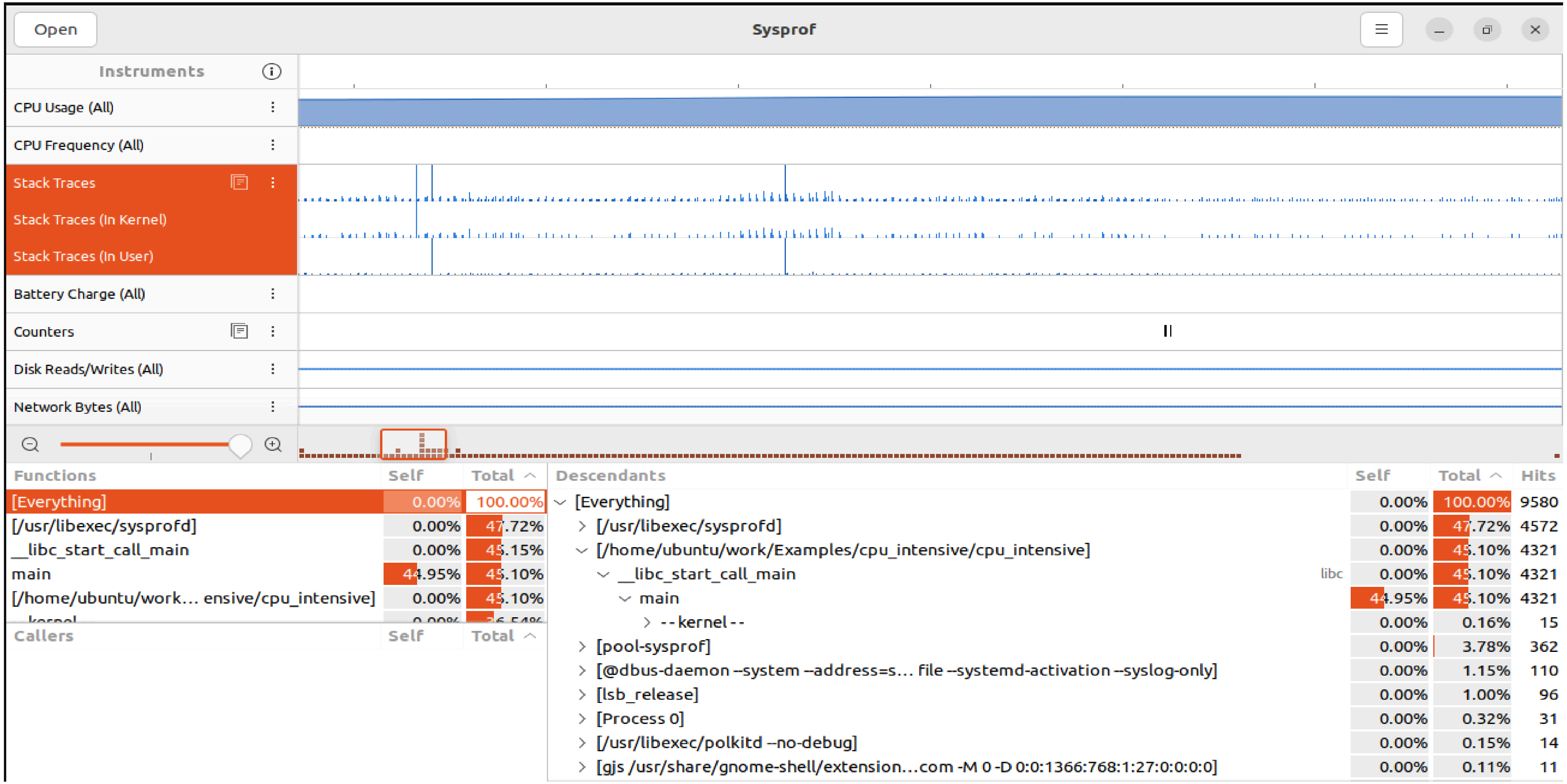
- Sysprof is a Linux sampling CPU profiler which records a stack trace of a process's activity multiple times per second.
- Sysprof is a system-wide Linux profiler that includes the kernel and all userspace applications.
- On Ubuntu platform, install it using `sudo apt install sysprof`.



Sysprof (2/2)

- Sysprof supports profiling of entire systems or specific processes. It also provides the flexibility to launch an application before recording the profile.
- Sysprof analysis includes CPU usage, thread execution, and function call stacks, helping identify performance bottlenecks and optimize applications for better performance.
- It uses frame pointer unwinding to record the stack trace. Ensure to use the `-fno-omit-frame-pointer` compiler option to instruct the compiler to store the stack frame pointer in a register.
- Following links provide detailed insight on frame pointers and working of profiler: Link-1, Link-2 and Link-3.

Sysprof example



GNU Profiler (gprof) [1/2]

- gprof is a profiling tool used to analyze the performance of user space application in terms of function call frequencies and execution times.
- It is typically used with programs written in C or C++ compiled with the `-pg` flag to generate profiling information.

```
$ gcc -o memory main.c -g -pg
```

- After compiling the program with `-pg`, execute it to generate a `gmon.out` file. Then, run gprof with the executable and gmon.out as arguments.

```
# Call graph output
$ gprof memory --graph gmon.out

# Flat Profile output
$ gprof memory -z gmon.out
```

- It produces a report showing the time spent in each function and the number of times each function was called.

GNU Profiler (gprof) [2/2]

- The generated report includes a flat profile (summary of each function) and a call graph (visual representation of function calls).
- Gprof helps identify bottlenecks and areas for optimization in the program's execution.
- It may not be suitable for multithreaded or heavily optimized programs; other tools like perf may be more appropriate in such cases.
- Refer to the gprof manual for detail on usage and limitation.

Callgrind (1/2)

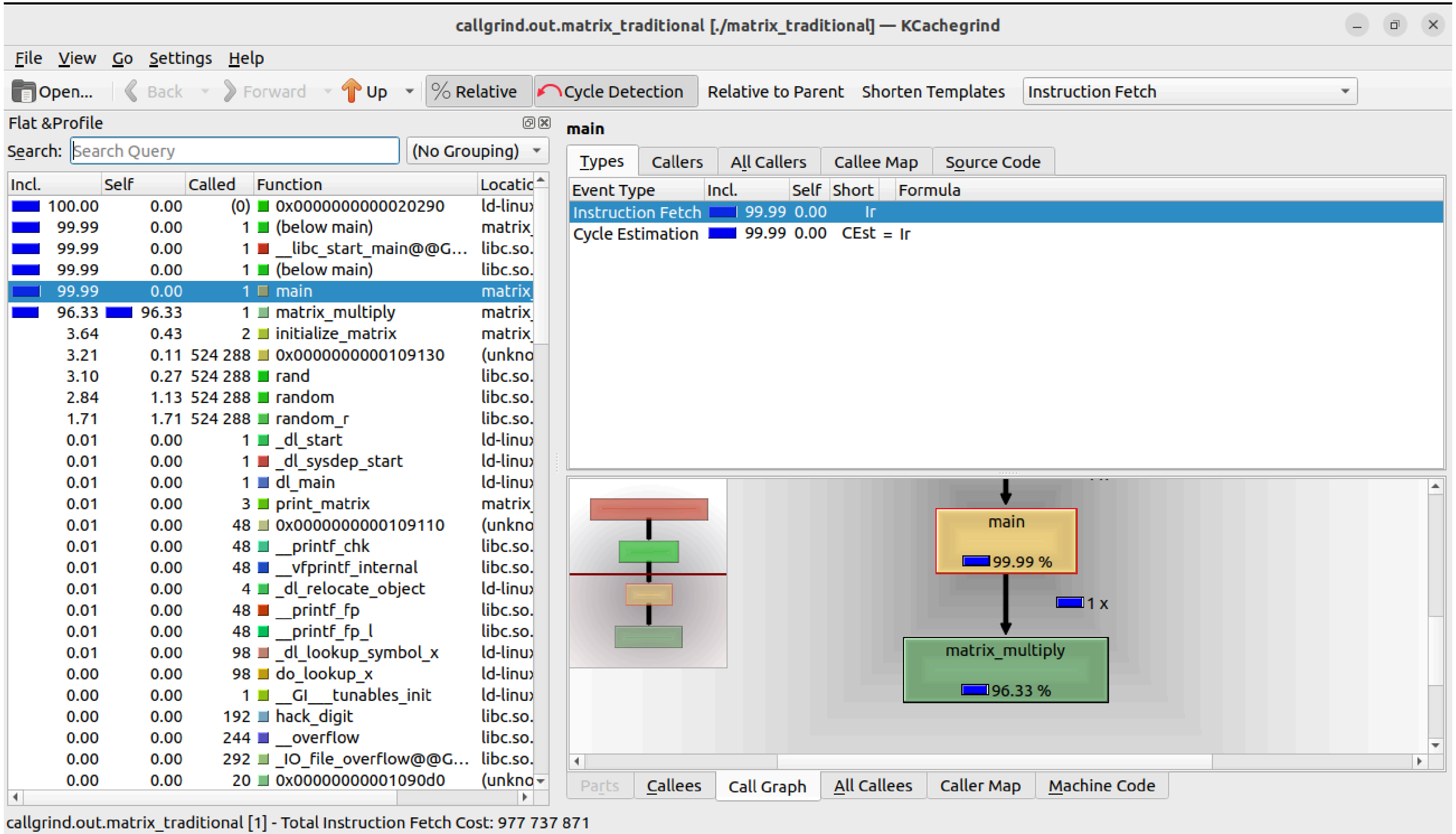
- callgrind is a part of the Valgrind tool suite which records the call history among functions in a program's run as a call-graph.
- The default collected data includes the number of executed instructions, their association with source lines, the caller-callee relationship between functions, and the counts of such calls.
- Callgrind propogates costs across function boundaries.
  - e.g: function foo calls bar, the costs from bar are added into foo's costs.
  - Provides an inclusive costs of entire program, where the cost of each function includes the costs of all functions it called, directly or indirectly.
- Callgrind usage: `valgrind --tool=callgrind [callgrind options] your-program [program options]`

Callgrind (2/2)

```
ubuntu@sandbox:~/work/Examples/matrix/$ valgrind --tool=callgrind ./matrix_traditional
==7474== Callgrind, a call-graph generating cache profiler
==7474== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==7474== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==7474== Command: ./matrix_traditional
...
==7474== Events    : Ir
==7474== Collected : 977737730
==7474==
==7474== I   refs:      977,737,730
```

- We are using the matrix multiplication example for the illustration given above.
- The detailed profiling data is written to a file named `callgrind.out.{pid}`
- `callgrind_annotate` is the command line tool which reads in the profile data, and prints a sorted lists of functions, optionally with source annotation.
- `kcachegrind` can provide the callgrind report in GUI form.
- For more details on advance usage, please refer to callgrind manual.

KCachegrind



Cachegrind (1/2)

- Cachegrind is a cache profiler tool included in the Valgrind suite and it simulates program interaction with the CPU cache hierarchy.
- Cachegrind collects precise and reproducible profiling data.
  - **Precise**: Cachegrind precisely counts program instructions and provides detailed data at file, function, and line levels.
  - **Reproducible**: Instruction counts are reliably reproducible, often perfectly so. This enables precise measurement of even minor program changes.
- The cache simulator,simulates a machine with split (Independent instruction and data cache) L1 cache with a unified L2 cache.
- For the accurate results, the program should be compiled with debugging symbols and optimization turned ON.

Cachegrind (2/2)

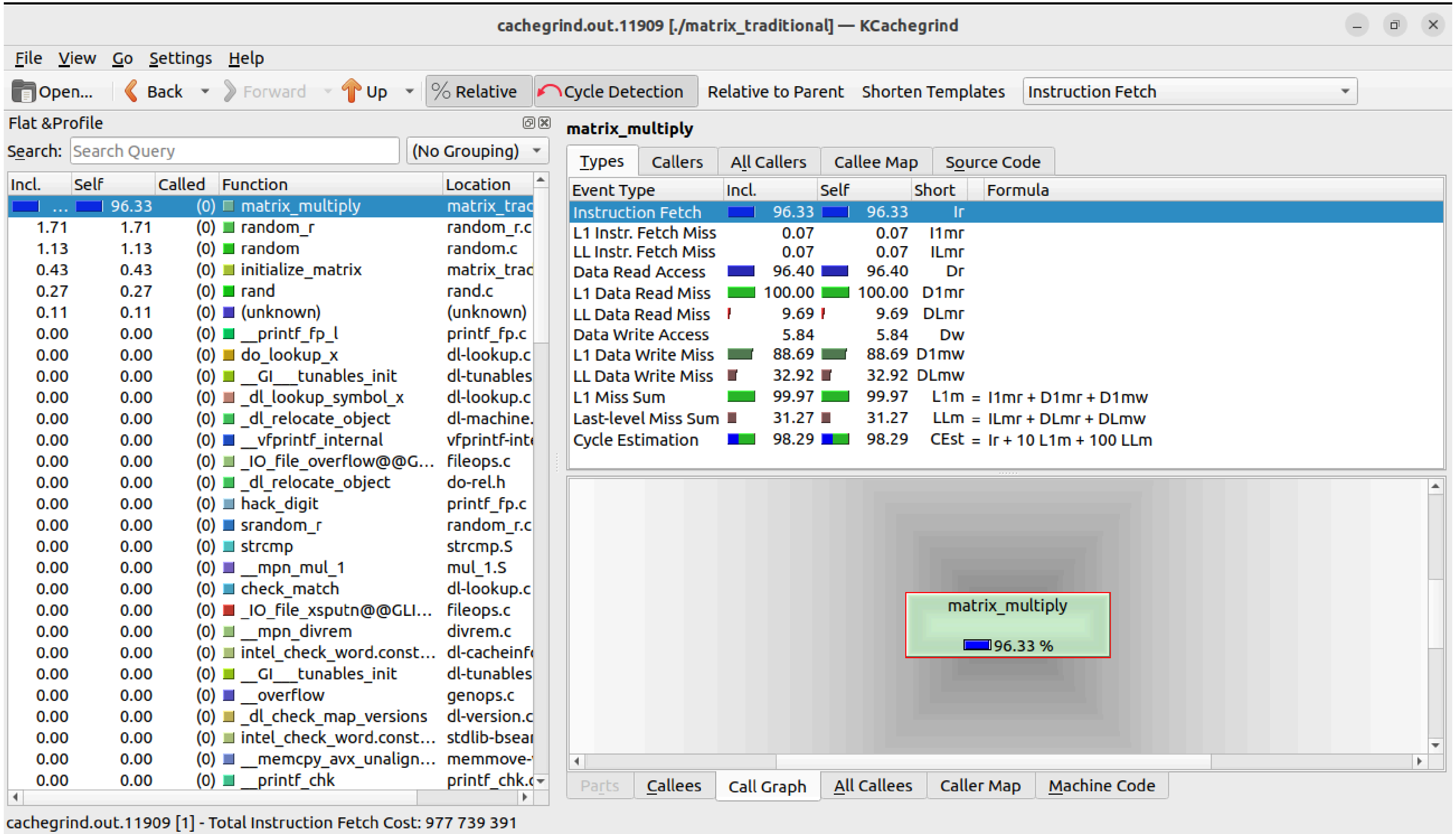
- Usage: `$ valgrind --tool=cachegrind ./program`
- The detailed profiling data is written to a file named `cachegrind.out.{pid}`
- Cachegrind output details cache misses, hit rates, and access patterns, aiding optimization with visualized hotspots.
- `cg_annotate` is a command line tool which summarizes cache-related information per function and annotates source code with color-coded markers for cache events.
- `kcachegrind` can provide the cachegrind report in GUI form.
- For more details on advance usage, please refer to cachegrind manual.

Cachegrind usage

- We are using the matrix multiplication example for the illustration given below.

```
ubuntu@sandbox:~/work/Examples/matrix$ valgrind --tool=cachegrind ./matrix_traditional
==11909== Cachegrind, a cache and branch-prediction profiler
==11909== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==11909== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==11909== Command: ./matrix_traditional
==11909==
...
==11909==
==11909== I   refs:      977,739,391
==11909== I1  misses:          1,532
==11909== LLi misses:          1,525
==11909== I1  miss rate:        0.00%
==11909== LLi miss rate:        0.00%
==11909==
==11909== D   refs:      282,943,530  (278,458,337 rd   + 4,485,193 wr)
==11909== D1  misses:    134,808,940  (134,513,355 rd   +   295,585 wr)
==11909== LLd misses:         51,364  (      1,590 rd   +    49,774 wr)
==11909== D1  miss rate:        47.6% (       48.3%     +       6.6%  )
==11909== LLd miss rate:         0.0% (        0.0%     +       1.1%  )
==11909==
==11909== LL refs:       134,810,472  (134,514,887 rd   +   295,585 wr)
==11909== LL misses:          52,889  (      3,115 rd   +    49,774 wr)
==11909== LL miss rate:          0.0% (        0.0%     +       1.1%  )
```

KCachegrind



Perf stat (1/2)

- perf stat is a Linux performance measurement tool that collects and reports detailed hardware and software performance counters such as CPU cycles, instructions, cache accesses, and branch mispredictions.
- By collecting and displaying detailed performance metrics, perf stat aids in the performance tuning and optimization of applications.
- Use the `perf list` command to view all available hardware and software events that can be monitored.

```
ubuntu@sandbox:~/work/linux-kernel/linux-6.2/tools/perf$ ./perf list
List of pre-defined events (to be used in -e or -M):
  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
...
  cgroup-switches                                    [Software event]
  context-switches OR cs                             [Software event]
...
  L1-dcache-load-misses                              [Hardware cache event]
  L1-dcache-loads                                    [Hardware cache event]
...
   cache-misses OR cpu/cache-misses/                  [Kernel PMU event]
```

Perf stat (2/2)

- `perf stat` provides options to capture specific events for a command.

```
perf stat -e cycles:uk dd if=/dev/zero of=/dev/null count=100000
```

- We can also use modifiers to count events in userspace (:u) or kernel space (:k).
- If there are more events than counters, the kernel uses multiplexing to give each event a chance to access the monitoring hardware.
- This leads the tool to scale the count and provide an estimate based on total time enabled vs time running.
- Reduce the number of events with each run to avoid scaling.
- Refer to perf wiki for more details.
- The standard Linux perf tool package offers basic support. Following these instructions, building your own version is recommended.

Perf stat example

```
root@sandbox:/home/ubuntu/work/linux-kernel/linux-6.2/tools/perf# ./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, 1.76965 s, 289 MB/s

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

          1,770.13 msec task-clock                       #    0.998 CPUs utilized
                14      context-switches                 #    7.909 /sec
                 5      cpu-migrations                   #    2.825 /sec
                80      page-faults                      #   45.194 /sec
    5,47,18,32,553      cycles                           #    3.091 GHz
    2,93,68,65,191      instructions                     #    0.54  insn per cycle
      50,06,69,465      branches                         #  282.843 M/sec
         61,40,189      branch-misses                    #    1.23% of all branches

       1.774383620 seconds time elapsed

       0.781372000 seconds user
       0.989737000 seconds sys
```

- In the example above, `perf stat` captures software events like context switches and generic hardware events like CPU cycles.
- The derived metrics such as "Instruction per cycle" are presented after the '#' sign.

CPU Frequency Profiling

Understanding CPU Frequency Scaling

- Modern CPUs dynamically adjust frequency to balance power and performance.
- Frequency is scaled based on workload demands and system policy — higher frequency = more performance, lower = more power savings.
- Linux uses scaling governors: powersave, performance and schedutil. The OS governor's job is to provide hints or requests to the underlying hardware based scalar.
- Hardware scalar (HWP) offload frequency decisions to the CPU itself. Intel Speed Shift is a prime example of HWP.
- CPU frequency impacts:
    - Response time
    - Instruction throughput
    - Power and thermal envelope

Tools for CPU Frequency Observation

- cpupower: A CLI tool to view and set CPU scaling governors or frequency limits system-wide, such as performance or powersave.
- cpufreq-info: Displays per-core scaling governor, available frequencies, driver (e.g., intel_pstate), and current policy. It is useful to verify CPU capabilities and scaling behavior.
- s-tui: An interactive terminal-based UI that monitors real-time CPU frequency, utilization, temperature, and power. It is ideal for visualizing the impact of workload on CPU scaling.
- turbostat: Provides per-core CPU MHz, Bzy_MHz (active cycles), package power (PkgWatt), and temperature (PkgTemp). It's excellent for measuring fine-grained hardware-level CPU performance under load.
- perf stat: Part of the Linux perf toolset. It tracks instructions, cycles, IPC, user/sys time, and cache events. Best suited for profiling workload efficiency and execution cost.

Demonstration: Powersave vs Performance

- **Workload**: Synthetic CPU stress test using sysbench or stress-ng.

Metric	Expectation in Powersave	Expectation in Performance
CPU Frequency	Low, dynamic	High, locked at max
Instructions	Similar	Similar
Cycles	Higher (slower)	Lower (faster)
IPC	Lower	Higher
Total CPU Time	Longer	Shorter
Power/Temp	Lower	Higher

- Set scaling governer:
  - **Performance**: `sudo cpupower frequency-set -g performance`
  - **Powersave**: `sudo cpupower frequency-set -g powersave`
- Refer to this helper script to dump the CPU frequency results.

Demonstration: Key observation

- Higher and more stable frequencies in performance mode.
- Lower CPU time, higher IPC in perf stat.
- Higher Avg_MHz, Bzy_MHz and higher temparature in turbostat.
- CPU governor has a significant impact on workload efficiency.
    - `performance` mode reduces runtime by keeping frequency high
    - `powersave` mode may hurt latency-sensitive tasks
- Use frequency profiling to:
    - Optimize performance vs power trade-offs
    - Debug unexpected slowdowns in real-world deployments

References (1/2)

- Histogram - Statquest
  - https://www.youtube.com/watch?v=qBigTkBLU6g
- Flamegraph -Brendan Gregg
  - https://www.youtube.com/watch?v=VMpTU15rIZY
- eBPF - Brendan Gregg
  - https://www.youtube.com/watch?v=HKQR7wVapgk
- sysprof
  - https://fedoramagazine.org/performance-profiling-in-fedora-linux/
- Debugging with Frame pointers
  - https://developers.redhat.com/articles/2023/07/31/frame-pointers-untangling-unwinding#
  - https://blogs.oracle.com/linux/post/unwinding-stack-frame-pointers-and-orc

References (2/2)

- Perf tool building instructions - Manas Marawaha.
  - https://medium.com/@manas.marwah/building-perf-tool-fc838f084f71
- SIMD and AVX2 instruction set
  - https://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX2
  - https://en.ittrip.xyz/c-language/c-matrix-simd-optimization
  - https://v0dro.in/blog/2018/05/01/building-a-fast-matrix-multiplication-algorithm/