Technische Universität Dresden

Christian von Elm — CIDS — 50. VI-HPS Tuning Workshop

An Introduction to lo2s

31. August 2026 • GWDG Göttingen

Introduction

Linux

- Built around `perf_event_open` and other Linux kernel interfaces

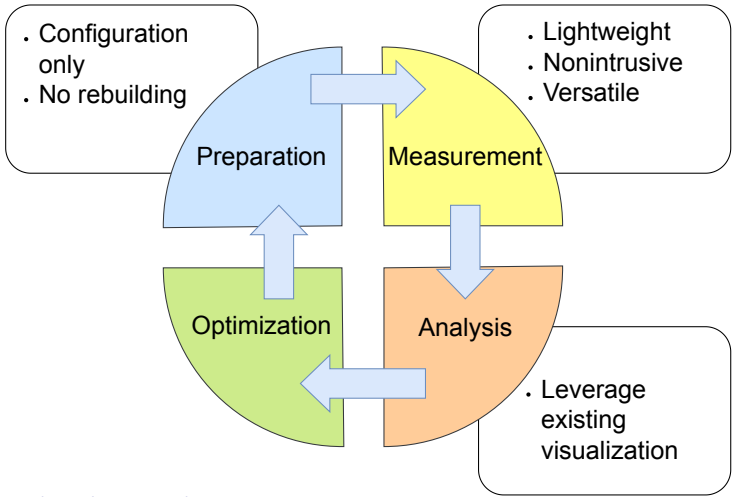
OTF-**2**

- Stores traces in OTF-2 format

Sampling

- Relies on sampling, not instrumentation

Why lo2s?


• Configuration only
• No rebuilding
• Lightweight
• Nonintrusive
• Versatile
Preparation
Measurement
Optimization
Analysis
• Leverage existing visualization

Sampling vs. Instrumentation

- Works with unmodified executables
  - lo2s only relies on HW methods that do not touch the program at all
  - Pro: Due to HW use, it can be very fast
  - Pro: Can record the whole system
  - Con: Beholden to HW and Kernel support for interfaces
  - Con: If we can record everything, a lot of measurements are not available to the average joe
- CPU periodically record instruction pointer while program is running
- Sampling is a statistical process, thus statistical errors (especially bias) come into play

Sampling Artefacts

C-Code:

```
int a, b;

void  foo()
{
    a++;
}

void bar()
{
    b++;
}

int main()
{
    for (int i = 0; i++; i < 100)
    {
        foo();
        bar();
    }
}
```

Sampling Artefacts

C-Code:

```c
int a, b;

void  foo()
{
    a++;
}

void bar()
{
    b++;
}

int main()
{
    for (int i = 0; i++; i < 100)
    {
        foo();
        bar();
    }
}
```

Assembly (shortened):

```
foo:
 mov    0x0(%rip),%eax
    R_X86_64_PC32 a-0x4
 add    $0x1,%eax
 mov    %eax,0x0(%rip)
    R_X86_64_PC32 a-0x4
bar:
 mov    0x0(%rip),%eax
    R_X86_64_PC32 b-0x4
 add    $0x1,%eax
 mov    %eax,0x0(%rip)
    R_X86_64_PC32 b-0x4
main:
# for([...]) {
call   42 <main+0x16>
   R_X86_64_PLT32 foo-0x4
call   47 <main+0x1b>
   R_X86_64_PLT32 bar-0x4
# }
```

Sampling Artefacts

C-Code:

```c
int a, b;

void  foo()
{
    a++;
}

void bar()
{
    b++;
}

int main()
{
    for (int i = 0; i++; i < 100)
    {
        foo();
        bar();
    }
}
```

Assembly (shortened):

```
foo:
 mov    0x0(%rip),%eax
    R_X86_64_PC32 a-0x4
 add    $0x1,%eax
 mov    %eax,0x0(%rip)
    R_X86_64_PC32 a-0x4
bar:
 mov    0x0(%rip),%eax
    R_X86_64_PC32 b-0x4 # SAMPLE
 add    $0x1,%eax
 mov    %eax,0x0(%rip)
    R_X86_64_PC32 b-0x4
main:
# for([...]) {
call   42 <main+0x16>
   R_X86_64_PLT32 foo-0x4
call   47 <main+0x1b>
   R_X86_64_PLT32 bar-0x4
# }
```

Trace:

Time	Function
5	bar()
15	bar()
25	bar()
35	bar()
45	bar()
55	bar()
65	bar()

Sampling Artefacts

C-Code:

```c
int a, b;

void  foo()
{
    a++;
}

void bar()
{
    b++;
}

int main()
{
    for (int i = 0; i++; i < 100)
    {
        foo();
        bar();
    }
}
```

Assembly (shortened):

```
foo:
 mov    0x0(%rip),%eax
    R_X86_64_PC32 a-0x4
 add    $0x1,%eax # SAMPLE
 mov    %eax,0x0(%rip)
    R_X86_64_PC32 a-0x4
bar:
 mov    0x0(%rip),%eax
    R_X86_64_PC32 b-0x4
 add    $0x1,%eax
 mov    %eax,0x0(%rip)
    R_X86_64_PC32 b-0x4
main:
# for([...]) {
call   42 <main+0x16>
   R_X86_64_PLT32 foo-0x4
call   47 <main+0x1b>
   R_X86_64_PLT32 bar-0x4
# }
```

Trace:

Time	Function
10	`foo()`
20	`foo()`
30	`foo()`
40	`foo()`
50	`foo()`
60	`foo()`
70	`foo()`

Sampling Artefacts

- This is highly unlikely under normal circumstances
- Less pronounced sampling biases, however, *can* occur
- Default sampling rate is a large prime (11010113)

The Catch: Permissions

- Whole-system measurements mean that *theoretically* you can look at everything
  - Other users programs, kernel internals, system daemons
- Wide open door for security issues
- For security reasons, the following prevent a user from using all of `lo2s`:
  - `perf_event_paranoid`
  - `/sys/kernel/tracing`
  - `root`
- For every example, I will give the required permissions

Permissions: `perf_event_paranoid`

- Governs how much of `perf` the average user can use
- `sudo sysctl kernel.perf_event_paranoid=$LEVEL`

Level	Permissions
2	Only user-space measurements ← *the default*
1	Kernel and userspace measurements
0	Everything but tracepoints ← *nice HPC-admins will set this for exclusive jobs*
-1	Free-for-all

Permissions: /sys/kernel/tracing

- Tracepoints give details of the kernels execution (e.g. `sched:sched_switch`)
- Tracepoints are controlled through `/sys/kernel/tracing`, which is usually owned by `root`
- `chown -R $user /sys/kernel/tracing`
- Tracepoints also require `perf_event_paranoid=-1`

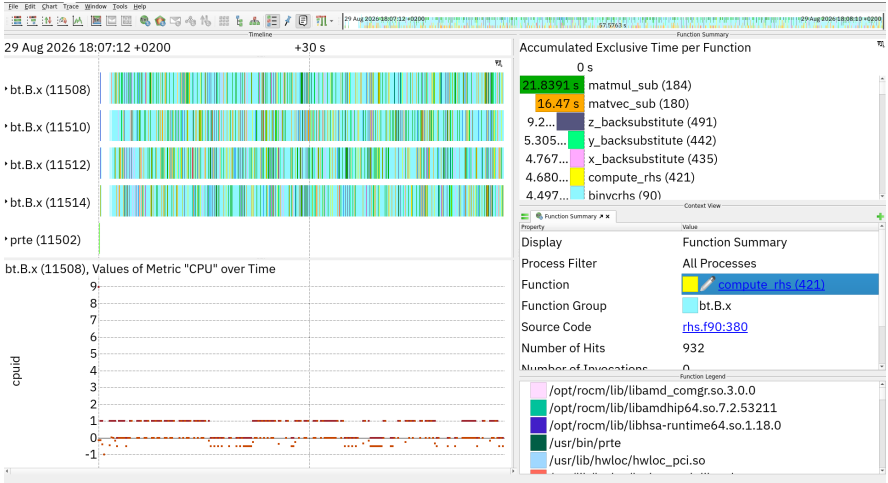
Permissions: `root`

- `root` is allowed everything
- BPF based measurement interfaces (I/O) are *only* available to `root` :(
  - We are thinking about mitigations for that
- Some of these measurement interfaces are interesting to admins

Working with OTF-2

- You ran lo2s and you have an OTF-2 trace, now what?
- To main processing steps I use: Vampir and OTF-2 Python bindings
- Vampir
  - Program for the visualization of traces
  - This is sadly not free, but your institution might already own it
- OTF-2 Python bindings
  - Python library to read OTF-2 traces.
  - Shipped with OTF-2 library.
  - Ideal for ingesting trace into Pandas, for example

Working with OTF-2: Vampir



Working with OTF-2: Python

```
import otf2

with otf2.reader.open("somewhere/traces.otf2") as trace:
    for location, event in trace.events:
        if location.name == "hwmon1":
            print("Got a hwmon reading: at time " + event.time +
                  " value: " event.values[0])
```

Tour of `lo2s` measurement options

Per-Process and Per-CPU View

- `lo2s` has two major modes: per-CPU and per-process.
- **per-process**: Measurement points are recorded from the POV of a process
- **per-CPU**: Measurement points are recorded from the POV of a CPU
- per-process requires less permissions
- per-CPU allows for interesting Kernel↔Daemons↔Application interaction traces
  - We had a case where the real performance problem was lock-contention in the kernel

Basic `lo2s` Workflow: Measuring Per-Process and Per-CPU

Per-Process Mode: `lo2s -- ./bt.B.x`

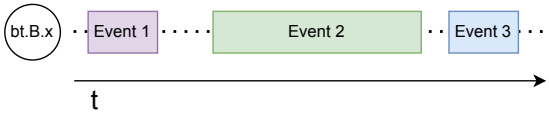
- `lo2s` will start `./bt.B.x` and record events from it and all of its children until its death.
- Ctrl-C will stop the program-under-test but not abort the trace!

Per-CPU-Mode: `lo2s -A -- [./bt.B.x]`

- `lo2s` will start recording CPU-wide events.
- Either until Ctrl-C is pressed or until the given program exits

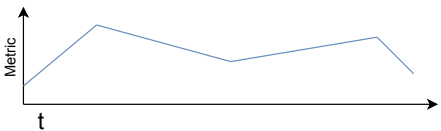
Overview of all measurement interfaces

Events



- `perf` samples
- `syscalls`
- POSIX I/O
- Block I/O
- AMD/NVIDIA GPU kernels
- OpenMP pragmas

Metrics



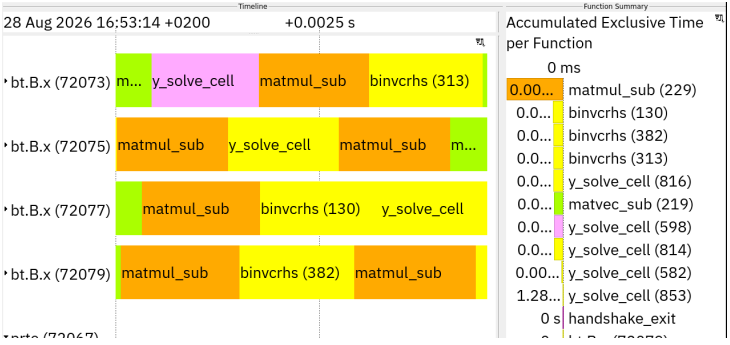
- `perf` metrics
- `lm_sensors` sensors
- Kernel tracepoints
- Score-P Plugins

Sampling – Process Mode

```
lo2s -- ./bt.B.x
```

Records Program Activity.

`perf_event_paranoid=2`



Sampling – System Mode (`lo2s -A`)

```
lo2s -A -- ./bt.B.x
```

Records Global CPU Activity.

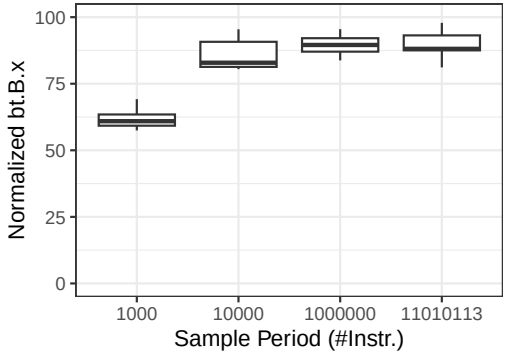
```
perf_event_paranoid=0
```

Sampling Rate (`lo2s -c $SAMPLE_RATE`)

```
lo2s -c 50000 -- ./bt.B.x
```

Record a sample after `$SAMPLE_RATE` events
Lower rate ⟶ more samples
Lower rate ⟶ more overhead

```
perf_event_paranoid=2
```



Sampling Event (`lo2s -e $EVENT`)

```
lo2s -e instructions -- ./bt.B.x
```

Sets the interrupt generator event
Non-standard events can adaptively increase resolution in more interesting sections

```
perf_event_paranoid=2
```

Some interesting events:

Event	Description
`dummy`	No interrupts
`instructions`	#Instructions (default)
`cpu-clock`	Time
`cache-misses`	Cache Misses

List All `perf` Events (`lo2s --list-events`)

`lo2s --list-events`

lo2s can make use of:
- predefined
- `perf` in-built
- `libpfm4`

events.

`perf_event_paranoid=2`

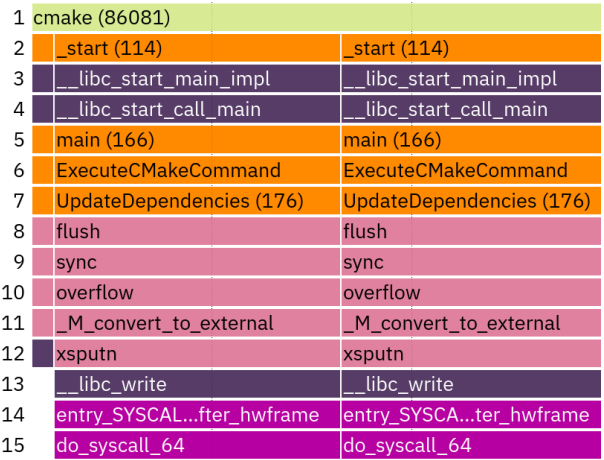
```
List of predefined events:
  cache-misses
  cache-references
  cpu-cycles
[...]
List of Kernel PMU events:
  msr/irperf/
  msr/mperf/
  msr/tsc/
  power/energy-pkg/ # [ CPU 0 ]
[...]
List of Libpfm events:
  amd64_fam19h_zen3::ALLOC_MAB_COUNT
  amd64_fam19h_zen3::DECODER_OVERRIDE_BRANCH_PRED
[...]
List of perf pmu-events:
  cpu::bp_l1_tlb_fetch_hit
  cpu::bp_l1_tlb_fetch_hit.if1g
  cpu::bp_l1_tlb_fetch_hit.if2m
  cpu::bp_l1_tlb_fetch_hit.if4k
[...]
```

Callstacks (`lo2s -g`)

```
lo2s -g [-A] -- ./bt.B.x
```

Samples the callstack.
Works in both process and system mode

```
perf_event_paranoid=2
```



CMake trace, because `bt.B.x` callstack is really boring

Python Sampling (`lo2s -g`)

`lo2s -g [-A] -- python fib.py`
Example: Naïve fibonacci with nice deep callstack
Hardware sampling of python applications. Requires callstacks (`-q`) to be useful.
`perf_event_paranoid=2`

```python
# fib.py
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1)
           + fibonacci(n-2)

print(fibonacci(100))
```

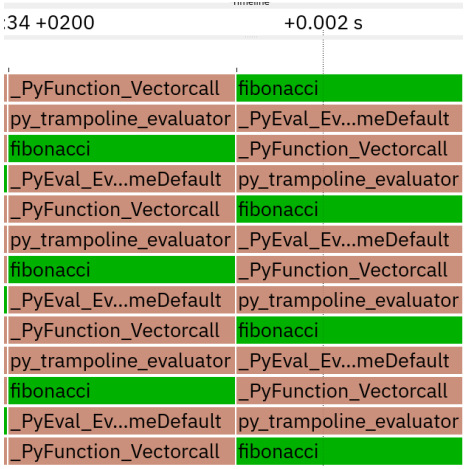
Python Sampling (`lo2s -g`)

```
lo2s -g [-A] -- python fib.py
```

Example: Naïve fibonacci with nice deep callstack

Hardware sampling of python applications.
Requires callstacks (`-g`) to be useful.

```
perf_event_paranoid=2
```

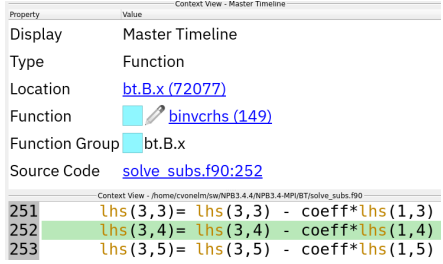


Symbol resolution and DWARF

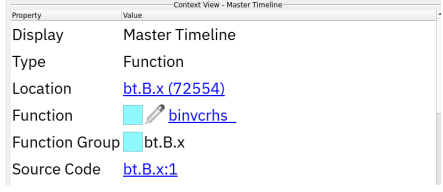
To get the name of functions lo2s uses:

- function symbols (always available)
- DWARF debug information (compile with `-g`)
- lo2s now supports `debuginfod`
- `lo2s --dwarf=full` → automatic DWARF download (like GDB)

With DWARF:

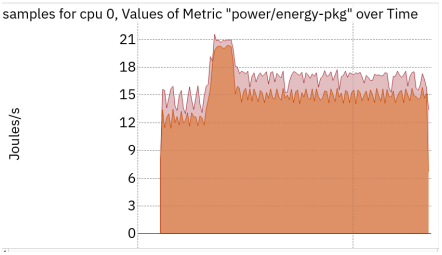


Without DWARF:



perf Metrics

`lo2s -E power/energy_pkg`
Records a timeline of a `perf` metric.
Example: Measure `power/energy_pkg` to estimate energy consumption.
per-PID: `perf_event_paranoid=2` per-CPU: `perf_event_paranoid=0`



perf Metric Leader (lo2s --metric-leader $EVENT --metric-count $N)

`lo2s --metric-leader cache-misses --metric-count 100000 -- ./bt.B.x`
Example: Record metrics every 100000 cache misses
Sets the metric read-out interrupt generator. Metrics are recorded every `$N` events. Same caveats as for the sampling event.
`perf_event_paranoid=2`

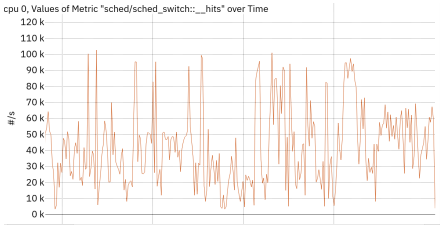
Tracepoints (`lo2s -t`)

`lo2s -t sched:sched_switch`

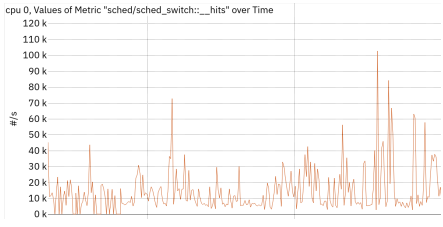
Records kernel tracepoints.

Example: Amount of scheduler switches.
More switches → worse contention

`perf_event_paranoid=-1`,
`/sys/kernel/tracing`



Oversubscribed `bt.B.x`



Normal `bt.B.x`

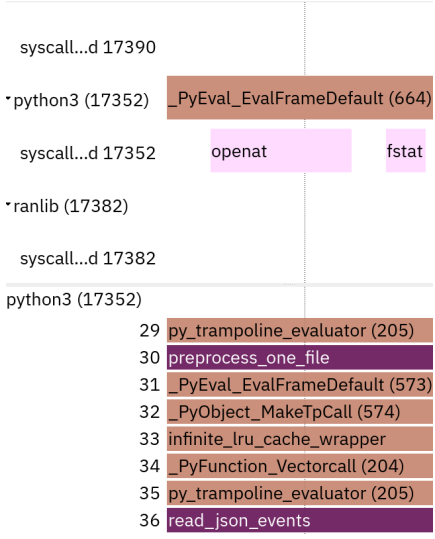
Syscall tracing (`lo2s -s ($SYSCALL_NAME/all)`)

```
lo2s -s all -- python
```

Example: All syscall trace from the `jevents.py` event list generator

Records a trace of the syscalls in the application. `all` to record all syscalls. (Huge overhead!)

`perf_event_paranoid=-1` `/sys/kernel/tracing` access



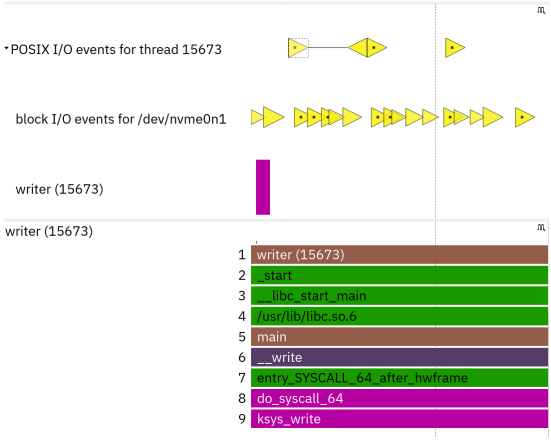
Block I/O and POSIX I/O (`lo2s --posix-io --block-io`)

`lo2s --posix-io`

Example: Program that writes 1GiB of data to a file, with and without O_DIRECT

Amount of scheduler switches. More switches: worse contention

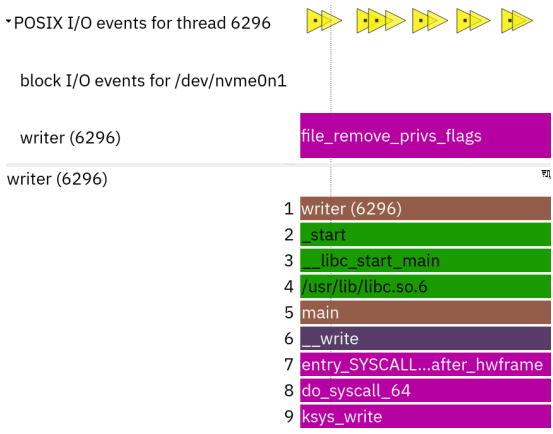
`root`



With O_DIRECT

Block I/O and POSIX I/O (`lo2s --posix-io --block-io`)

`lo2s --posix-io`
Example: Program that writes 1GiB of data to a file, with and without O_DIRECT
Amount of scheduler switches. More switches: worse contention
`root`



Without O_DIRECT

...and many, many more

Curious?

```
$ lo2s --help
$ man lo2s
```

Obtaining lo2s

- `https://github.com/tud-zih-energy/lo2s` (GPLv3 Open-Source Software)
- *Required:* Linux ($\geq$4.3), OTF-2, `elfutils`, CMake, C++17 compiler
- `cmake`, `make`, and you are off to the races
- Have an idea, comment or bug-report? Leave me a GitHub issue!

Hands-On

`$PROJECT_DIR/software/lo2s` contains a `lo2s` compiled for medium96s

- Or build it locally and enjoy the full `perf_event_paranoid=-1` experience!

Examples to (possibly) try:

- The above directory contains a stock-standard NPB BT class B. Try to sample it with lo2s
- the above directory contains `matmul_ijk.c` and `matmul_ikj.c`, why is `matmul_ikj.c` faster?
  - You probably already know the solution but how can lo2s help us discover it?
- Above all, try your own software and ideas!