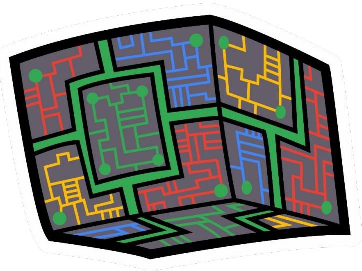
# Scalability of performance monitoring

Stephane Eranian
Rohit Jnagal
Ian Rogers

Linux Plumbers Conference 2019
Lisbon, Portugal

Confidential + Proprietary

# Perf use-cases at Google



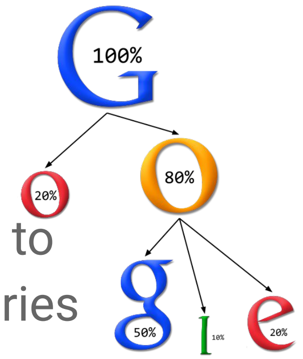
## Borg : Google Cluster Mgmt

Multi-tenant mix-workload environment

Running 2B+ of containers every week.

Collect perf data for all tasks:

- **Counting based** per-cgroup collection
- Used for scheduling & isolation
- Published for debugging & analysis



## Google-wide Profiling

Continuous fleet profiler to identify hot fns and binaries and visualize resource utilization trends.

- **Sampling based** per-machine collection
- Interested in h/w counters & profiles.
- Built for Observability

## Perf Data usage for Cluster Management

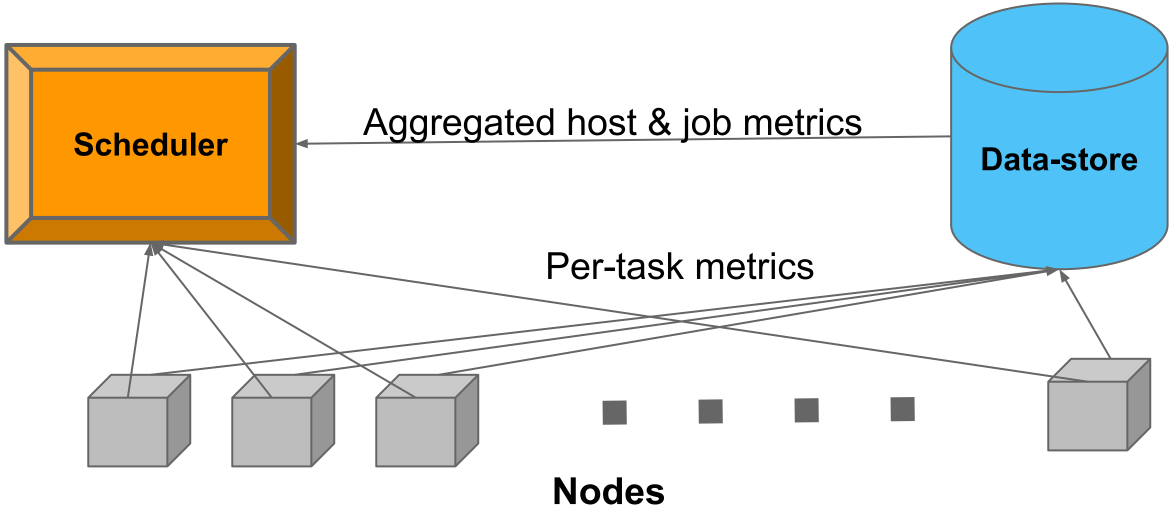
Perf SLOs

Antagonist Detection

Scheduling Decisions

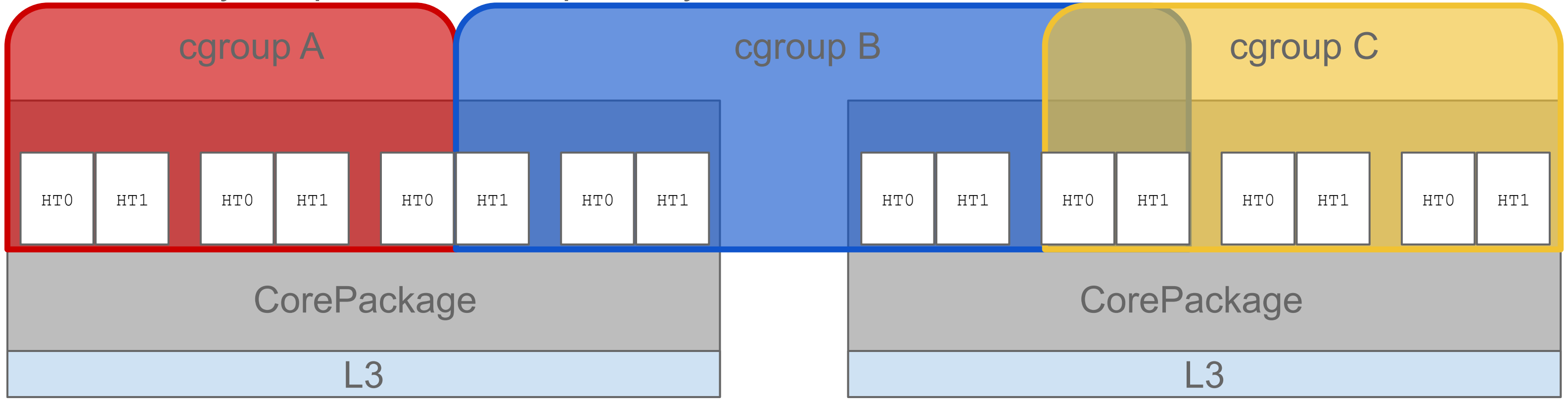
Hot Functions & Binaries

S/W & H/W Design



## Google workloads runs in cgroups

- Any workload runs inside a cgroup
- System daemons (incl. monitoring) run in a cgroup with limited memory
- Bigger systems → more cores → more jobs
  - Memory is  for apps not system daemons (data center tax)
- `perf` tool and `perf_events` scaling become a serious issue
  - memory footprint, file descriptors, cycles of overhead



## Perf Collection for a Borg Node

- Run perf in counting mode.
- Cgroup-based profiling.
- Bounded amount of memory used for profiling.
  - Want to measure X events on Y cpus for Z cgroups
- Run perf for few seconds every 5 mins
- Select subset of cgroups to profile every run.
- Also run separate collectors for uncore events.
- Use results to detect isolation issues or other machine health signals :
  - Take Node-level actions
  - Pass unresolved issues to scheduler system
- Push new samples to a time-series DB. Available to service owners.

Overheads

Interference & staleness

Google

# Perf collection : Counting-mode Issues

- Scalability issues on single machines as they get bigger
  - Overheads : For counting in cgroup mode,  needs { #events * #cpus * #cgroups } file descriptors.
  - Staleness : Limiting number of cgroups scanned causes blind spots.
  - Interference : Frequent save-restore on context switches
  - Accuracy : Multiple events need to be multiplexed (more events than counters) causing inaccuracies.

**Borg expects fresh and accurate samples for all jobs every 5 mins at very low cost.**

**Perf overhead bounded to 1-2%. Services should not observe any blips.**

# Perf Collection for Google Wide Profiling

- Run perf in sampling mode.
- Machine-wide profiling.
- Attribute events back to processes and cgroups.
    - Capture pids and IP when the events were being collected.
- Run perf on a small subset of machines every day
- All running cgroups would be profiled.
- Also collects traces and symbols to generate per-function view.
- Aggregate results from many machines over multiple days to :
    - Generate fleetwide trends for functions and binaries
    - Monitor and improve DC tax.
- Push new samples to a time-series DB. Available to service owners.

Inaccuracies

Data overload

Google

## Perf at scale : Sampling-mode Issues

- Scalability issues on single machines as they get bigger
  - Processes get recycled or lost : up to 15% samples not matched to the parent task.
  - Larger data dumps of records to process and parse.
- Large fleet → exposes all sorts of problems
  - Race conditions
  - Corner cases
- Fleet can tolerate errors - no staleness issues.
- Perf causing resources overcommit, slowdown, crashes is **not** acceptable.

## Deeper Look at Perf-events

# Perf record: MMAP scanning

- More cores → more jobs → bigger `/proc/PID`
  - More time spent scanning `/proc/PID`
  - More jobs submitted = more dynamic `/proc/PID`
- Intel SkylakeX 112 CPUs idle: 1300 PIDs
  - **90%** are kernel threads which are **IRRELEVANT** for `perf`
- 3000-20000 processes configs may exist
- `perf record` has no parallel scanning
  - Google planning to contribute: `perf record` parallel and optimized scan

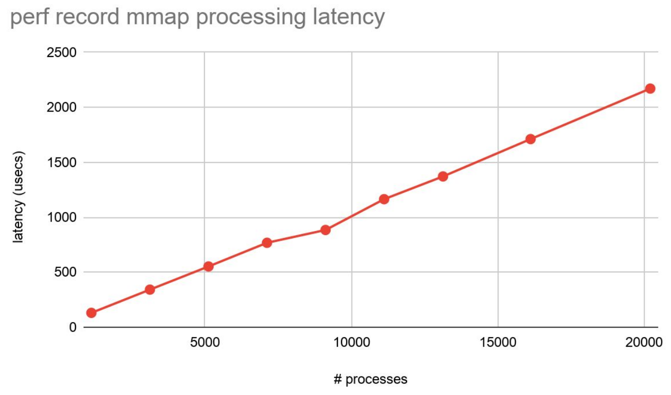
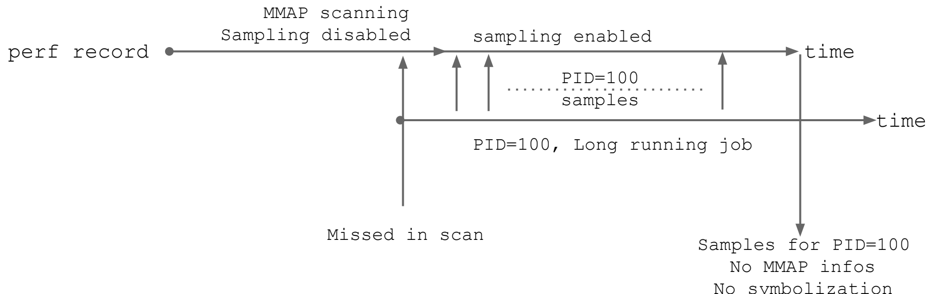


Chart from Intel SkylakeX, perf record -a -e cycles sleep 1, timed __perf_event__synthesize_threads()

Google

## Perf record: MMAP race condition

- More cores → more time spent scanning `/proc/PID`
  - While jobs are launched and die
- Race condition: scan vs. creation of new PIDs
  - Death of PIDs is not a problem because can always symbolize
- Google seeing **10%** unsymbolized samples
  - Missed PID → Missed `RECORD_MMAP` → no symbolization possible
- Multithreading does not help with race window (single snapshot of `/proc/PID`)
  - Do not control how `/proc/PID/maps` is generated in the kernel



Google

## Perf record: MMAP race condition solution

- Solution is to track MMAP from start of `perf record`
- Use the `DUMMY` event
    - Does not track any actual event
- Aggregate all `RECORD_MMAP` to `DUMMY`, remove from all other events
    - Used by BPF (Berkeley Packet Filter)[1] support even when no BPF event, but no MMAP tracking
- Problem remains: rate of `mmap()` system-wide
    - `DUMMY` sampling buffer may run out of space but no polling until all events enabled
    - Must poll `DUMMY's` buffer **while** scanning `/proc/PID` → Create `DUMMY`-dedicated thread
- Google planning to contribute patch to add `DUMMY mmap()` tracking
    - Does not cover buffer polling (yet)

[1] based on Linux-5.3-rc3

Google

## Perf record: pipe mode

- Google uses `perf record` in **pipe mode** only
  - No sampling data touches the disk
  - Single threaded, single output: the pipe!
- More cores = more sampling buffers to process
  - Multi-thread processing of buffer overflow
  - Affinity of threads to NUMA node to limit remote memory traffic (Alexey's patches)
  - But still 100+ threads writing to single pipe!
- Pipe mode does not benefit from multi output changes by Jiri
  - Maybe, we could create a meta-data pipe and a sample pipe streams
- Why maintain two `perf.data` formats: file vs. pipe?
  - Pipe mode receives less testing, Google found many issues
  - Standardize on pipe mode format

## Perf cgroup mode: too many file descriptors

- Large number of file descriptors (fds): 1 fd/event/cpu/cgroup
  - 100 cgroups, SkylakeX (112 CPUs), 6 events/cgroup = 112 x 100 x 6 = **67,200** fds
  - 200 cgroups, AMD Rome (256 CPUs), 6 events/cgroup = 200 x 256 x 6 = **307,200** fds
- Large number of events per-cpu:
  - 100 cgroups, SkylakeX (112 CPUs), 6 events/cgroup = 100 x 6 = **600** events/CPU
  - 200 cgroups. AMD Rome (256 CPUs), 6 events/cgroup = 256 x 6 = **1536** events/CPU

| Structure names | Size (bytes) | Intel SkylakeX Total size (bytes) | AMD Rome Total size (bytes) |
|---|---:|---:|---:|
| `struct file` | 256 | 17MB | 78MB |
| `struct perf_event` | 1136 | 76MB | 348MB |
| Source: Linux-5.3-rc3, pahole | TOTAL<br>4KB Pages | 93MB<br>22,705 | 427MB<br>104,400 |

## Perf cgroup mode : how to minimize number of file descriptors?

**num_fds = #events x #cgroups x #cpus**

- Must eliminate 1 or more factor of the equation
- Avoid creating the same events for each cgroup
- A new cgroup type event (eliminates #cgroups)
  - System-wide event not tied to a specific cgroup
  - Maintains counts per cgroup internally on context switches
  - Use `lseek()` on `read()` to extract counts using cgroup identifier (e.g., cgroup inode) as offset

Google

## Perf cgroup mode: context switch too expensive

- Users complain about performance overhead whenever cgroup mode is on
    - Typically workloads with high context switch rates
    - **Even when simply using counting mode**
- Cgroup mode: system-wide but must save/restore PMU state if cgroup changes
- Cgroup mode context switches are very expensive (Ian's work, next slides)
    - Must find events for incoming **cgroup hierarchy**
    - May have to scan all events in the RB tree. Worst case #events/cgroup < #num counters
- **2** options:
    - 1: Make cgroup context switch **cheaper**: better event management and scheduling
    - 2: Use system-wide sampling and reconstruct counts

## Options & Experiments

# Perf cgroup: use sampling for cgroup counting!

- Use system-wide sampling
  - **+** No overhead on context switch
  - **-** Interrupt, memory, post-processing overheads
  - **-** need way to reconstruct counts **per cgroup**
- Namhyung's cgroup tag/sample patches (Namhyung's patches)
  - Each sample tagged with cgroup identifier (`u64`)
  - Reconstruct counts = #samples x period x scaling (no IP, only cgroup tag + eventID + period)
  - Cgroup id →job easy to retrieve vs. buildid
- **not as accurate** as counting mode for workloads with phases
  - Hard to tweak period + duration for accuracy
- Must find a way to optimize common case: same events across all cgroups

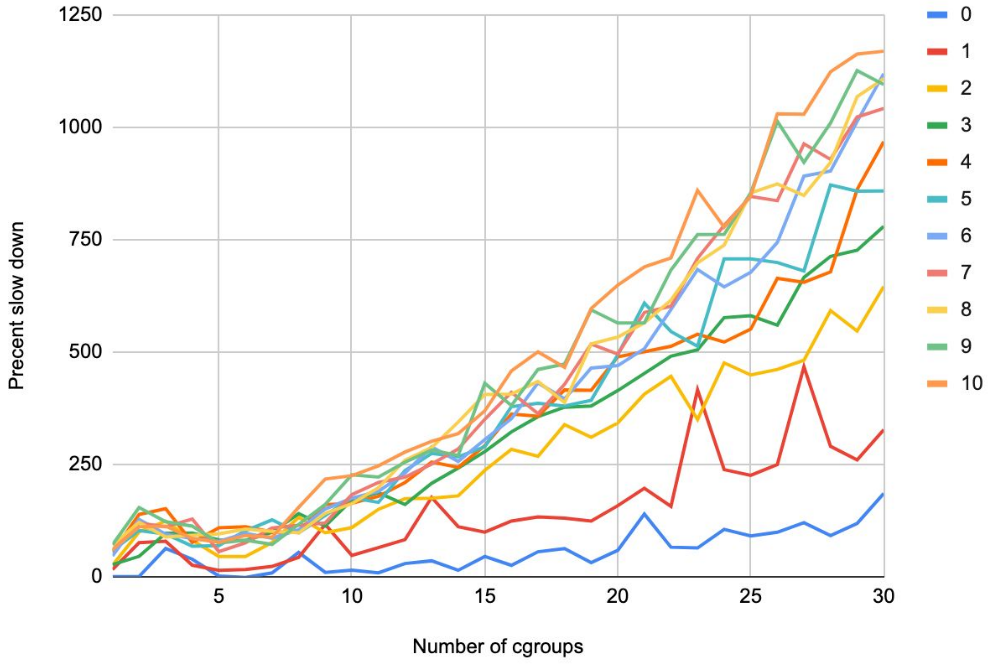
Google

Namhyung's patches: https://lkml.org/lkml/2019/8/28/127

# Scheduling overhead - what does it look like?

- Context switch must schedule out existing events and in new events, unless running in system wide mode.
- For cgroups a switch of events is only necessary when going between cgroups.
- All events are visited to build a transaction of active events which are then committed to the PMU.

Overhead from additional cgroups and perf events on Skylake



Google

## Scheduling overhead improvements - event rotation

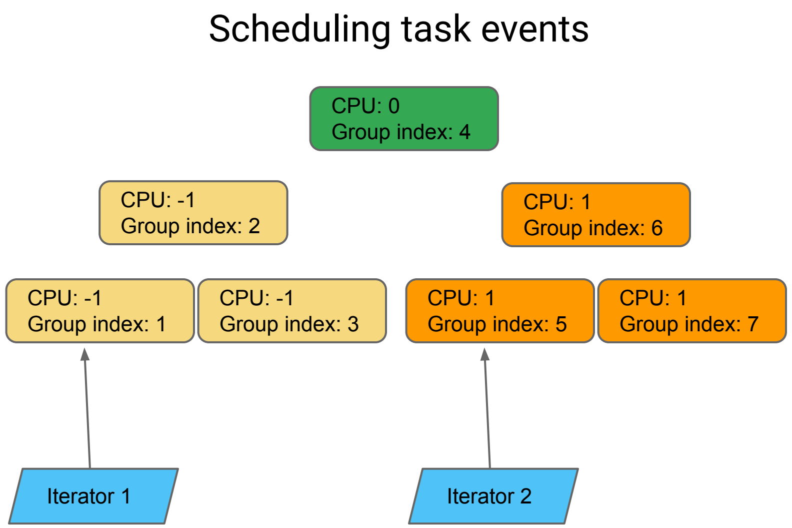
Multiplexing of events is done by a high precision timer removing the first active event and adding it to the end of event list.

Multiplexing was considered necessary if the number of active events was less than the number of events on a CPU. This is always true with cgroups, as we don't wish to schedule events from a different cgroup, which leads to unnecessary multiplexing.

Fixed to only consider multiplexing when events fail to be scheduled in:

https://lkml.kernel.org/r/20190601082722.44543-1-irogers@google.com

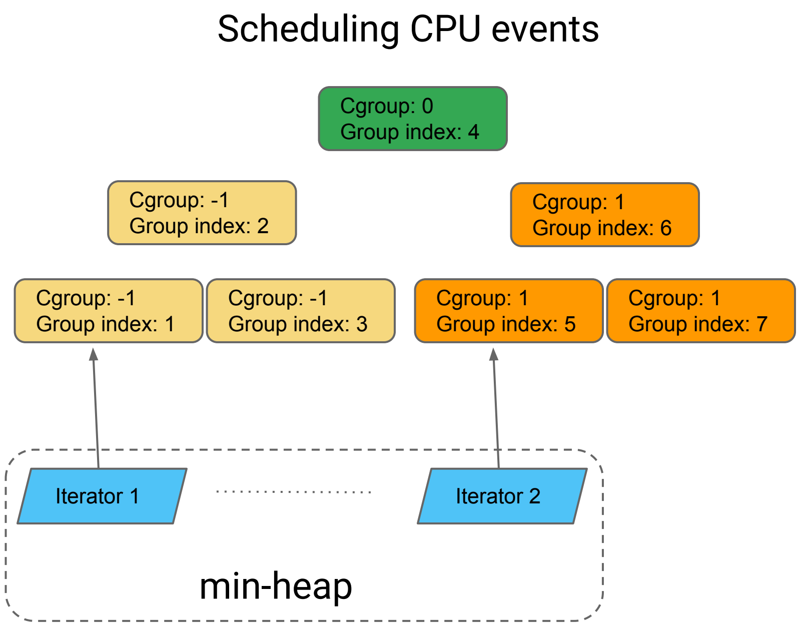
## Scheduling overhead improvements - organize events by cgroup



Since Linux 4.17 events are held in RB trees ordered by CPU and a time of insertion index. The insertion index is necessary to facilitate rotation.

Two iterators are used for scheduling task events. One iterator is for global events, while the other is per CPU. Events are scheduled in the order of their insertion (aka group) index.

# Scheduling overhead improvements - organize events by cgroup



Cgroups are hierarchical, need an iterator for the cgroup and its parents that are being scheduled in.

Min-heap avoids searching all iterators for the lowest group index on each iteration.

Min-heap needs to be sized large enough for maximum cgroup depth, sized when an event is created.

For 2 events and 30 “flat” cgroups (645% slow down on Skylake), go from processing 60 events per context switch to 2.

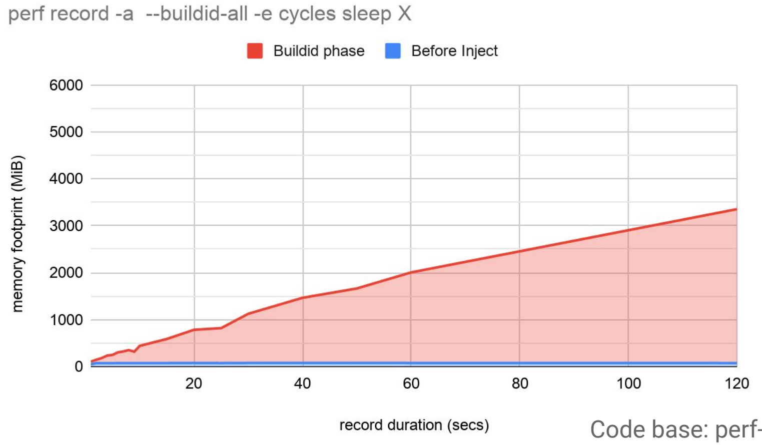
Patch set up for review, much thanks to Kan Liang at Intel.
https://lkml.org/lkml/2019/7/2/84

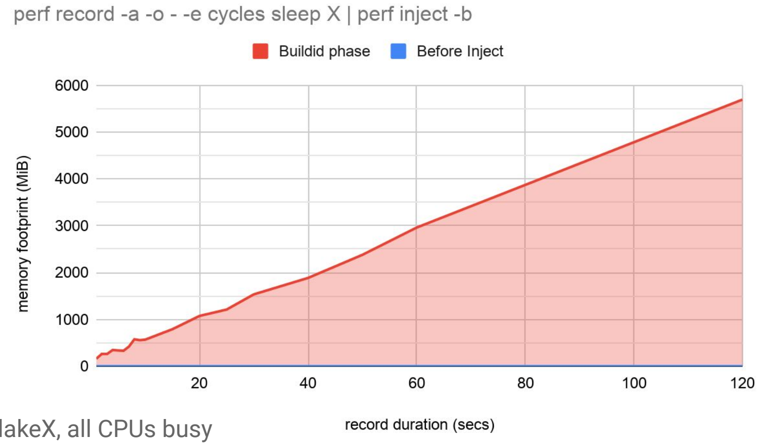
Performance improvement is in the region of 5%.

Google

## Perf record/inject: memory footprint during BuildID collection

- Bigger machines → more samples/time unit
- `perf record/perf inject -b` consume a lot of RSS memory
- Memory/time  consumption mostly during  BuildID collection
- File mode: single `perf.data` mmap, all pages touched
- Pipe mode:





Code base: perf-5.3-rc7, SkylakeX, all CPUs busy

Google

## Perf record/inject: collecting BuildID differently

- Injection of BuildID very big consumer of cycles and memory
  - Could they be collected differently?
- Save meta-data separately from samples (out-of-band)
  - Timestamps used to synchronize
- Maybe cheaper to generate BuildID for all executable MMAPS (`--buildid-all`)
- Could the kernel help collect BuildID more efficiently?
  - BuildID needed for any executable module (binaryh, shared libs) → `PERF_RECORD_MMAP`
  - But would force `perf_events` code to read ELF on `mmap()`, likely too high overhead.
- Out-of-Bound data stream likely the right approach

## Conclusions

- At scale, you will see many race conditions and corner cases
- Bigger machines → more algorithmic pressure on `perf_events` and `perf` tool
- Need to improve cgroup mode to reduce file descriptor pressure
- Need to improve event scheduling to avoid repeating operations unnecessarily
- Need to make perf tool scalable: multi-thread, affinity, multi-output, memory
- Google is contributing to the effort
- Google is hiring developers in this field!