Profiling Postgres Perils

Andres Freund
PostgreSQL Developer & Committer
Email: andres@anarazel.de
Email: andres.freund@microsoft.com

https://anarazel.de/talks/2026-05-21-pgconf-pgconf-profiling-postgres-perils/profiling-postgres-perils.pdf

Why Profile?

- Improve Performance
  - Measure & Hack & Compare & Repeat
  - Tiny changes are important
- Understand Production Problems
  - See where time is spent, verify that’s sensible
  - Does not need to be precise

Performance Counters

- Limited number of counters (~4 per core)
- Events like cycles, cache misses,
- Absolute counts (perf stat)
- Interrupt at overflow, instruction pointer at the overflow (perf record & perf report)

Pitfalls: Profiling Cycles: Contention

```
BEGIN;
LOCK pgbench_accounts;
SELECT SUM(abalance) FROM pgbench_accounts;
COMMIT;
```

1 Client
5.793 tps

Samples: 114K of event 'cycles:P', Event count (approx.): 89534446706

Overhead	Command	Shared Object	Symbol
19.60%	postgres	postgres	[.] tts_buffer_heap_getsomeattrs
17.42%	postgres	postgres	[.] ExecInterpExpr
13.07%	postgres	postgres	[.] ExecSeqScan
8.74%	postgres	postgres	[.] heapgettup_pagemode
5.86%	postgres	postgres	[.] ExecStoreBufferHeapTuple
5.86%	postgres	postgres	[.] heap_prepare_pagescan
4.73%	postgres	postgres	[.] heap_getnextslot
4.34%	postgres	postgres	[.] fetch_input_tuple
4.13%	postgres	postgres	[.] ExecAgg
2.23%	postgres	postgres	[.] hash_search_with_hash_value
1.86%	postgres	postgres	[.] MemoryContextReset

8 Clients
5.919 tps

Samples: 130K of event 'cycles:P', Event count (approx.): 101421444476

Overhead	Command	Shared Object	Symbol
19.66%	postgres	postgres	[.] tts_buffer_heap_getsomeattrs
17.69%	postgres	postgres	[.] ExecInterpExpr
12.72%	postgres	postgres	[.] ExecSeqScan
8.78%	postgres	postgres	[.] heapgettup_pagemode
5.96%	postgres	postgres	[.] heap_prepare_pagescan
5.58%	postgres	postgres	[.] ExecStoreBufferHeapTuple
4.61%	postgres	postgres	[.] heap_getnextslot
4.26%	postgres	postgres	[.] fetch_input_tuple
4.23%	postgres	postgres	[.] ExecAgg
2.28%	postgres	postgres	[.] hash_search_with_hash_value
1.83%	postgres	postgres	[.] MemoryContextReset

Pitfall: Profiling Cycles: Contention

- Alternatives:
  - Use offcpu profiling
  - Look at wait events and attach with perf probe or trace events

Pitfall: Profiling Cycles: Other CPU Bottleneck

- Look at perf stat -ddd <cmd/-p/-a>
- Common other bottlenecks
  - Memory Latency (write code that hides latency, improve locality)
  - Data TLB misses (use huge pages)
  - Instruction TLB misses (make code more local, try to use huge pages for code)
  - Branch misses (reduce branches, hint branch likelihoods)
- Look at “Top Down” Methodology on Intel HW

Pitfall: Profiling Cycles: Not measuring cycles

- Virtualization can hide performance counters
- Perf can “emulate” cycles profiling
  - regular timer fires, captures current IP
- Can be very inaccurate, particularly with lots of context switches

```
Samples: 19K of event 'cycles:P', Event count (approx.): 12822501037
  Overhead  Command          Shared Object            Symbol
+    2.16%  swapper          [kernel.kallsyms]        [k] enqueue_task_fair
+    2.07%  postgres         [kernel.kallsyms]        [k] dequeue_entities
+    1.97%  postgres         postgres                 [.] _bt_compare
     1.87%  swapper          [kernel.kallsyms]        [k] update_load_avg
+    1.66%  swapper          [kernel.kallsyms]        [k] acpi_processor_ffh_cstate_enter
     1.41%  postgres         postgres                 [.] LockBufferInternal
+    1.31%  postgres         postgres                 [.] heap_page_prune_opt
     1.20%  postgres         postgres                 [.] PostgresMain
     1.19%  postgres         [kernel.kallsyms]        [k] unix_stream_read_generic
     1.15%  postgres         [vdso]                   [.] __vdso_gettimeofday
+    1.11%  postgres         libc.so.6                [.] __memmove_avx512_unaligned_erms
     1.08%  postgres         [kernel.kallsyms]        [k] __schedule
```

```
Samples: 20K of event 'task-clock:P', Event count (approx.): 5031750000
  Overhead  Command          Shared Object               Symbol
+   16.15%  swapper          [kernel.kallsyms]           [k] cpuidle_enter_state
+   10.22%  swapper          [kernel.kallsyms]           [k] finish_task_switch.isra.0
+    5.51%  postgres         postgres                    [.] _bt_compare
+    5.47%  postgres         [kernel.kallsyms]           [k] finish_task_switch.isra.0
+    5.20%  swapper          [kernel.kallsyms]           [k] flush_smp_call_function_queue
+    3.55%  postgres         postgres                    [.] hash_search_with_hash_value
+    2.61%  postgres         [kernel.kallsyms]           [k] _raw_spin_unlock_irqrestore
+    1.59%  postgres         [kernel.kallsyms]           [k] do_syscall_64
+    1.16%  postgres         libc.so.6                   [.] __memmove_avx512_unaligned_erms
+    1.11%  postgres         libc.so.6                   [.] __internal_syscall_cancel
+    1.09%  postgres         postgres                    [.] AllocSetAlloc
```

Pitfall: Repeated / Long runs

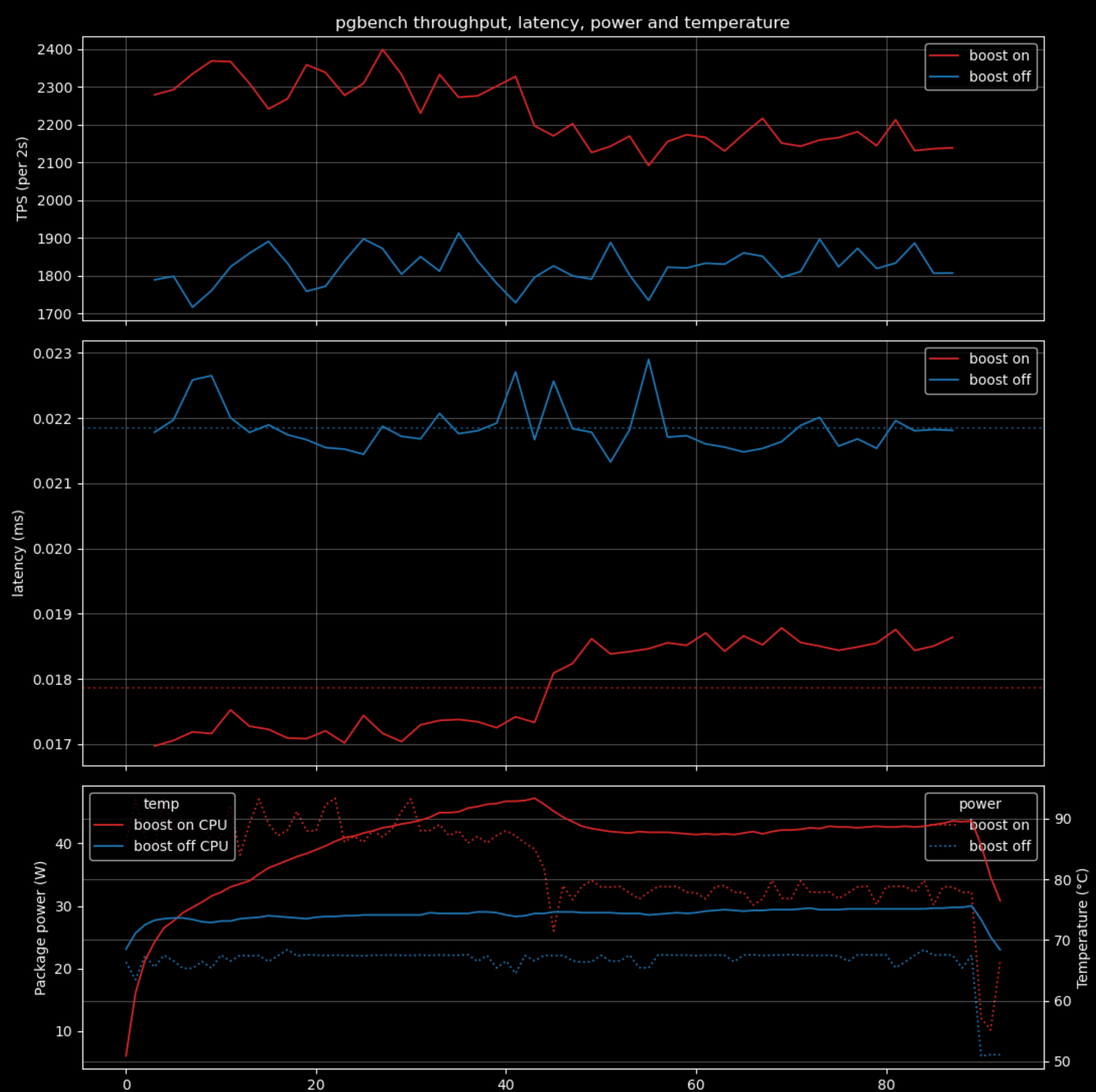
- How do you expect the TPS of this to look?

```
pgbench -Mprepared -S -c$c -j$c -T${duration}
```

- Common benchmark pattern:
  1) run baseline
  2) run patched
  3) compare
  4) hack
  5) repeat

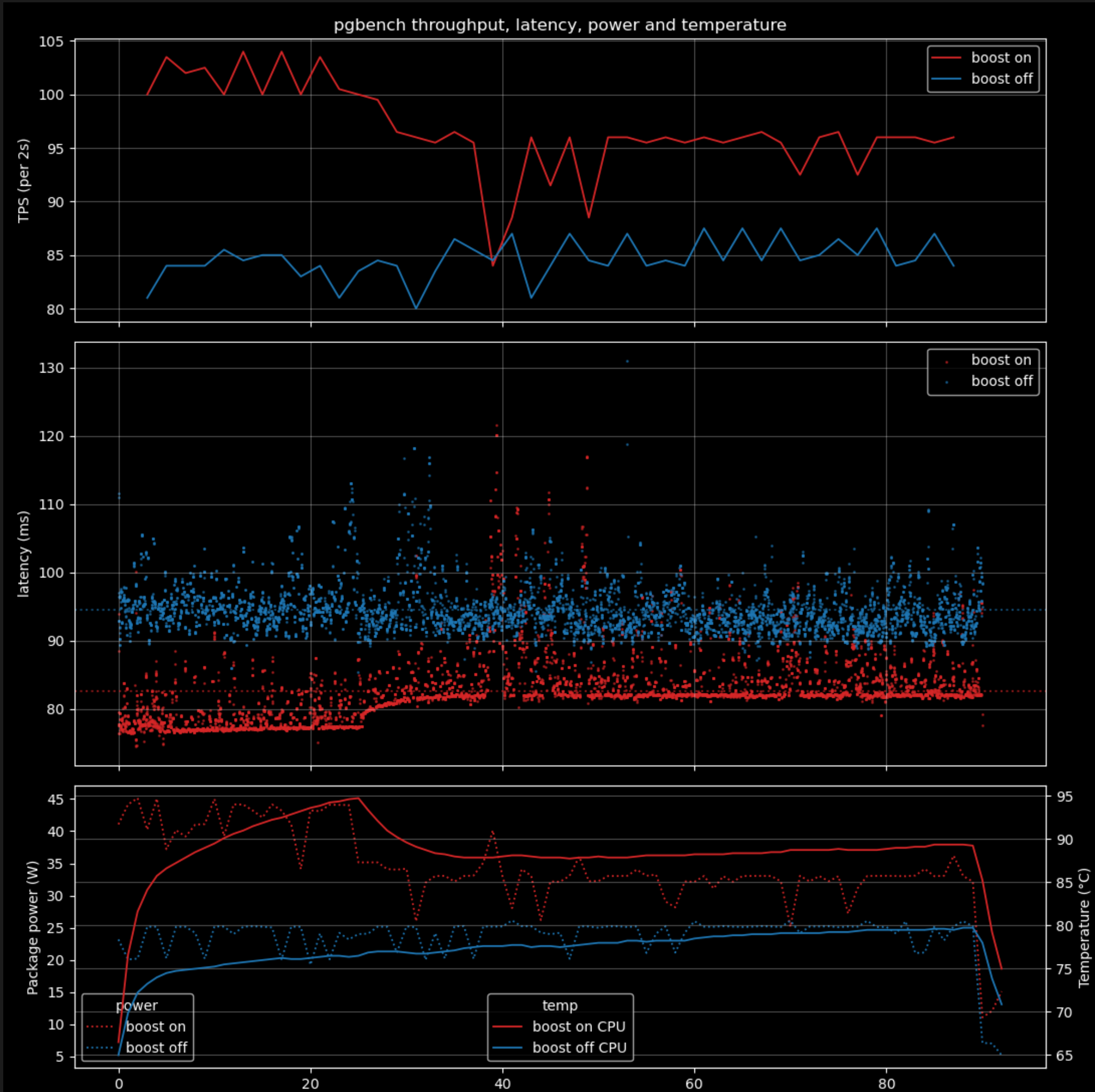
Workload:

- pgbench RO
- sampled, aggregated latencies
- 8 out 8 cores / 16 threads
- Laptop (but same is true on servers)



Workload:

- SELECT SUM(abalance) FROM pgbench_accounts
- 8 out 8 cores / 16 threads
- Laptop (but same is true on servers)



Repeated / Long runs: Workarounds

- Disable Boost
  - Slower but predictable / repeatable performance
  - Lower power usage / lower temps

DO NOT DO THIS IN PRODUCTION!

  - Intel:

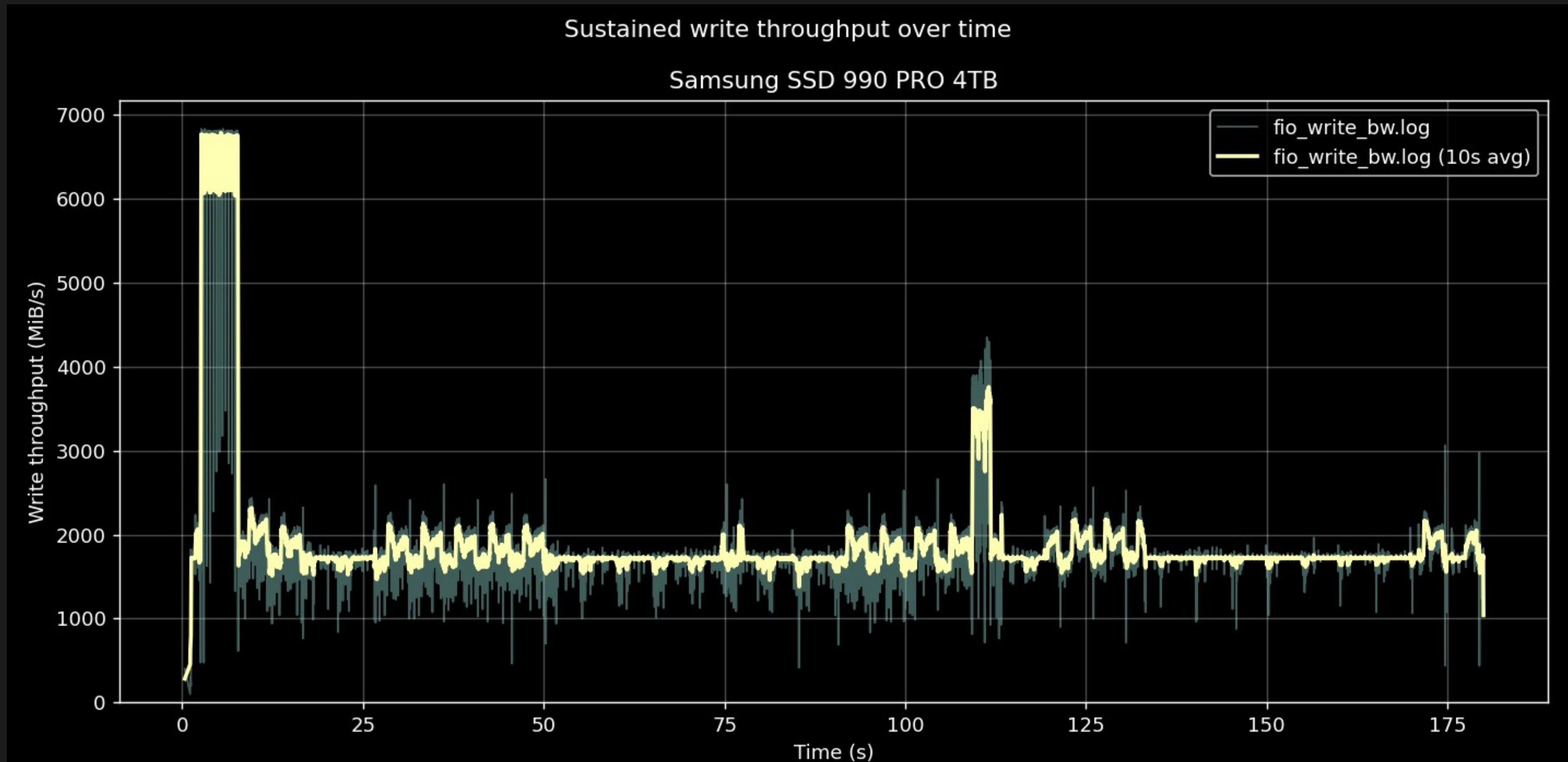
    ```
    echo 1 > /sys/devices/system/cpu/intel_pstate/no_turbo
    ```

  - AMD:

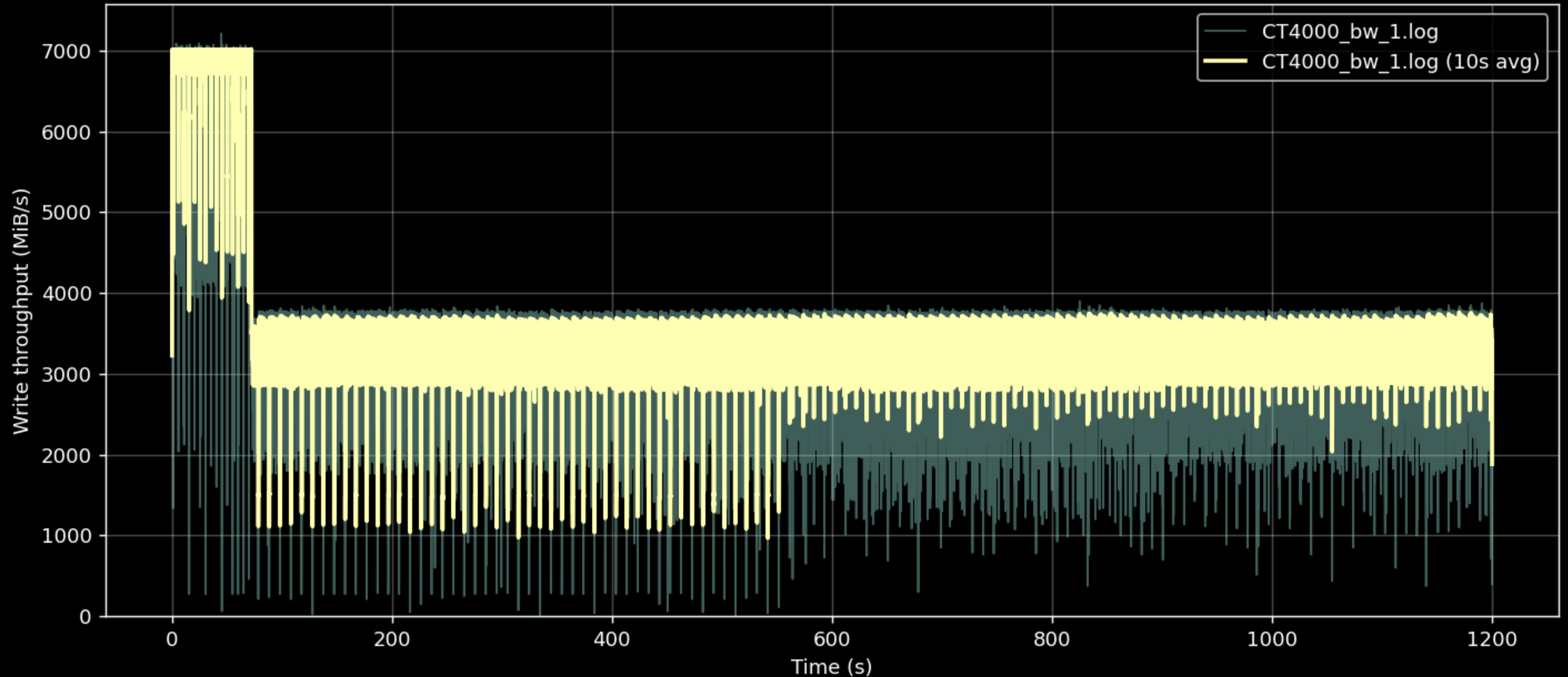
    ```
    echo 0 > /sys/devices/system/cpu/cpufreq/boost
    ```

- `Swap order of benchmarks`
- `Wait between runs (10s of seconds)`

Repeated / Long write heavy runs



Repeated / Long write heavy runs



Repeated / Long write heavy runs: Workarounds

- Overprovision SSDs
- Trim between runs: `fstrim -v /path/to/fs`
- Use small proportion of fast “amount”
- Use better SSDs
- Wait between runs (minutes to hours)

Pitfall: Cores are not created equal

- Performance differences
  - different boost speeds (I’ve seen up to 15%)
  - different types of cores (“efficiency” and “performance”)
  - different distances to memory etc
  - some cores get more interrupts
- Workarounds:
  - Pin to subset of cores
    - numactl –physcpubind 1,17 <task>
  - Disable boosting

Pitfall: Power States

```
statement latencies in milliseconds and failures:
         0.001           0 \set aid 1
         0.071           0 SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
         0.063           0 SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
         0.057           0 SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
         0.050           0 SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
         0.047           0 SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
        10.071           0 \sleep 10ms
         0.235           0 SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
         0.159           0 SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
         0.114           0 SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
         0.075           0 SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
         0.070           0 SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

Danger: Power States

- Lower IDLE States → lower power usage
- Lower IDLE States → slower “wakeup”
- Disabling IDLE states: “Less” Turbo boost is available

Pitfall: Power States

Workload:

- pgbench -S, 1 client, pinned to core
- server pinned to core

pgbench & connection cores	Core A	Core B
Core A	42552 TPS	40057 TPS

- context switch between programs on same core is faster
- different cores mean CPU is idle, enters lower power states
- larger machines or older kernels: Much bigger effects
- See also: https://vondra.me/posts/benchmarking-is-hard-sometimes/

Danger: Power States Workarounds

- Disable lower power states:
  - all cores: cpupower idle-set -D0
  - some cores: cpupower -c 11,12 idle-set -D0
  - reenable with cpupower idle-set -E
  - reduces turbo potential!
- Use batching (pipelining, DO loops)
- Use dedicated C test helpers

DO NOT DO THIS IN PRODUCTION!

Good Benchmarks

- Maximize to-be-benchmarked portion of workload
  - Context switches are expensive (use pipelining, DO $$)
- Dedicated Benchmark C functions can help a lot, see e.g. deform_bench

Maximize to-be-benchmarked

- Unusable (numeric not even visible):

  select 1::numeric/3.333
- Bad (0.3%):

  select random()::numeric/3.333
- Okay (17.88%):

  select sum(i/3.333) from generate_series(1::numeric, 1000) g(i);
- Okay (21.55%):

  select sum(i/3.333) from (SELECT generate_series(1::numeric, 1000)) g(i);
- Decent (49.5%):

  SELECT * FROM (SELECT generate_series(1::numeric, 1000)) g(i)
  WHERE I / 3.333 / i / i = 0;

	Portion of workload	Remainder	Speedup	Time
Before Speedup	1.50%	98.50%	10.00	60.00
After	0.15%	99.85%		59.19
Before Speedup	1.50%	98.50%	1.10	60.00
After	1.36%	98.64%		59.92
Before Speedup	50.00%	50.00%	10.00	60.00
After	5.00%	95.00%		33.00
Before Speedup	50.00%	50.00%	1.10	60.00
After	45.45%	54.55%		57.27

Pitfall: Huge pages for code

```
c=16
perf stat -a \
    --delay 1 \
    -e iTLB-loads:u,iTLB-load-misses:u \
    -e bp_l1_tlb_fetch_hit.if2m:u \
    -e bp_l1_tlb_fetch_hit.if4k:u \
    -e bp_l1_tlb_miss_l2_tlb_miss.coalesced_4k:u \
    \
    pgbench -n \
        -c$c -j$c \
        -Mprepared -P1 \
        -f ~/tmp/pgbench-select-only-batch-1000.sql \
        -t 1000
```

Pitfall: Huge pages for code

```
tps = 1445.073262 (without initial connection time)

 Performance counter stats for 'system wide':

     5,959,137,110      iTLB-loads
       188,963,304      iTLB-load-misses                 #    3.17% of all iTLB cache accesses
        79,081,064      bp_l1_tlb_fetch_hit.if2m:u
   106,189,019,828      bp_l1_tlb_fetch_hit.if4k:u
         4,149,038      bp_l1_tlb_miss_l2_tlb_miss.coalesced_4k:u

      11.117988359 seconds time elapsed
```

- Some userspace (it's PG) code uses some 2MB code pages
- But also lots of 4k accesses
- Coalesced entries (on newer CPU) is better than 4k, but worse than 2MB
- Hit rate not great
- No guarantee 2MB pages are available: fragmentation increases, defragmentation is deferred

Pitfall: Huge pages for code

-Wl,-zcommon-page-size=0x200000 -Wl,-zmax-page-size=0x200000

```
tps = 1539.624547 (without initial connection time)

 Performance counter stats for 'system wide':

     1,023,184,296      iTLB-loads
        15,259,274      iTLB-load-misses                 #    1.49% of all iTLB cache accesses
    15,725,998,374      bp_l1_tlb_fetch_hit.if2m:u
    95,919,605,623      bp_l1_tlb_fetch_hit.if4k:u
         7,263,772      bp_l1_tlb_miss_l2_tlb_miss.coalesced_4k:u

      10.430384664 seconds time elapsed
```

- Faster & better hit rate
- Still has 4k accesses
  - end of PG code isn't mapped 2MB
  - libc etc

Pitfall: Huge pages for code

- grep READ_ONLY_THP /boot/config-$(uname -r)
- Compile with

  -Wl,-zcommon-page-size=0x200000 -Wl,-zmax-page-size=0x200000
- Workaround:

  mount -t tmpfs -o huge=always,uid=andres none /mnt/tmpfs_huge

  cp /path/to/postgres /mnt/tmpfs_huge
- None of the workararunds are great

Other Tipps

- use constant amount of work with perf stat, to make absolute values comparable
- perf report –children and –no-children are both very useful
  - --no-children: Big bottlenecks immediately visible
  - --children: Allows to see “dispersed” bottlenecks