# Frequency Scale Invariance on x86

**Giovanni Gherdovich**
**ggherdovich@suse.cz**

OSPM 2019
21/05/2019

# performance @ SUSE, professional bias

- **server workloads**
- **performance oriented**
  - assume PM patches meet energy-saving goals
  - make sure performance trade-off is acceptable
- **support x86, ARM, PPC, s390 (a.k.a. mainframe)**
  - grid continuously running performance regression testing
  - current machine pool is ~15 Intel’s and 1 ARM
  - AMD’s and PPC on the way to the team’s pool

# outline

- concepts, definitions
- prototype patch
- test results

# frequency scale invariance

**Task running at half the freq takes twice the time ■ ▲…**

**… but `util_avg` is a function of running time…**

**… hence `util_avg` is ill-defined.**

**⇨ cannot be compared across CPUs or time**

■ assuming performance is proportional to clock frequency

▲ if the machine is not saturated. If util is maxed out, we cannot quantify the compute demand

# x86 situation

NOT F-INVARIANT

$$Q$$

F-INVARIANT

$$Q * \frac{f_{curr}}{f_{max}}$$

**PROBLEM: in x86 `fmax` is not a constant:**
**turbo states availability depends on neighboring cores**

# proposed approaches

1. **normalize against 1-active-core turbo level (max turbo)**
2. **normalize against all-cores-active turbo level**
3. **use power data (RAPL) to infer max available freq**
   - AVX workloads don’t even reach nominal freq
4. **keep average of recent past and normalize against that**

# proposed approaches

1. normalize against 1-active-core turbo level (max turbo)
2. normalize against all-cores-active turbo level
3. use power data (RAPL) to infer max available freq
   - AVX workloads don't even reach nominal freq
4. keep average of recent past and normalize against that

# V. Guittot’s new PELT invariance

## scale invariance via dilating time (at RQ level)

```
delta = time since last PELT update
delta *= freq_percent
```

- **the lower the frequency, the less PELT segments we have**
- **but ONLY if RQ is not idle. Idle time is not warped.**

```
absolute time   | 1| 2| 3| 4| 5| 6| 7| 8| 9|10|11|12|13|14|15|16
@ max capacity  ------******---------------******---------------
@ half capacity ------************---------************---------
clock pelt      | 1| 2|    3|    4| 7| 8| 9|   10|   11|14|15|16
```

# querying the hardware for OPP

DEFINITIONS

`base frequency`: max non turbo OPP.
`APERF`: counter spinning at actual core frequency. Stops at idle.
`MPERF`: counter spinning at `base frequency`. Stops at idle.
`TSC`: exactly like `MPERF` but doesn't stop at idle.

FORMULAS

```
Avg_MHz  = delta_APERF / delta_time = delta_APERF * base_freq / delta_TSC
Busy%    = delta_MPERF / delta_TSC
Busy_MHz = delta_APERF / delta_MPERF * base_freq
```

# patch description

## GLOBAL STATE

```
PER-CPU VARIABLE : freq_percent
CONSTANT         : freq_max
```

## AT PELT UPDATE

```
delta = time since last PELT update
delta *= freq_percent
...
```

util_avg → util_est → OPP selection, load balancing, Wake-up path

load_avg → OPP selection, load balancing, Wake-up path

## AT SCHED TICK

READ Δaperf

READ Δmperf

$$freq\_percent = \frac{BusyMHz}{freq\_max} = \frac{\Delta aperf}{\Delta mperf} * \frac{freq\_base}{freq\_max}$$

```
IF freq_percent > 1
        freq_percent = 1
```

WRITE freq_percent TO THIS CPU

# machine descriptions and setup

- **8x-SKYLAKE-UMA**
  - 4 cores (8 threads) Skylake (2015), single socket, 32G of memory, SSD storage
- **80x-BROADWELL-NUMA**
  - 40 cores (80 threads) Broadwell (2014), two sockets, 512G of memory, SSD storage
- **48x-HASWELL-NUMA**
  - 24 cores (48 threads) Haswell (2013), two sockets, 64G of memory, rotary disk

baseline kernel: **v5.0**

cpufreq driver/governor: **intel_pstate passive (aka intel_cpufreq) / schedutil**

filesystem: **XFS**

# neutral benchmarks

- pgbench read / write
- flexible I/O (FIO)
- NAS Parallel Benchmarks (NPB) using MPI, some computational kernels
- NPB using OpenMP, some computational kernels
- netperf on TCP (loopback)

statistically significant　meh　statistically significant

# non-neutral benchmarks

| BENCHMARK | 1 x SKYLAKE 8 CORES | | | | 2 x BROADWELL 80 CORES | | | | 2 x HASWELL 48 CORES | | | | UNIT | BETTER IF |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pgbench-ro | 0.99 | 1 | 1 | 0.99 | 1.16 | 1.01 | 1.15 | 1.02 | 1.18 | 1.07 | 1.21 | 1.04 | TRANS_PER_SECOND | higher |
| sqlite | 1.02 | 1.02 | 1.01 | 1.01 | 1.07 | 1.07 | 1.08 | 1.07 | 1.20 | 1.20 | 1.20 | 1.19 | TRANS_PER_SECOND | higher |
| dbench | 1 | 1 | 1 | 0.99 | 0.91 | 0.92 | 0.91 | 0.91 | 0.94 | 0.95 | 0.94 | 0.94 | TIME_MSECONDS | lower |
| nas-mpi-cg | 1 | 0.99 | 1 | 1 | 0.99 | 0.99 | 1 | 1 | 1 | 1 | 1.02 | 1 | TIME_SECONDS | lower |
| nas-mpi-lu | 1 | 0.99 | 1 | 1 | 1 | 1 | 1.01 | 1 | 1 | 0.99 | 1 | 1 | TIME_SECONDS | lower |
| nas-mpi-mg | 1 | 0.99 | 1 | 1 | 1 | 1 | 0.99 | 1 | 1.01 | 1 | 1.01 | 0.99 | TIME_SECONDS | lower |
| nas-mpi-sp | 1 | 0.99 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | TIME_SECONDS | lower |
| nas-omp-cg | 1.01 | 1 | 1.01 | 1 | 1 | 0.99 | 1 | 1 | 0.99 | 0.99 | 1.01 | 1 | TIME_SECONDS | lower |
| nas-omp-lu | 1.01 | 1 | 1.01 | 1 | 1 | 1.13 | 1.13 | 1.03 | 1.04 | 0.97 | 1.04 | 1.04 | TIME_SECONDS | lower |
| nas-omp-mg | 1.01 | 1 | 1.01 | 1 | 1 | 0.99 | 0.99 | 0.99 | 1 | 1 | 1.01 | 1 | TIME_SECONDS | lower |
| nas-omp-sp | 1.01 | 1 | 1.01 | 1 | 1.02 | 1 | 1.02 | 1.02 | 0.99 | 1.02 | 1.01 | 0.99 | TIME_SECONDS | lower |
| nas-omp-ua | 1.01 | 1 | 1.01 | 1 | 1.02 | 0.99 | 1 | 1 | 0.98 | 1.01 | 0.97 | 0.99 | TIME_SECONDS | lower |
| netperf-udp | 1.02 | 0.99 | 0.99 | 1.01 | 1.01 | 0.99 | 1.02 | 1.06 | 1.10 | 1.07 | 0.98 | 1.16 | MBITS_PER_SECOND | higher |
| tbench | 1.14 | 1.12 | 1.16 | 1.10 | 1.30 | 1.03 | 1.48 | 0.99 | 1.12 | 1.03 | 1.20 | 1.01 | MBYTES_PER_SECOND | higher |
| hackbench-process-pipes | 1.01 | 0.99 | 1.01 | 0.98 | 0.71 | 1 | 1 | 0.82 | 1 | 0.99 | 1 | 0.95 | TIME_SECONDS | lower |
| kernbench | 0.98 | 0.97 | 0.98 | 0.98 | 0.84 | 0.84 | 0.84 | 0.84 | 0.92 | 0.91 | 0.91 | 0.93 | TIME_SECONDS | lower |
| gitsource | 0.65 | 0.65 | 0.65 | 0.65 | 0.47 | 0.47 | 0.48 | 0.47 | 0.68 | 0.68 | 0.67 | 0.66 | TIME_SECONDS | lower |

TURBO-ALL-CORES
TURBO-1-CORE-VINCENT
TURBO-ALL-CORES-VINCENT
TURBO-1-CORE

statistically significant meh statistically significant

# non-neutral benchmarks

| BENCHMARK | 1 x SKYLAKE 8 CORES | | | | 2 x BROADWELL 80 CORES | | | | 2 x HASWELL 48 CORES | | | | UNIT | BETTER IF |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| dbench | 1 | 1 | 1 | 0.99 | 0.91 | 0.92 | 0.91 | 0.91 | 0.94 | 0.95 | 0.94 | 0.94 | TIME_MSECONDS | lower |
| tbench | 1.14 | 1.12 | 1.16 | 1.10 | 1.30 | 1.03 | 1.48 | 0.99 | 1.12 | 1.03 | 1.20 | 1.01 | MBYTES_PER_SECOND | higher |
| kernbench | 0.98 | 0.97 | 0.98 | 0.98 | 0.84 | 0.84 | 0.84 | 0.84 | 0.92 | 0.91 | 0.91 | 0.93 | TIME_SECONDS | lower |
| gitsource | 0.65 | 0.65 | 0.65 | 0.65 | 0.47 | 0.47 | 0.48 | 0.47 | 0.68 | 0.68 | 0.67 | 0.66 | TIME_SECONDS | lower |

TURBO-ALL-CORES
TURBO-1-CORE-VINCENT
TURBO-ALL-CORES-VINCENT
TURBO-1-CORE

# tbench on 48x-HASWELL-NUMA

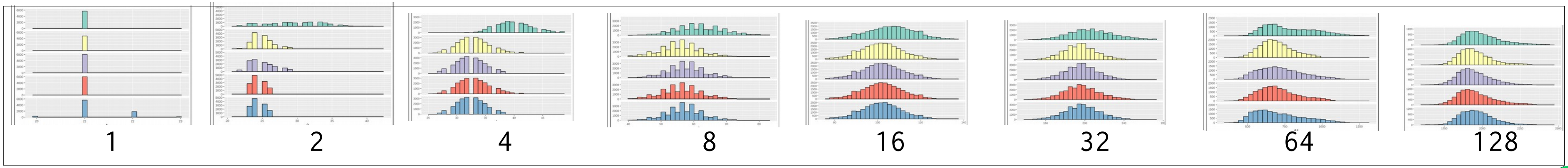
UNIT: MBYTES_PER_SECOND
HIGHER is better

| | 5.0.0 vanilla | | 5.0.0 turbo-all-cores | | | 5.0.0 turbo-1-core-vincent | | | 5.0.0 turbo-all-cores-vincent | | | 5.0.0 turbo-1-core | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 199.1 | ±0.4% | 205.0 | ±0.6% | ( 2.9%) | 199.0 | ±0.6% | ( -0.0%) | 221.8 | ±3.1% | ( 11.3%) | 200.3 | ±1.3% | ( 0.5%) |
| 2 | 389.5 | ±0.4% | 414.5 | ±0.9% | ( 6.4%) | 402.5 | ±0.5% | ( 3.3%) | 471.7 | ±2.9% | ( 21.0%) | 400.4 | ±0.3% | ( 2.7%) |
| 4 | 800.7 | ±1.7% | 916.2 | ±2.9% | ( 14.4%) | 832.0 | ±0.7% | ( 3.9%) | 1106.2 | ±1.6% | ( 38.1%) | 816.8 | ±4.0% | ( 2.0%) |
| 8 | 1595.3 | ±2.0% | 2322.0 | ±0.8% | ( 45.5%) | 1807.8 | ±1.5% | ( 13.3%) | 2656.1 | ±1.0% | ( 66.4%) | 1684.1 | ±1.0% | ( 5.5%) |
| 16 | 3246.7 | ±1.1% | 4458.2 | ±0.5% | ( 37.3%) | 3481.4 | ±0.4% | ( 7.2%) | 4937.5 | ±0.3% | ( 52.0%) | 3249.9 | ±0.4% | ( 0.1%) |
| 32 | 6315.4 | ±0.3% | 6730.0 | ±2.0% | ( 6.5%) | 6014.9 | ±0.2% | ( -4.7%) | 7257.5 | ±0.5% | ( 14.9%) | 6108.0 | ±0.3% | ( -3.2%) |
| 64 | 13226.2 | ±0.1% | 13290.6 | ±0.0% | ( 0.4%) | 13488.9 | ±0.2% | ( 1.9%) | 13183.2 | ±0.3% | ( -0.3%) | 13270.0 | ±0.4% | ( 0.3%) |
| 128 | 12063.0 | ±0.6% | 12087.0 | ±0.2% | ( 0.2%) | 11965.6 | ±0.5% | ( -0.8%) | 11836.8 | ±0.2% | ( -1.8%) | 11762.3 | ±0.0% | ( -2.4%) |
| 192 | 11639.3 | ±0.1% | 11878.3 | ±0.1% | ( 2.0%) | 11668.1 | ±0.1% | ( 0.2%) | 11710.1 | ±0.1% | ( 0.6%) | 11703.7 | ±0.1% | ( 0.5%) |

# dbench on 80x-BROADWELL-NUMA

UNIT: TIME_MSECONDS
LOWER is better

| | 5.0.0 vanilla | | 5.0.0 turbo-all-cores | | | 5.0.0 turbo-1-core-vincent | | | 5.0.0 turbo-all-cores-vincent | | | 5.0.0 turbo-1-core | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 22.1 | ±20.3% | 21.4 | ±12.3% | ( 3.1%) | 21.4 | ±11.0% | ( 3.0%) | 21.3 | ±9.3% | ( 3.3%) | 21.5 | ±9.9% | ( 2.8%) |
| 2 | 29.6 | ±16.2% | 25.1 | ±10.9% | ( 15.2%) | 25.2 | ±11.9% | ( 14.8%) | 24.7 | ±11.1% | ( 16.6%) | 24.7 | ±10.9% | ( 16.3%) |
| 4 | 40.4 | ±16.4% | 33.1 | ±18.0% | ( 18.0%) | 32.5 | ±17.5% | ( 19.5%) | 33.1 | ±17.2% | ( 17.9%) | 32.7 | ±17.8% | ( 18.9%) |
| 8 | 62.2 | ±24.9% | 56.2 | ±25.0% | ( 9.6%) | 58.0 | ±25.2% | ( 6.6%) | 56.7 | ±26.0% | ( 8.7%) | 57.4 | ±25.1% | ( 7.7%) |
| 16 | 107.1 | ±35.3% | 102.5 | ±35.5% | ( 4.2%) | 103.1 | ±35.6% | ( 3.7%) | 103.0 | ±35.5% | ( 3.8%) | 102.8 | ±35.6% | ( 3.9%) |
| 32 | 212.8 | ±48.5% | 199.3 | ±49.7% | ( 6.3%) | 200.4 | ±49.2% | ( 5.8%) | 202.0 | ±49.3% | ( 5.0%) | 201.9 | ±49.6% | ( 5.1%) |
| 64 | 809.2 | ±48.5% | 720.7 | ±53.5% | ( 10.9%) | 747.6 | ±50.4% | ( 7.6%) | 734.9 | ±50.1% | ( 9.1%) | 730.7 | ±52.8% | ( 9.6%) |
| 128 | 2128.7 | ±18.5% | 2071.8 | ±17.0% | ( 2.6%) | 2058.9 | ±16.6% | ( 3.2%) | 2078.9 | ±23.0% | ( 2.3%) | 2080.3 | ±16.6% | ( 2.2%) |



# kernbench on 48x-HASWELL-NUMA

```
UNIT: TIME_SECONDS
LOWER is better
```

| | 5.0.0 vanilla | | 5.0.0 turbo-all-cores | | | 5.0.0 turbo-1-core-vincent | | | 5.0.0 turbo-all-cores-vincent | | | 5.0.0 turbo-1-core | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| elps-2 | 393.5 | ±0.2% | 333.6 | ±0.1% | (15.2%) | 331.2 | ±0.2% | (15.8%) | 328.2 | ±0.1% | (16.5%) | 350.9 | ±0.2% | (10.8%) |
| elps-4 | 200.5 | ±0.2% | 172.9 | ±0.1% | (13.7%) | 172.6 | ±0.1% | (13.9%) | 171.7 | ±0.2% | (14.3%) | 179.4 | ±0.3% | (10.5%) |
| elps-8 | 103.5 | ±0.5% | 91.6 | ±0.9% | (11.4%) | 92.1 | ±1.5% | (11.0%) | 91.1 | ±0.3% | (11.9%) | 93.4 | ±0.4% | (9.7%) |
| elps-16 | 57.5 | ±0.7% | 52.9 | ±0.8% | (7.9%) | 52.2 | ±1.7% | (9.1%) | 52.5 | ±1.5% | (8.7%) | 53.5 | ±1.4% | (6.8%) |
| elps-32 | 38.8 | ±1.5% | 37.1 | ±1.1% | (4.3%) | 36.8 | ±0.8% | (5.2%) | 36.7 | ±2.2% | (5.5%) | 36.5 | ±2.0% | (6.0%) |
| elps-64 | 33.9 | ±2.5% | 32.0 | ±1.7% | (5.6%) | 32.4 | ±1.0% | (4.5%) | 31.7 | ±1.3% | (6.4%) | 32.8 | ±1.0% | (3.1%) |
| elps-96 | 34.6 | ±0.4% | 32.5 | ±1.5% | (6.0%) | 32.5 | ±0.5% | (6.2%) | 32.2 | ±0.6% | (6.8%) | 32.3 | ±1.5% | (6.6%) |

# gitsource on 8x-SKYLAKE-UMA

```
UNIT: TIME_SECONDS
LOWER is better
```

| | 5.0.0 vanilla | 5.0.0 turbo-all-cores | 5.0.0 turbo-1-core-vincent | 5.0.0 turbo-all-cores-vincent | 5.0.0 turbo-1-core |
|---|---|---|---|---|---|
| Elapsed | 976.9 ±7.2% | 635.3 ±0.1% ( 34.9%) | 636.5 ±0.2% ( 34.8%) | 638.1 ±0.3% ( 34.6%) | 639.3 ±0.3% ( 34.5%) |

# possible explanation

- normalizing against a freq in turbo range makes core look artificially under-utilized
- scheduler gives it more work
- self-fulfilling prophecy: core goes boost

- NOTE: for the largest gains, the workload needs to be just the right size (see tables)
- but smaller or larger workloads do not regress

# runtime search for `f-max`

P. Zijlstra's patch is defensive against machines not returning `MSR_TURBO_RATIO_LIMIT`

Q1: do we really have to expect that?

Q2: if yes, what runtime search for all-cores-active turbo level? Some fraction of observed `f-max`?

# all in all

- freq invariance is attained scaling things by `f_curr / f_max`
- x86: what is `f_max`?
- we pretend `f_max` is mild turbo (all-cores-active)
- large boosting opportunities are enabled for workloads of the right size
- plays well with Vincent Guittot’s new PELT invariance

# Still time?

# frequency scale invariance approximation

- the `f_next` formula today in (non-invariant) schedutil approximates scale invariance
- maybe that approximation is enough?
- some handwaving follows (no actual computations of the error)

# frequency scale invariance approximation

> schedutil formula

  > utilization is frequency invariant (ARM):

$$\mathtt{freq_{next} = 1.25 * freq_{max} * util}$$

  > utilization is not frequency invariant (x86):

$$\mathtt{freq_{next} = 1.25 * freq_{curr} * util}$$

# frequency scale invariance approximation

> schedutil formula

  > utilization is frequency invariant (ARM):

$$freq_{next} = 1.25 * freq_{max} * util$$

# frequency scale invariance approximation

> schedutil formula
>
> > utilization is frequency invariant (ARM):

$$\texttt{freq}_{\texttt{next}} = 1.25 * \texttt{freq}_{\texttt{max}} * \texttt{util}$$

> > > rationale: make $\texttt{freq}_{\texttt{next}}$ proportional to util
> > >
> > > since `1.25 * 0.8` is `1`, when util is `0.8` sets freq to max
> > >
> > > we consider 80% a high utilization, so better speed up
> > >
> > > note: after switching freq, utilization remains the same

# frequency scale invariance approximation

> schedutil formula

  > utilization is not frequency invariant (x86):

$$\texttt{freq}_{\texttt{next}} = 1.25 * \texttt{freq}_{\texttt{curr}} * \texttt{util}$$

# frequency scale invariance approximation

- > schedutil formula
  - > utilization is not frequency invariant (x86):

$$freq_{next} = 1.25 * freq_{curr} * util$$

- > derived from the invariant case, replace

$$util_{inv} = util_{raw} * freq_{curr} / freq_{max}$$

- > approximation: $util_{raw}$ is a PELT sum, each term needs to be scaled (with $freq_{curr}$ at that time)
- > $util_{raw}$ == 0.8 is the tipping point: less than 0.8 and freq goes down, more than 0.8 and freq goes up

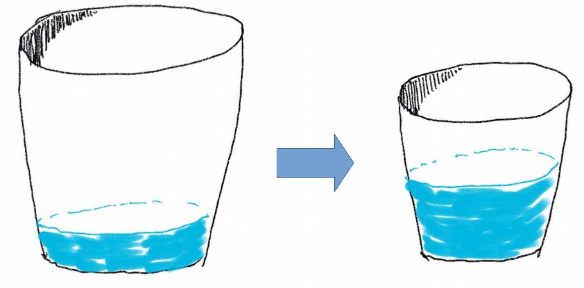
# frequency scale invariance approximation

> analogy for the non invariant case: bucket of water

You're given a bucket F with some water W. Let's call U the ratio of water volume by the total:

$$U = W / F$$

Find the volume of a new bucket F' to pour the water into so that the new utilization U' = W / F' is 0.8.



# frequency scale invariance approximation

> analogy for the non invariant case: bucket of water

You're given a bucket F with some water W. Let's call U the ratio of water volume by the total:

$$U = W / F$$

Find the volume of a new bucket F' to pour the water into so that the new utilization U' = W / F' is 0.8.

$$0.8 = W / F'$$

$$\Rightarrow F' = 1.25 * W$$

$$\Rightarrow F' = 1.25 * F * U$$

# frequency scale invariance approximation

> analogy for the non invariant case: bucket of water

> > water bucket: F is total volume, W is water volume
> > freq switching: F is current frequency, W is instructions per second (“useful work”).

> > if F is cycles per second, U = W / F would give instruction per cycle (IPC). Maybe?