yogini

Stretching the Linux Scheduler...
...to its Limits

Len Brown
Sr. Principal Engineer
Intel Open Source Technology Center

Linux Plumbers Conference   Dublin, Ireland Sept 2022

Opportunity

A tool integrating...

1. workload generation
2. hardware and software observation
3. report generation

Useful for scheduler+power+performance...

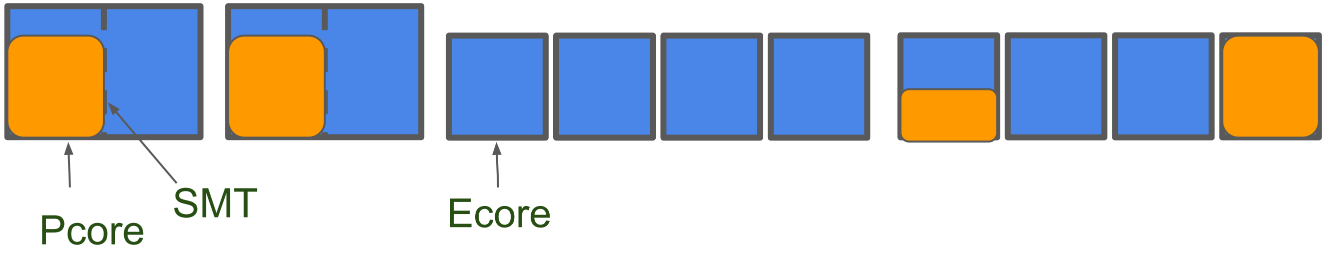
1. design
2. debug
3. tuning
4. regression testing

Agenda

1. Example
2. How yogini works
3. Another Example

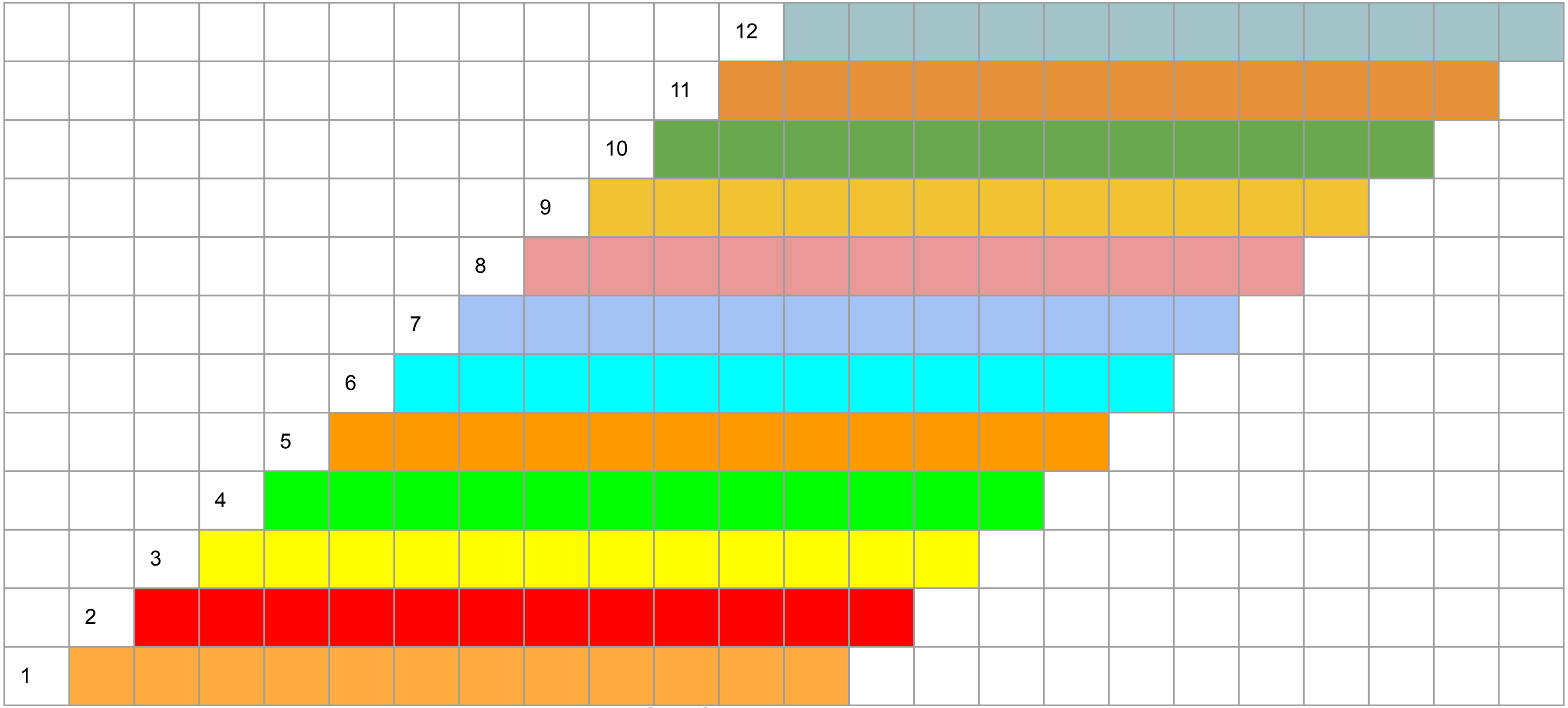
Linux v5.16 ITMT on Intel 2xPcore + 8xEcore

Task Placement:
1. Pcore
2. Ecore
3. Pcore HT sibling

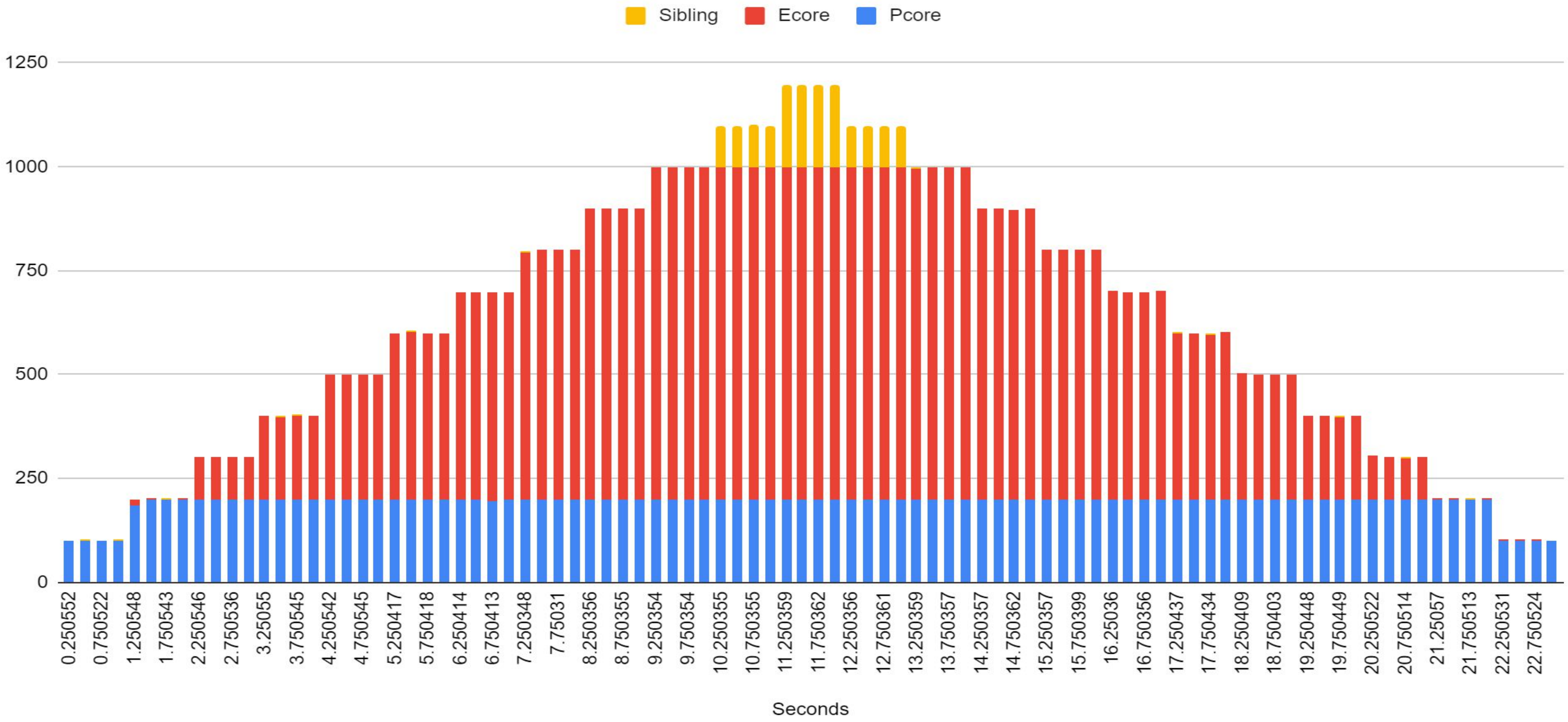


Scheduler spreads to Ecore before HT sibling.

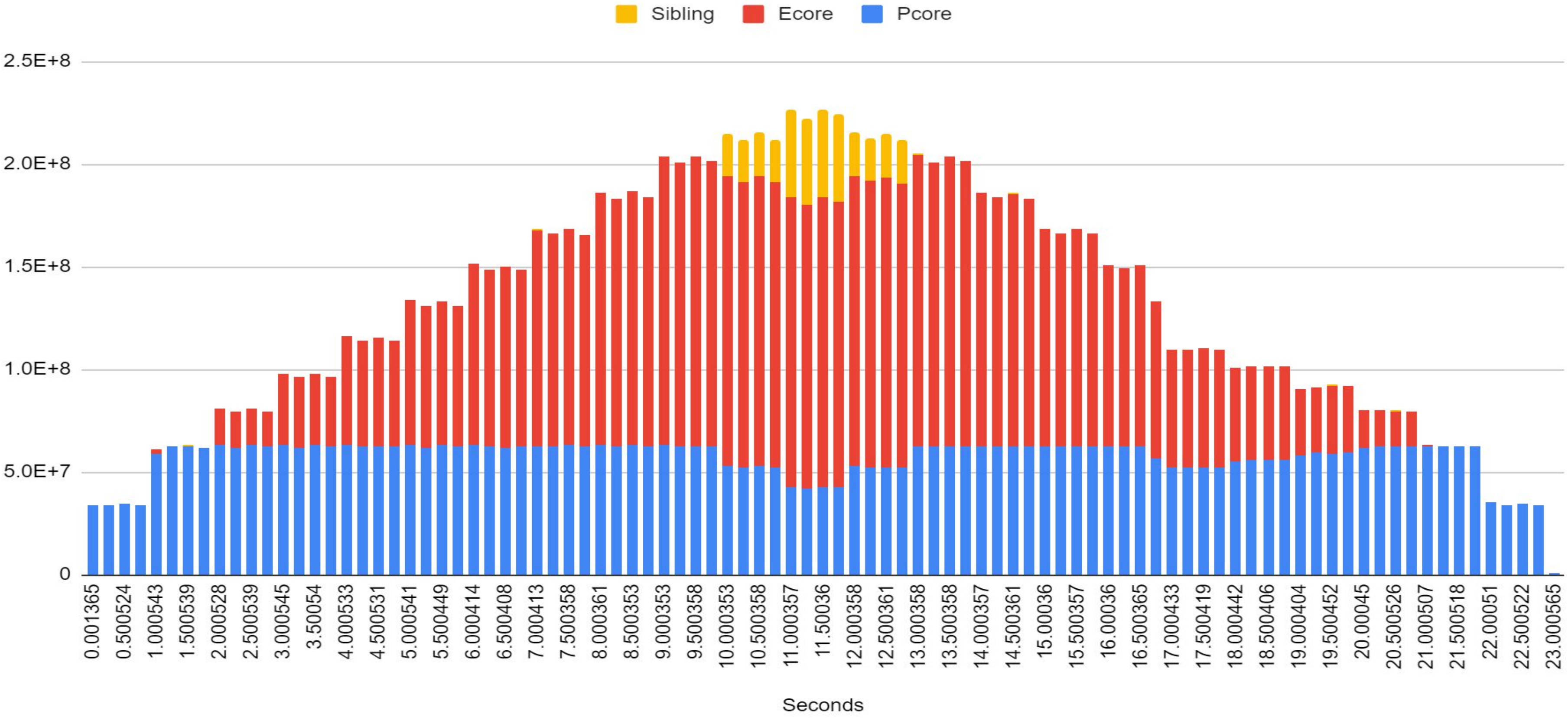
12-thread FIFO (100%) Stimulus



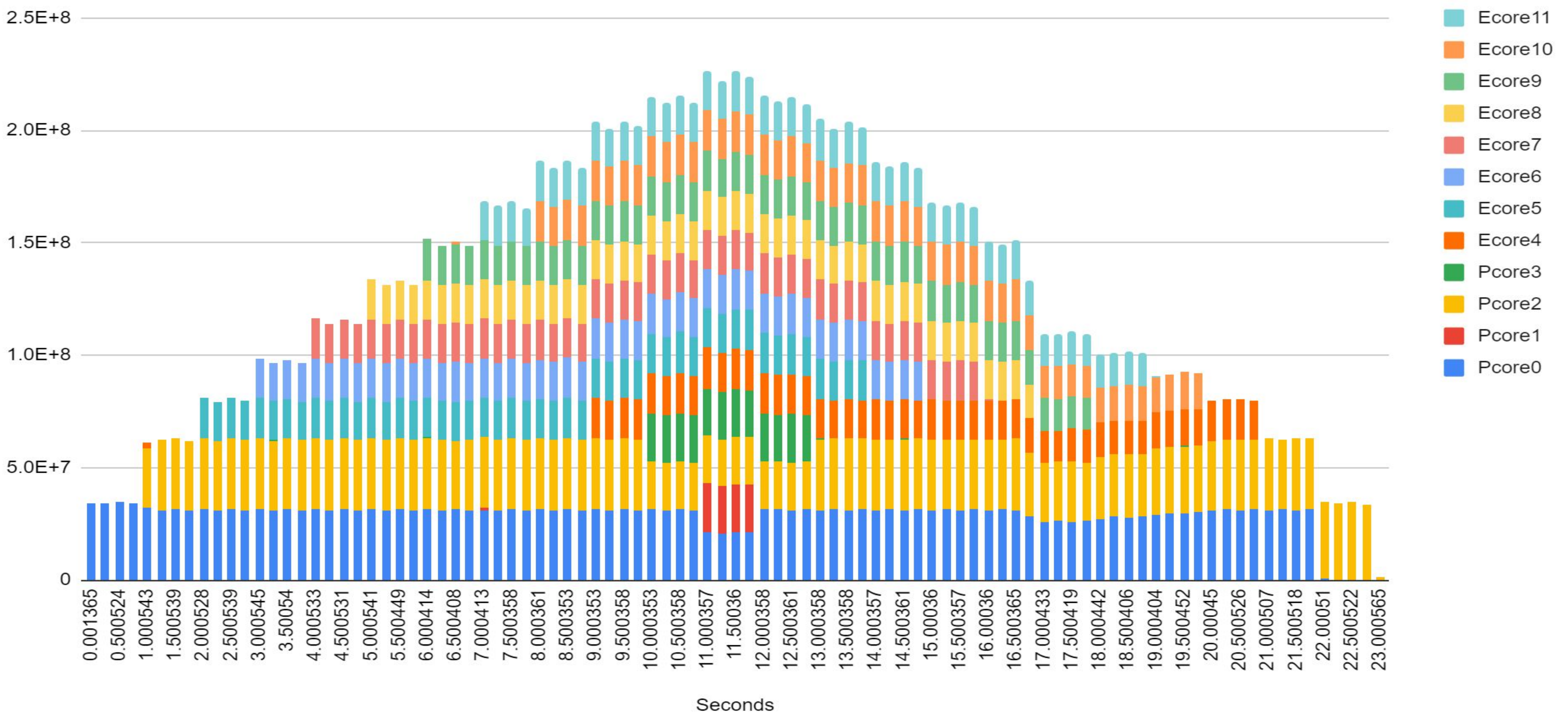
Yogini Pyramid100 Busy % by CPU type



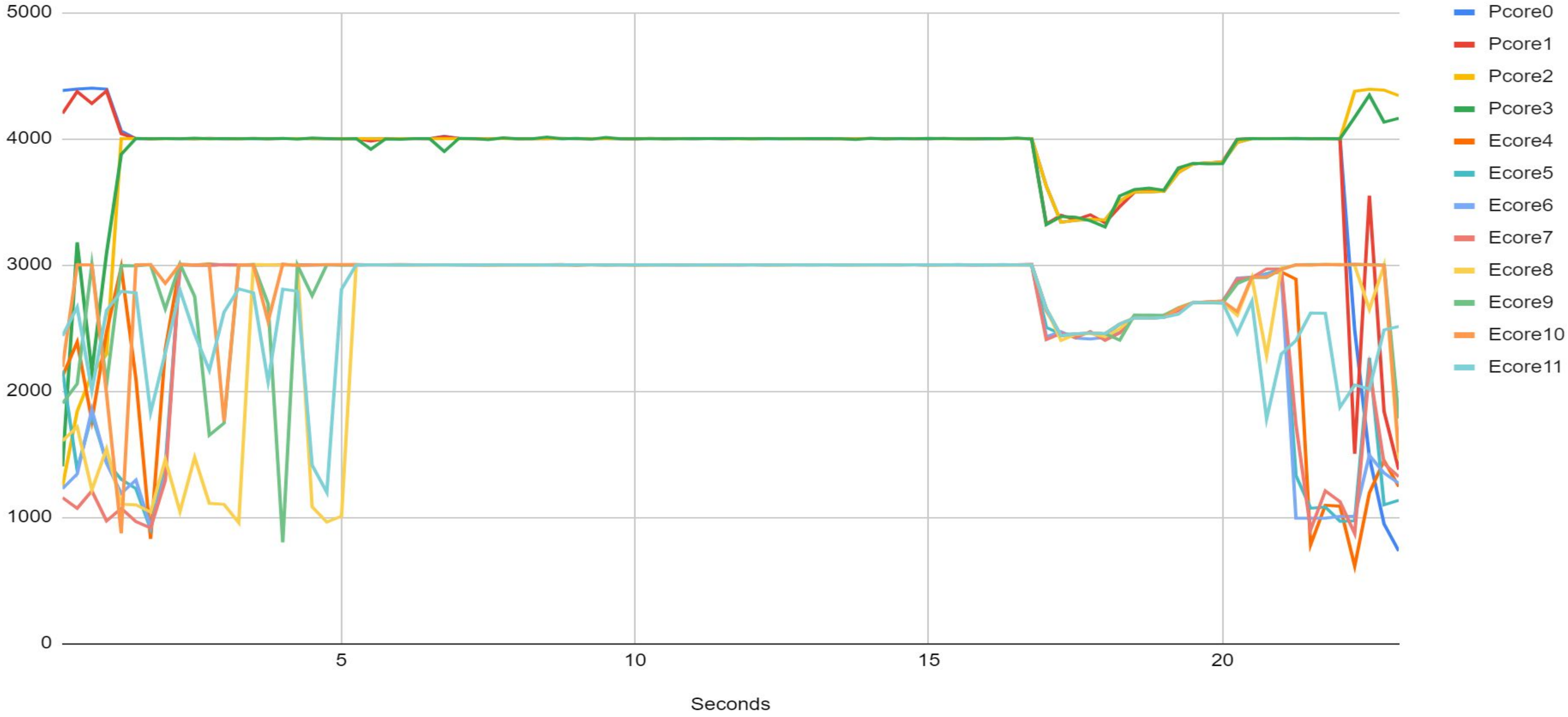
Yogini pyramid100 work done by CPU type



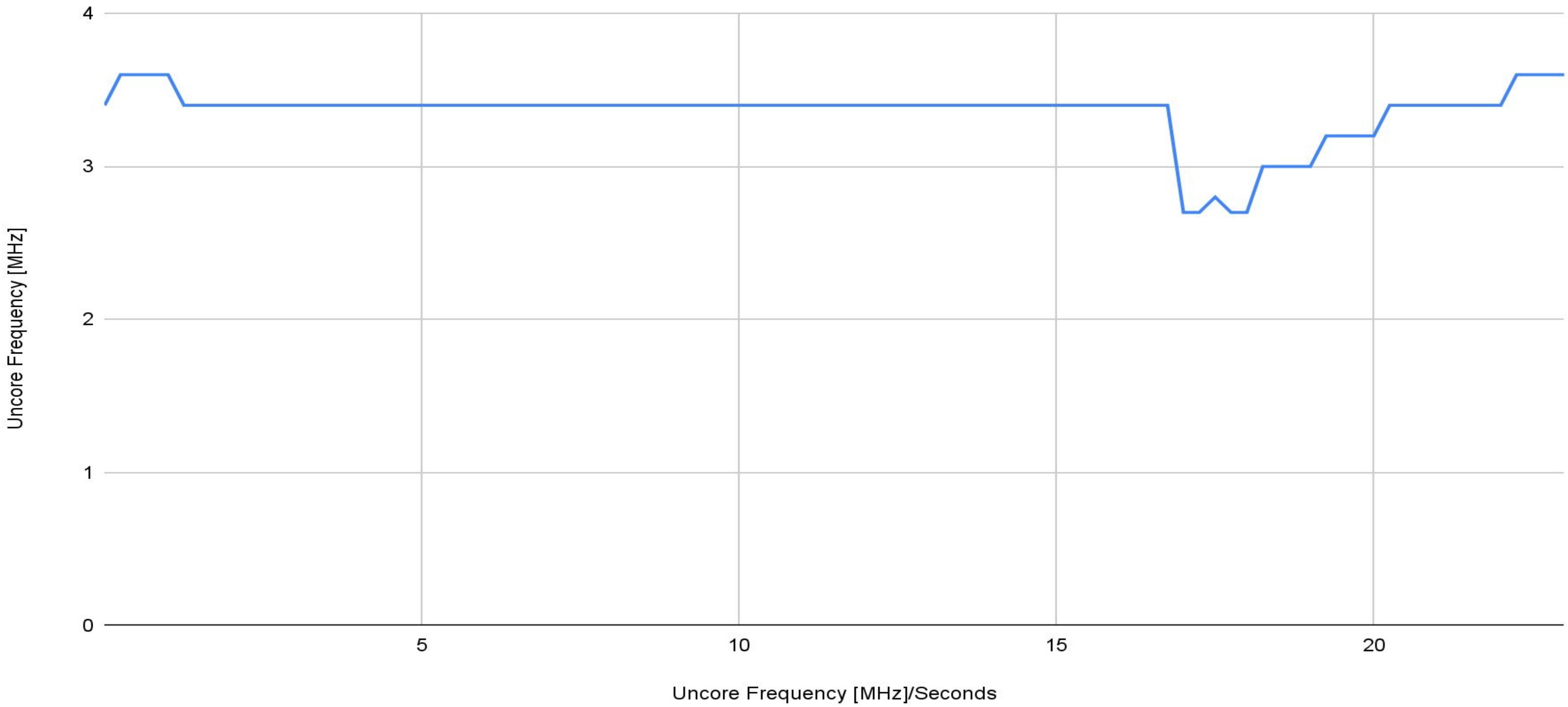
Yogini pyramid100 work done per CPU over time



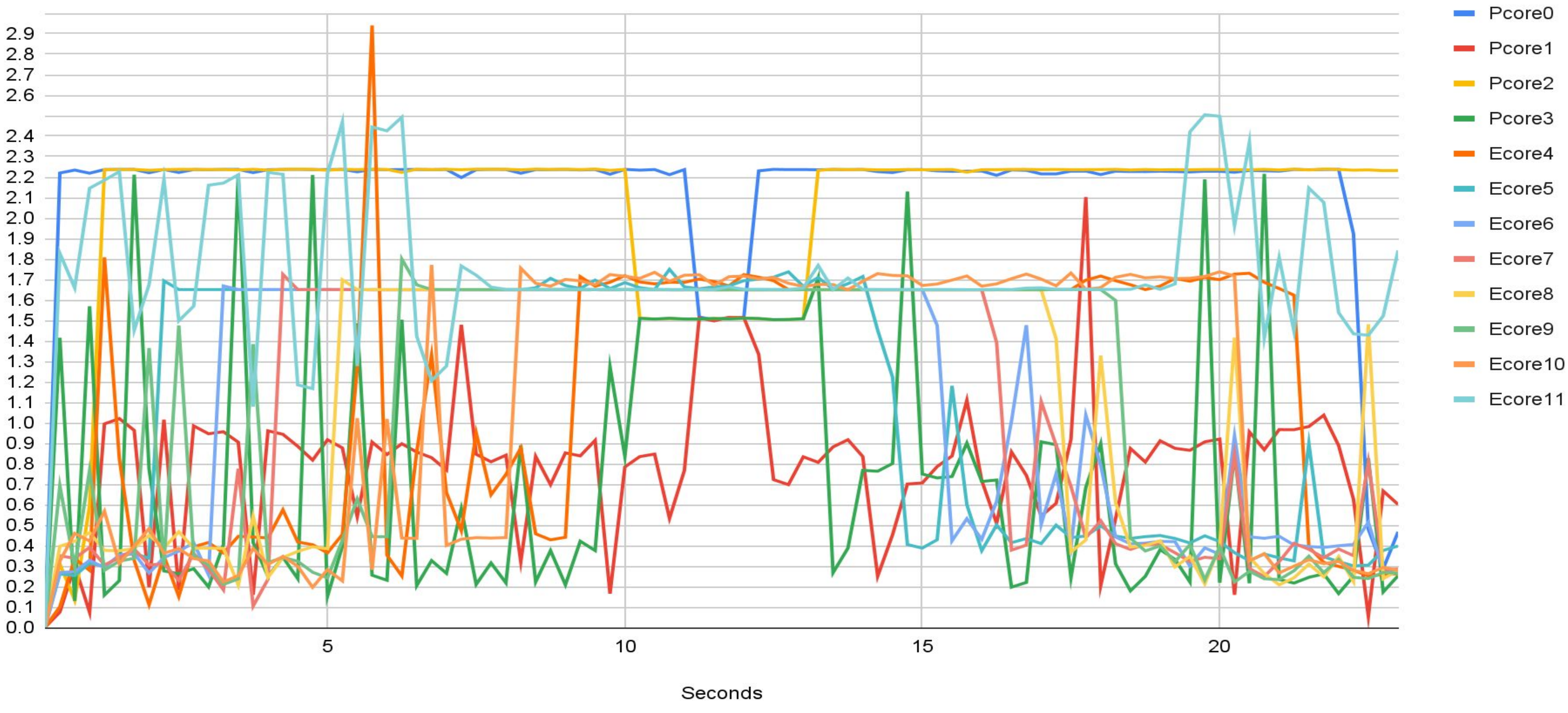
Yogini pyramid100 Frequency vs CPU over time



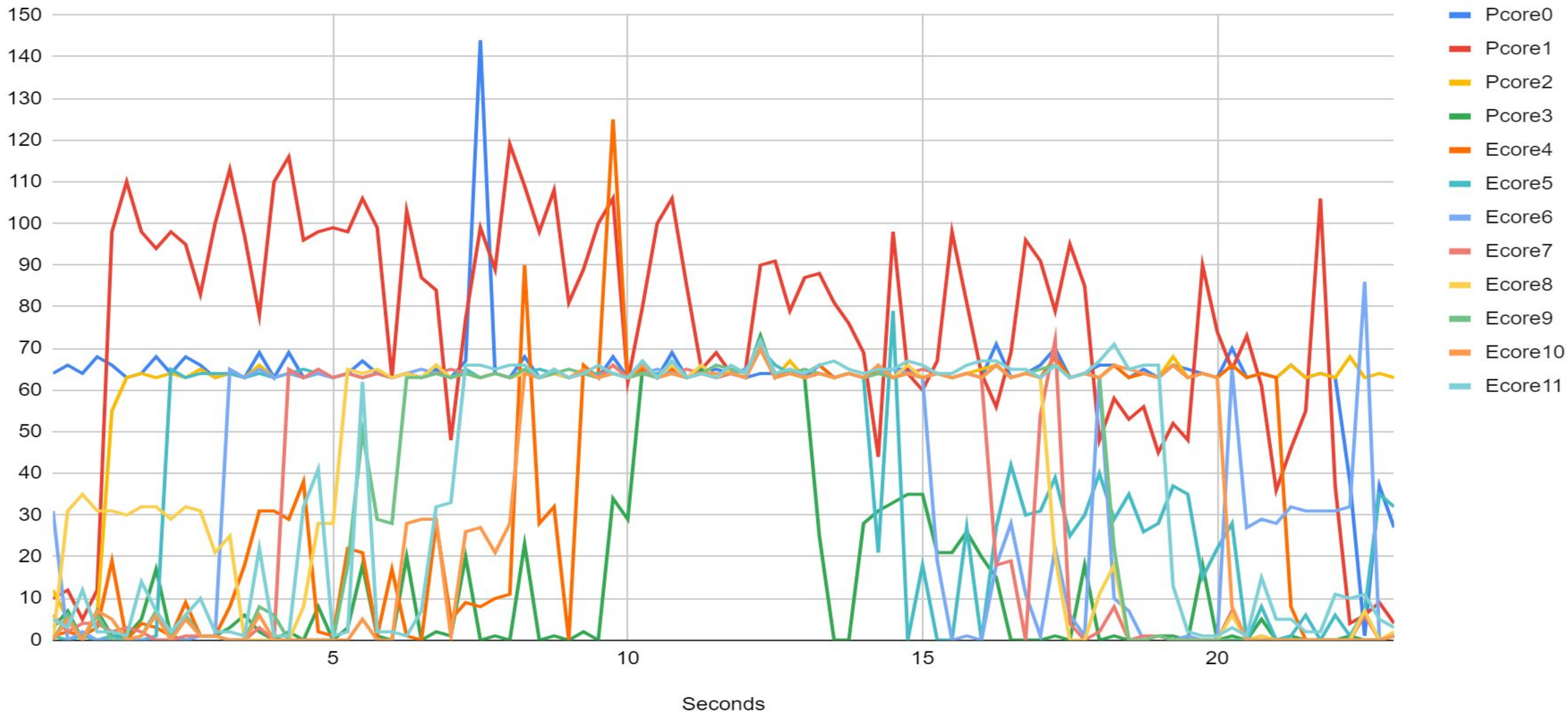
Yogini pyramid100 Uncore MHz vs. time



Yogini pyramid100 - IPC/CPU vs time

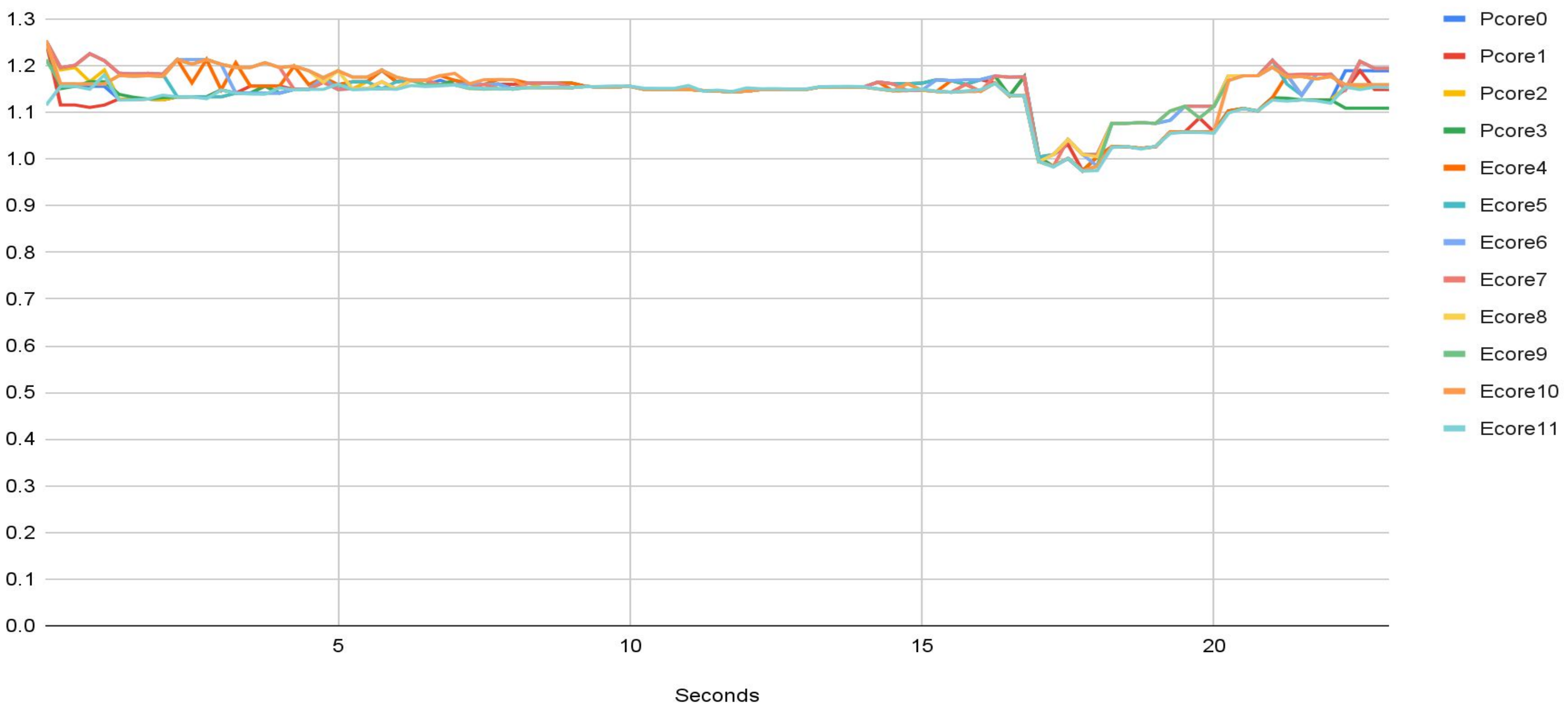


Yogini pyramid100 - IRQ/250ms per CPU

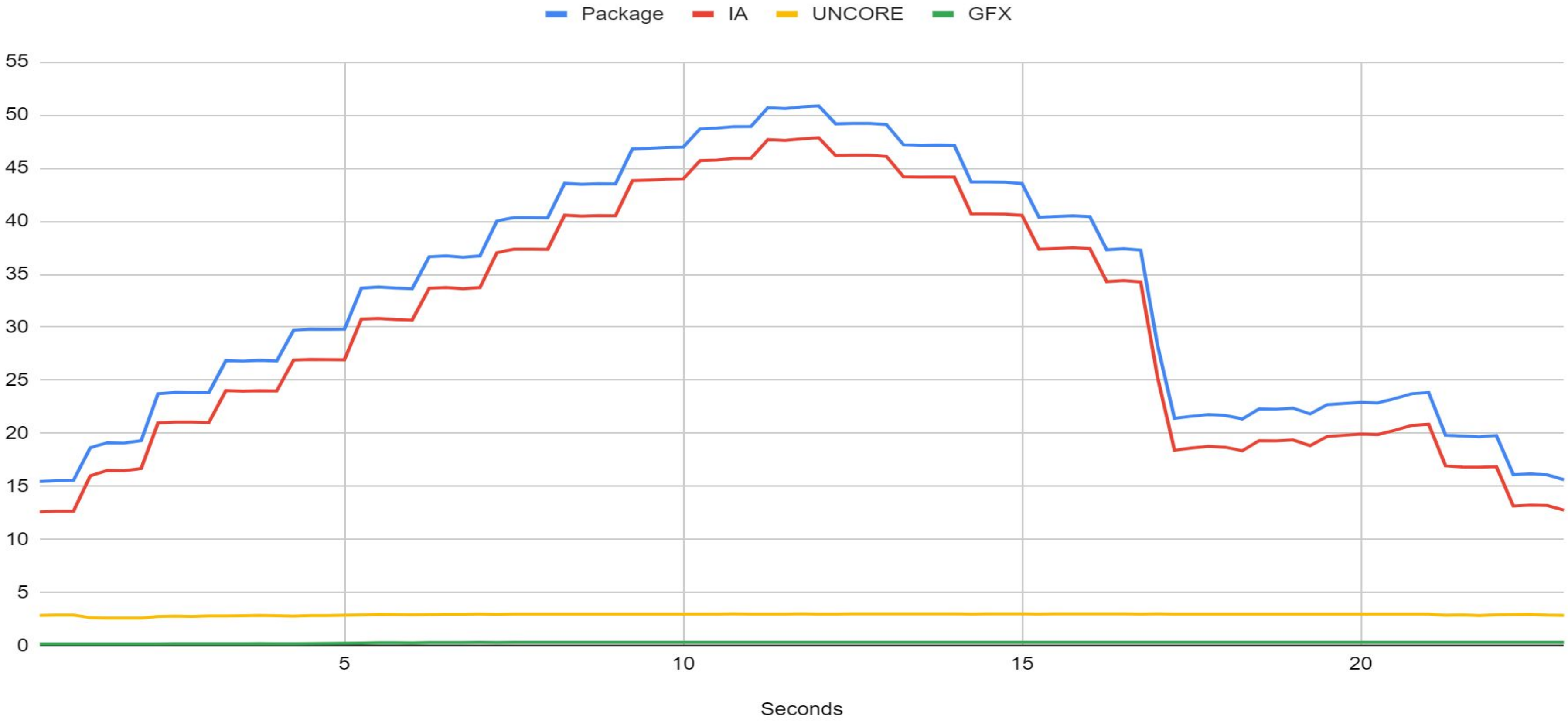


Yogini pyramid100 Temperature/CPU vs time

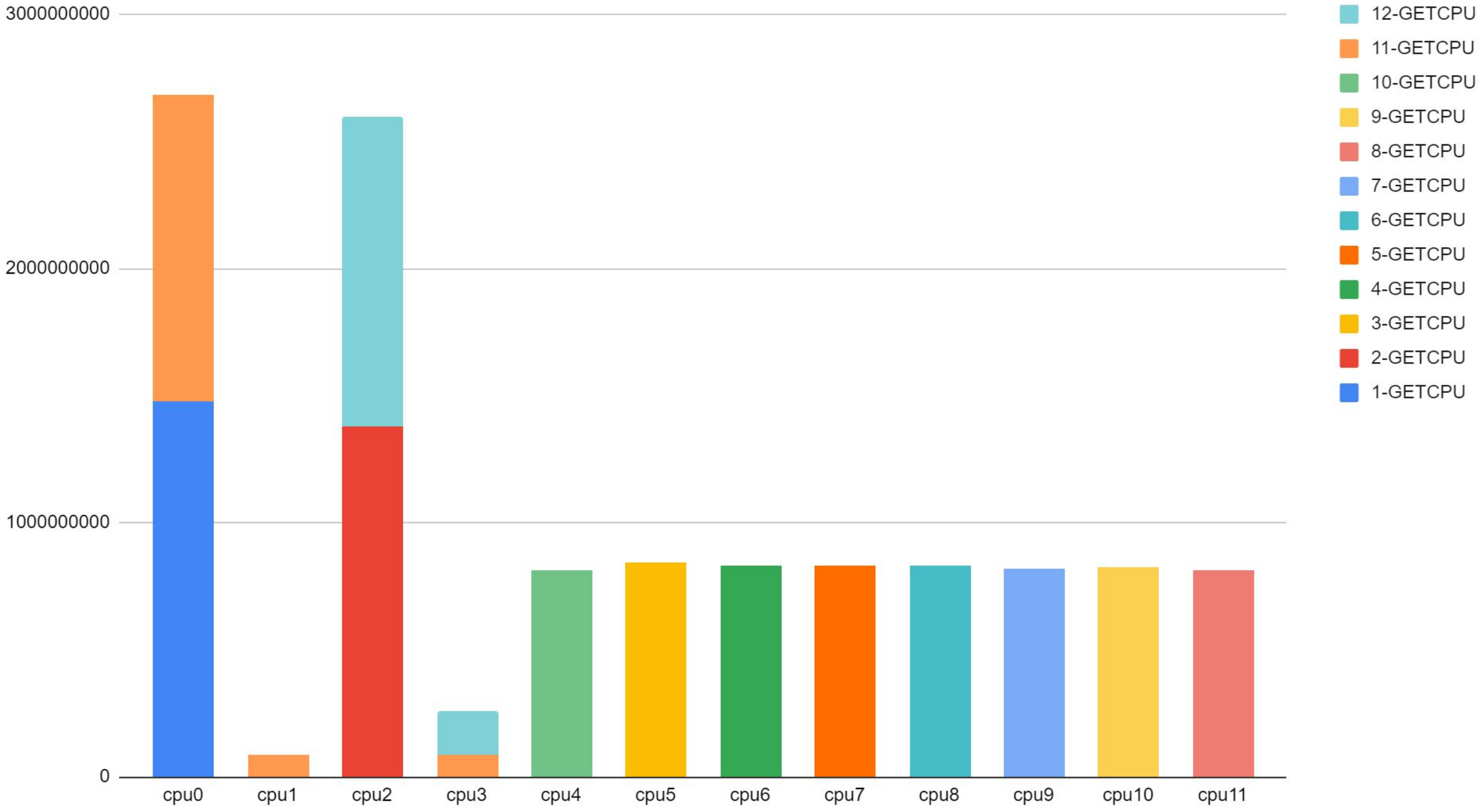
Yogini pyramid100 Volts/CPU vs time



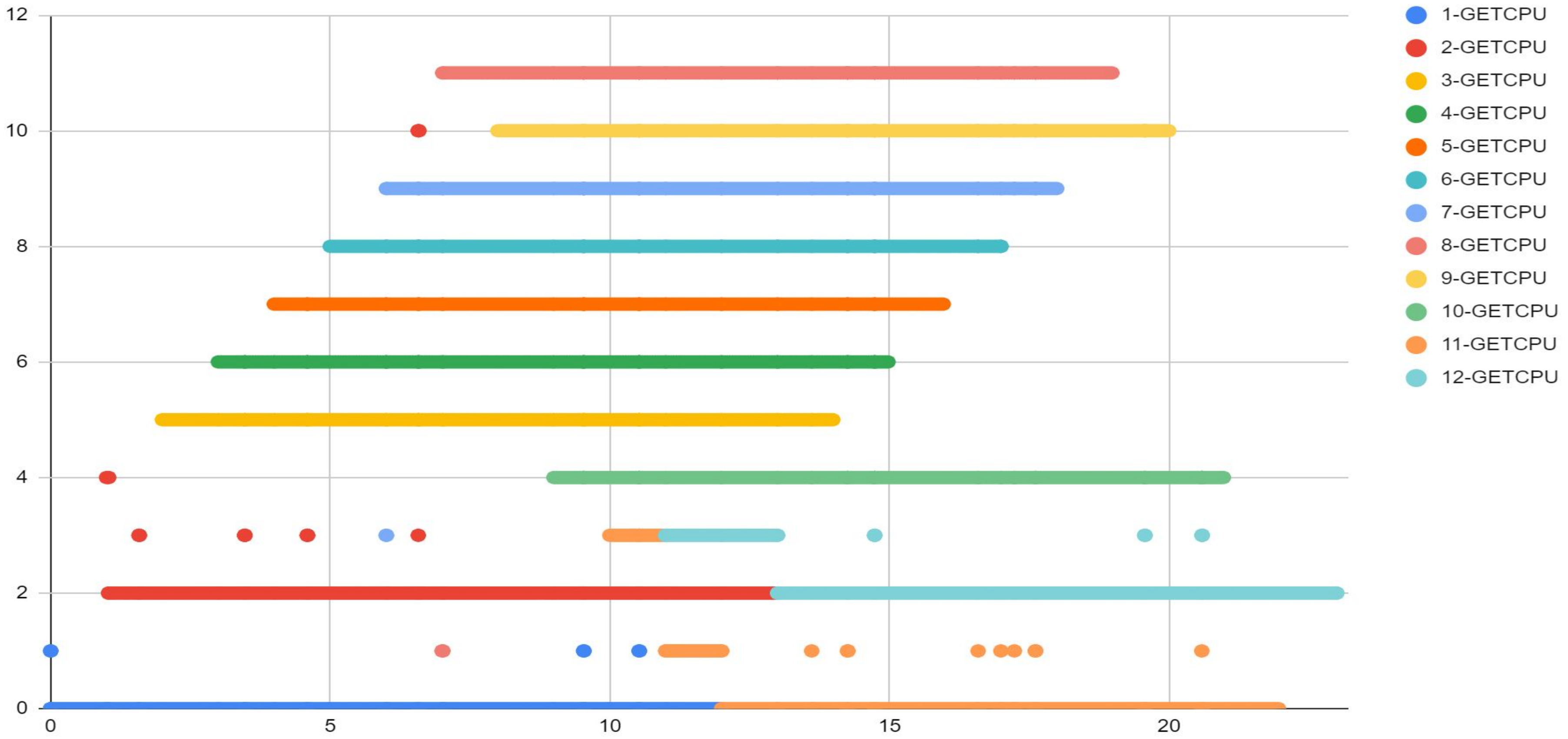
Package, IA, UNCORE and GFX Power vs time



Yogini Pyramid100 Thread Work done by CPU



Yogini Pyramid100 Thread CPU Residency Trace



Yogini Purpose: The ability to easily...

1. Generate well-understood workloads, to challenge Linux PM & scheduler
2. Observe scheduler's success/failure against those challenges
3. Foundation for regression test suite, to assure continuous improvement

Yogini Goals: It needs to be easy to...

1. Install
2. Run on any topology
3. Run on any version of Linux
4. Share results
5. Understand results
6. Reproduce results
7. Compare before/after
8. Extend with additional workloads

Quick Start: Install, Run, Observe, Share

```
# tar zxf yogini-VERSION.tar.gz

# cd yogini-VERSION

# ./yogini > output.tsv
```

google sheets: Import output.tsv

select data region, click "Insert Chart"

click SHARE

Optional Worker Parameters

Worker type

Number of copies (threads)

Waveform: eg. Rate of work, @ begin, @end

Start time, end-time, duty-cycle

affinity: start, stop, permanent

```
man ./yogini.8
```

How yogini works

1. Calibrate Hardware

2. Start System Monitor

3. Run work

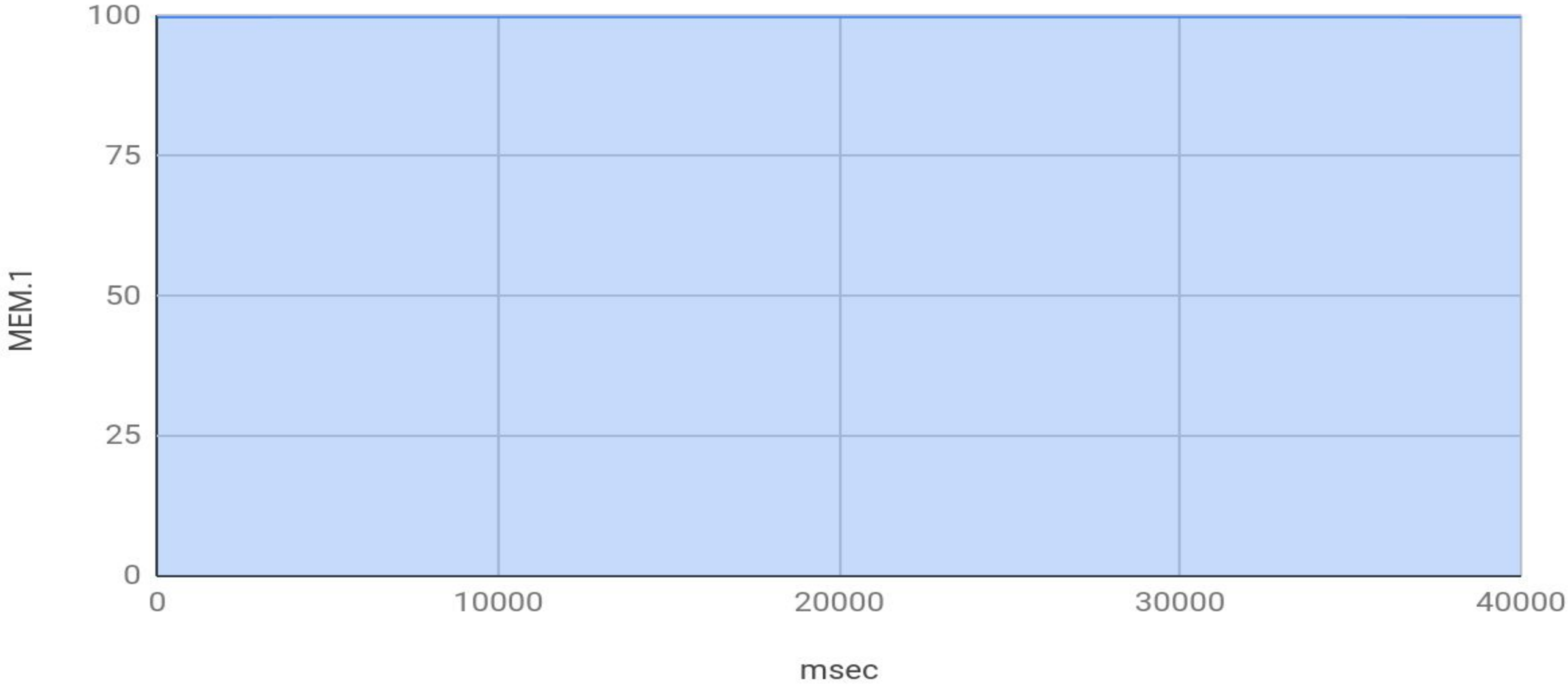
4. Output Results

Calibration sets "100%: Performance

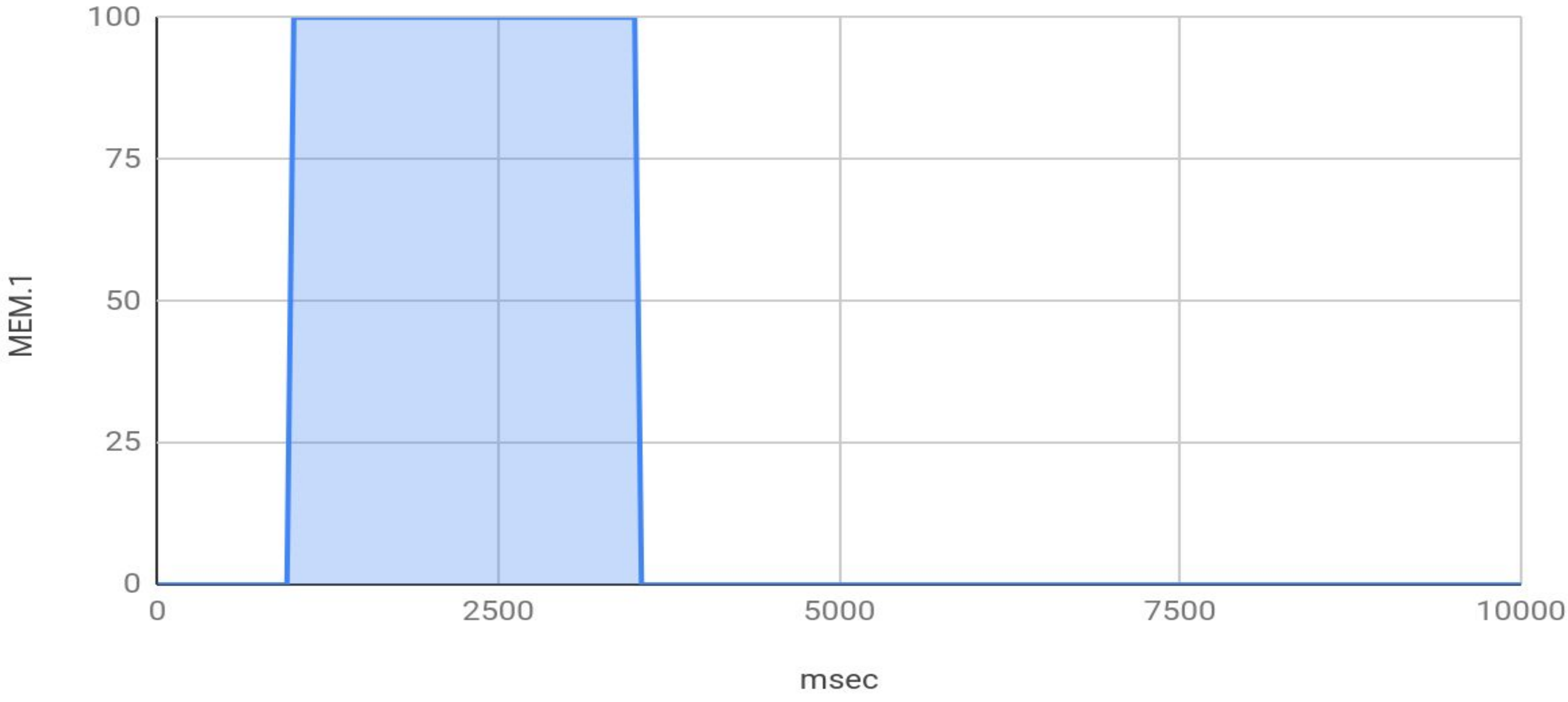
For every workload type, in a test, 100% performance must be known

1. use pre-calibrate: **`–calibrate AVX,12345678`**
2. measure on cpu0, or fastest of N CPUs: **`–calibrate N`**

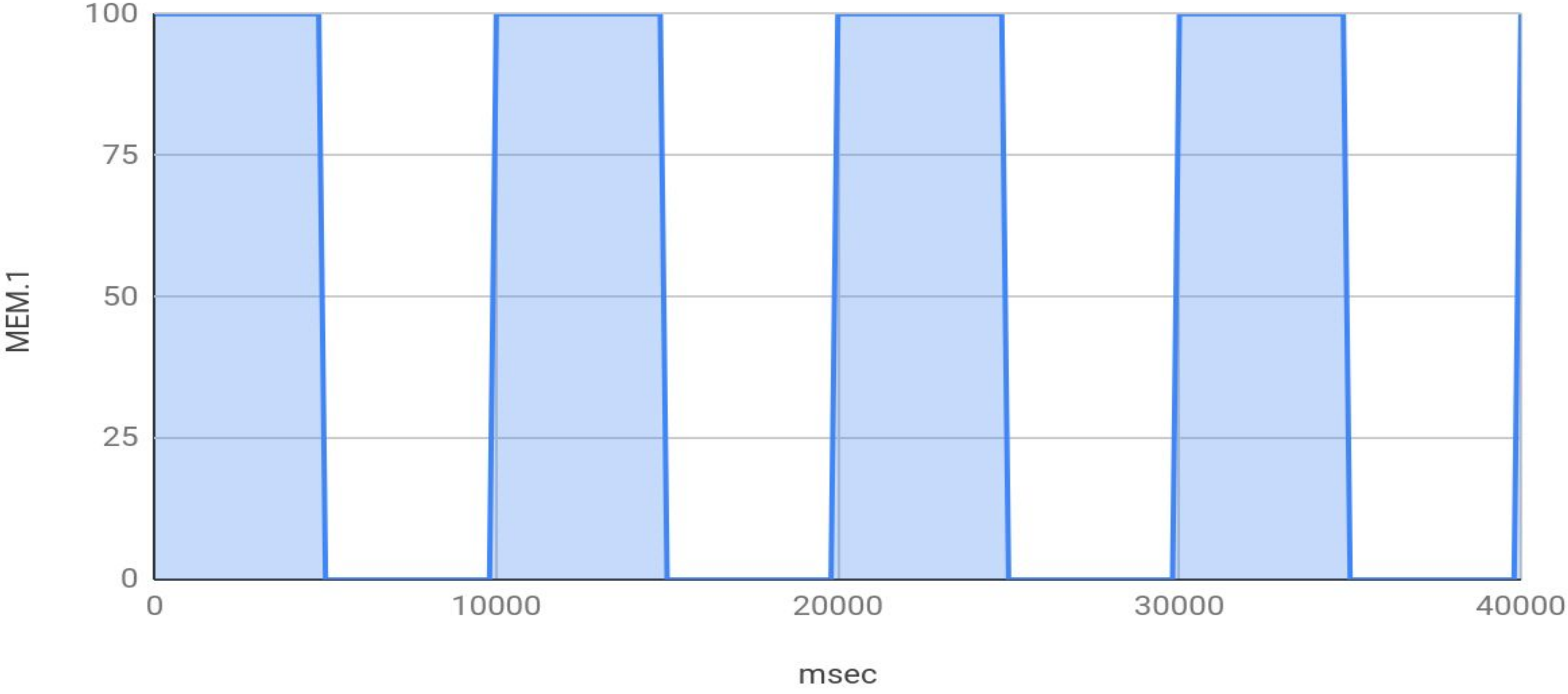
yogini -w rate100



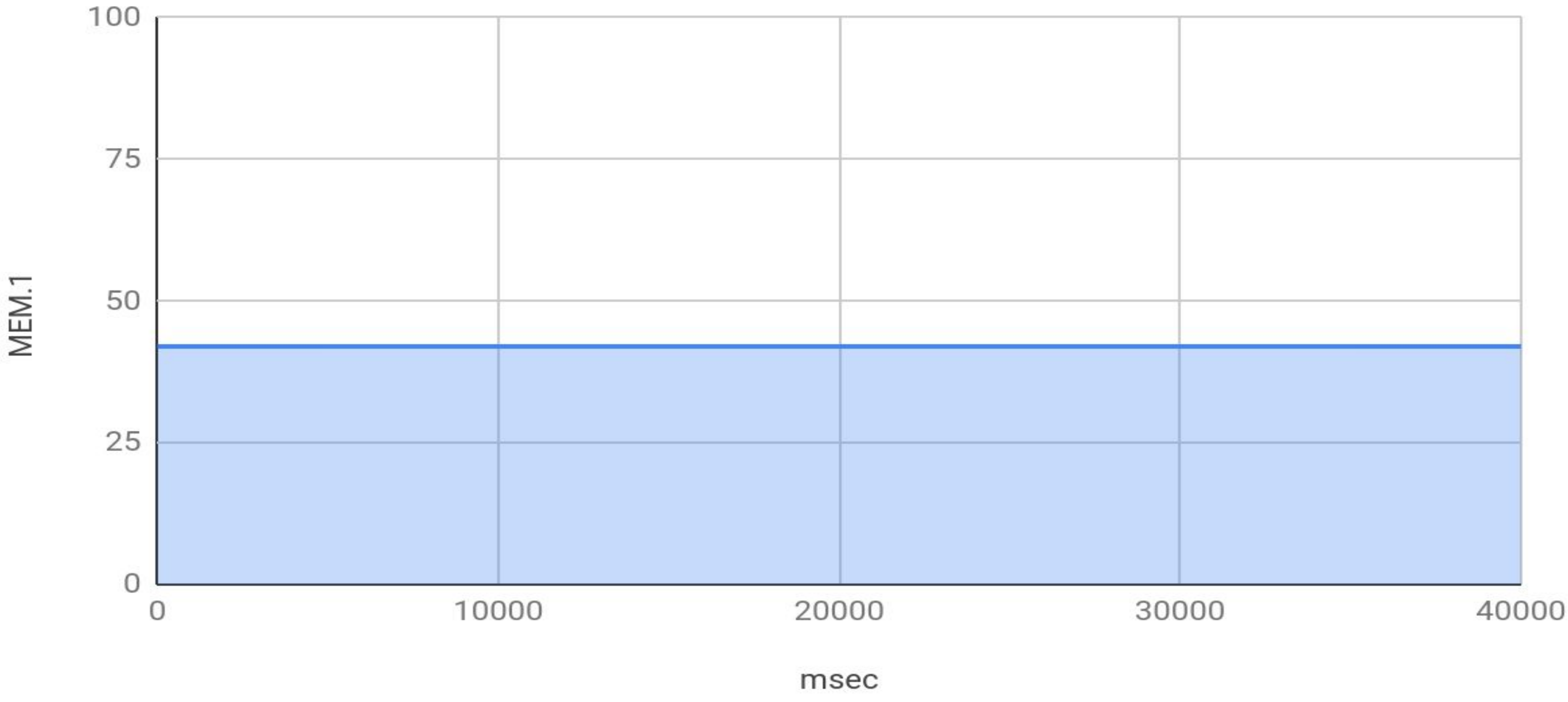
yogini -w start-msec1000,stop-msec3500



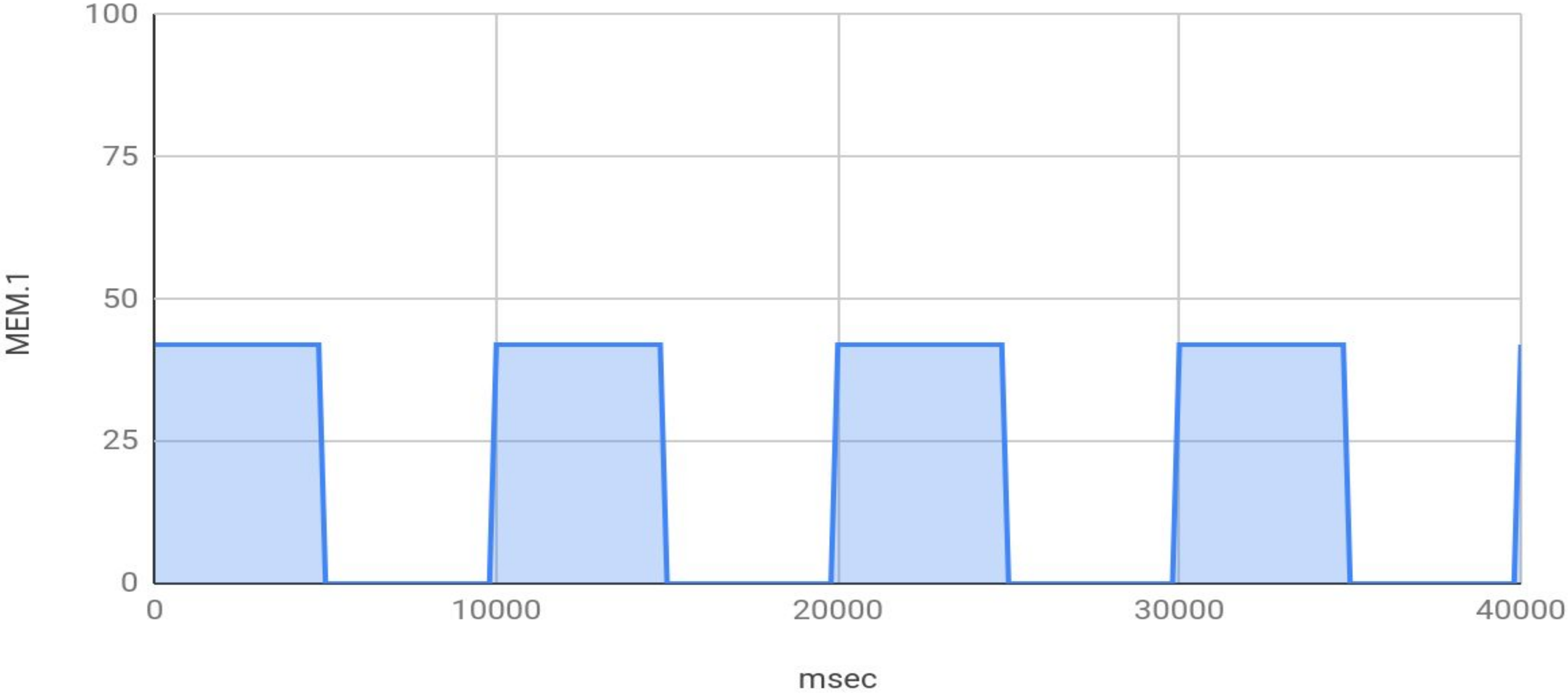
yogini -w duty-cycle50



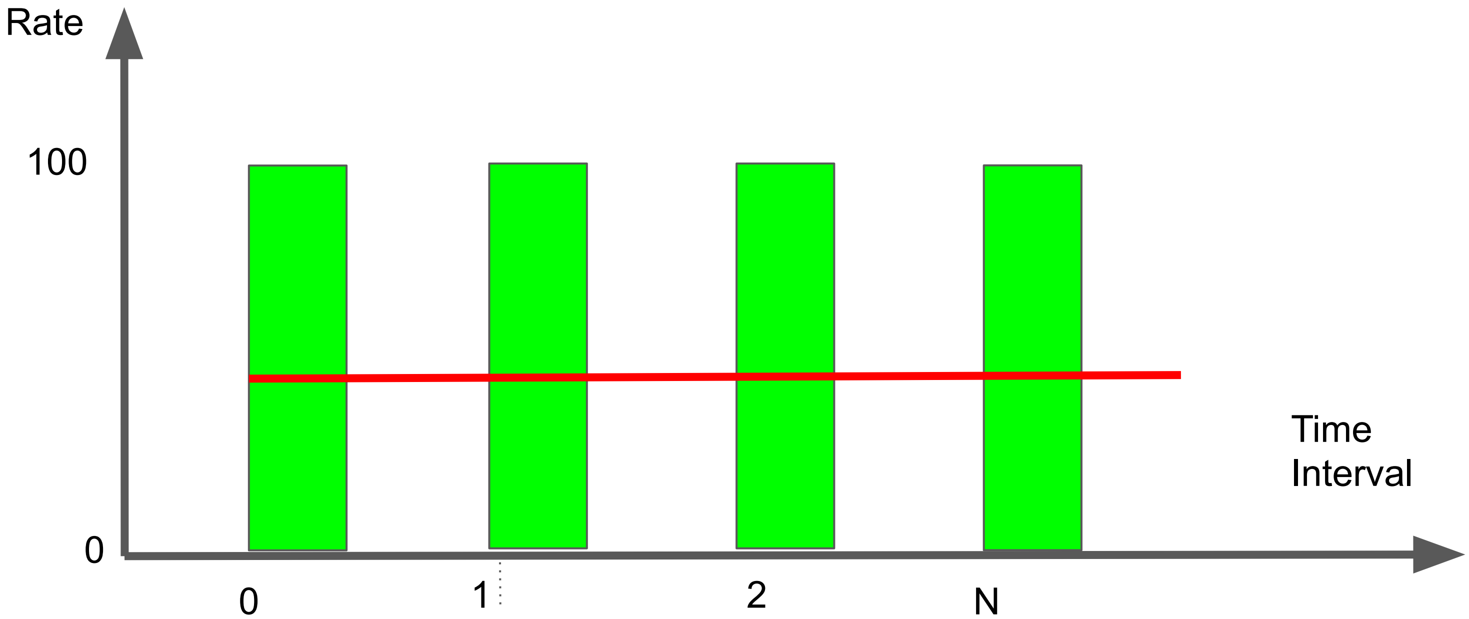
yogini -w rate42



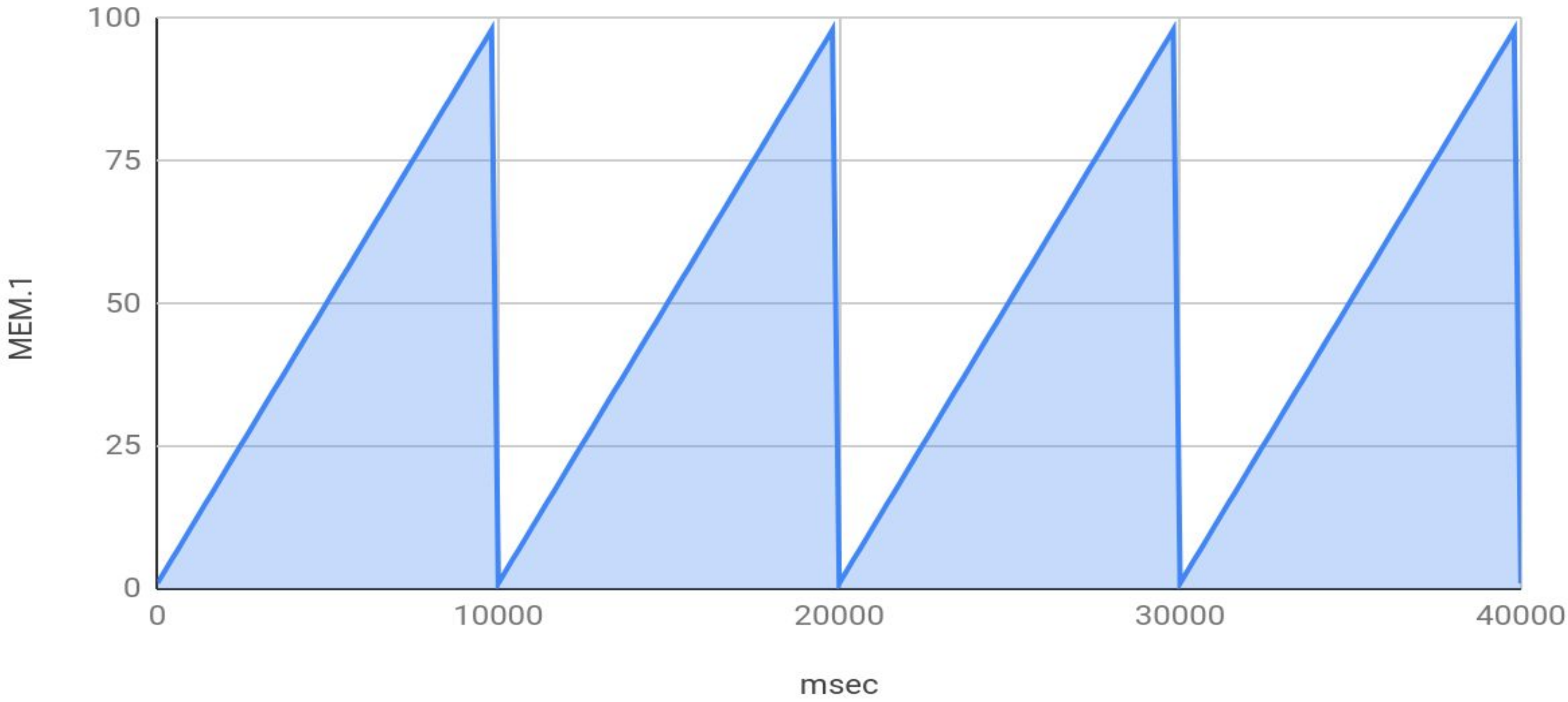
yogini -w rate42,duty-cycle50



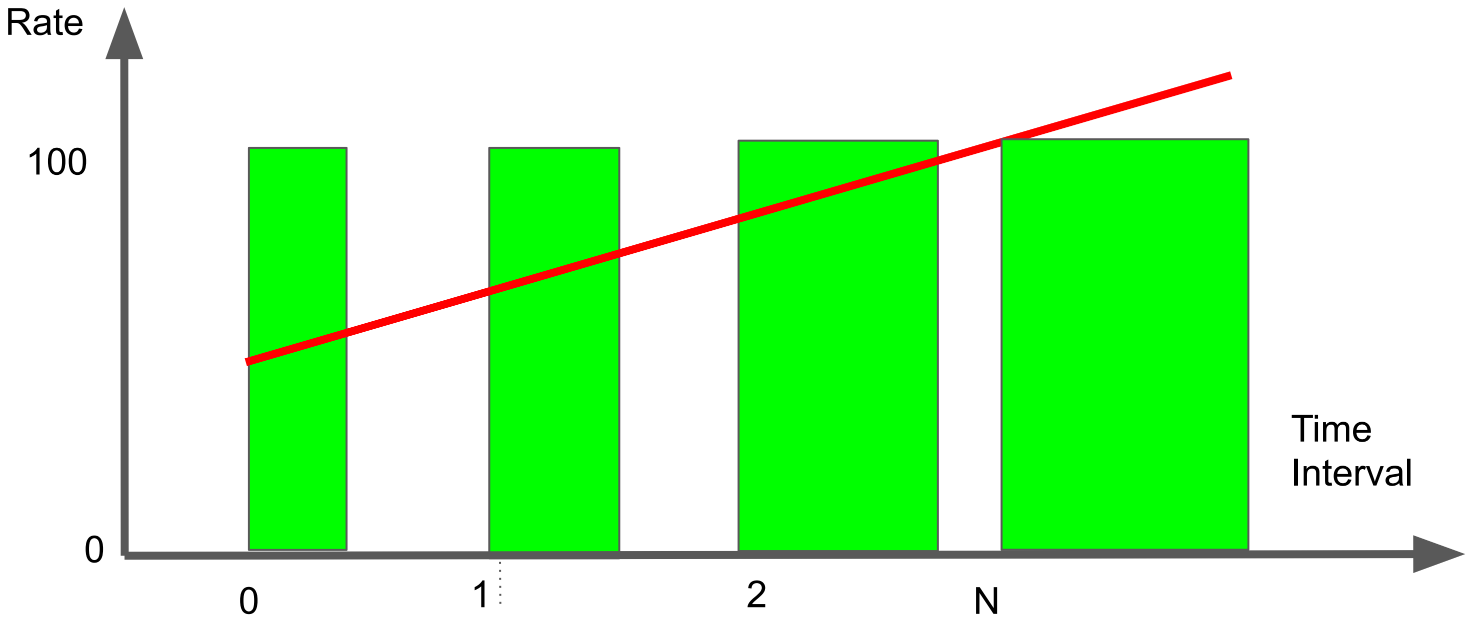
Constant Rate of Work/Time



yogini -w rate1-100



Variable Work/Time



Work != Utilization

Due to opportunistic turbo, 100% is often not attainable, or sustainable.

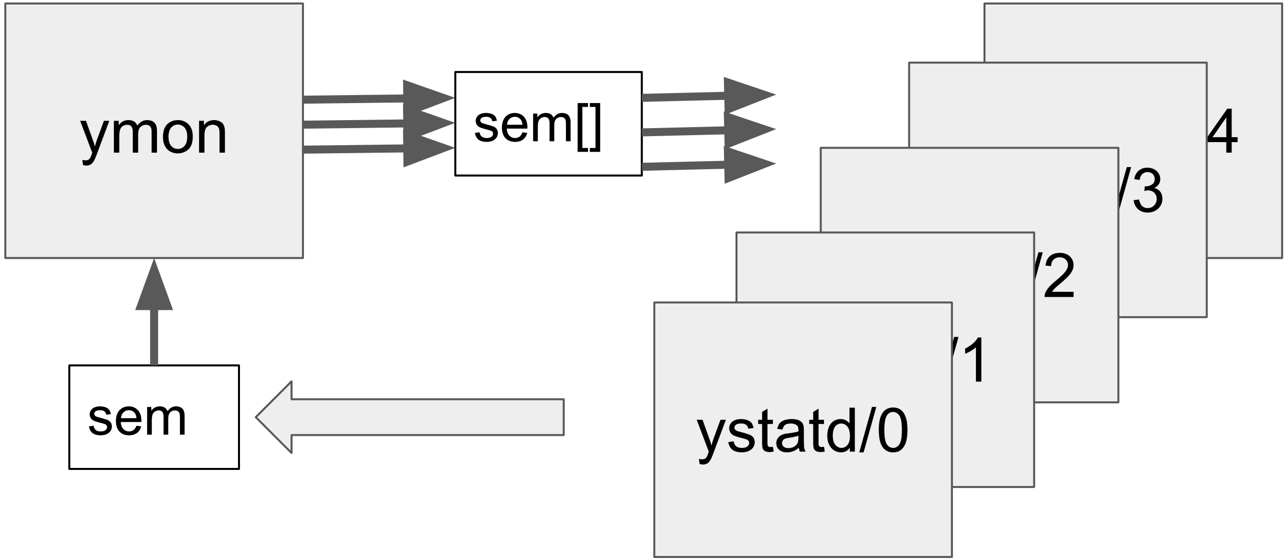
Yogini system monitor

monitor thread periodically* collects:

1. Utilization Busy% (per CPU)
2. RunQ length (per CPU)
3. Frequency (per CPU)
4. Linux run queue length (per CPU)
5. IRQs (per CPU)
6. Instructions per Cycle (IPC)
7. Temperature (per CPU DTS)
8. RAPL power (package, CPU, GFX, RAM, Uncore)

* **`--monitor wakemsec250`** (default 250 msec)

system monitor architecture



system monitor use

Monitor system for 10 sec:

```
# yogini
```

Fork my_program, monitor system until it exits:

```
# yogini my_program
```

Run built in AVX workload, monitor* until it exits

```
# yogini -w AVX
```

* skip monitor with `--monitor off`

Library of Built in Workloads

```
# yogini -w $WORKLOAD
```

WORKLOAD in:

1. GETCPU, RDTSC, PAUSE, TPAUSE, regAVX2, regVNNI

2. SSE, AVX, AVX2, AVX512, DOTPROD, VNNI

3. MEM, memcpy

Working Set Size

GETCPU, RDTSC, regAVX2, regVNNI, PAUSE, TPAUSE **[No size]**

SSE, AVX, AVX2, AVX512, DOTPROD, VNNI **[L1 dcache]**

MEM, memcpy [**L3 cache**]

Set working set size:

```
# yogini -w 256KB,AVX2 -w 100MB,MEM
```

worker thread instrumentation

Worker thread time slide granularity [16.66 ms]

Self record every time slice:

1. current CPU via getcpu(2)
2. work-done counter

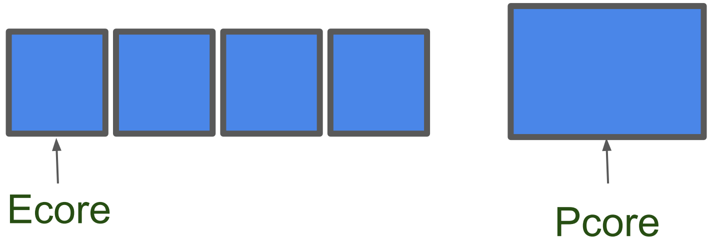
Set worker thread granularity to 10ms

```
# yogini -w wake-msec10
```

Linux EAS test on 4xEcore + 1xPcore

Task Placement:

1. Pcore
2. Ecore
3. Pcore HT sibling



EAS: Ecores more efficient than Pcores at low MHz

Example Ramp-Down on Big-Little

```
# yogini -w rate100-1
```

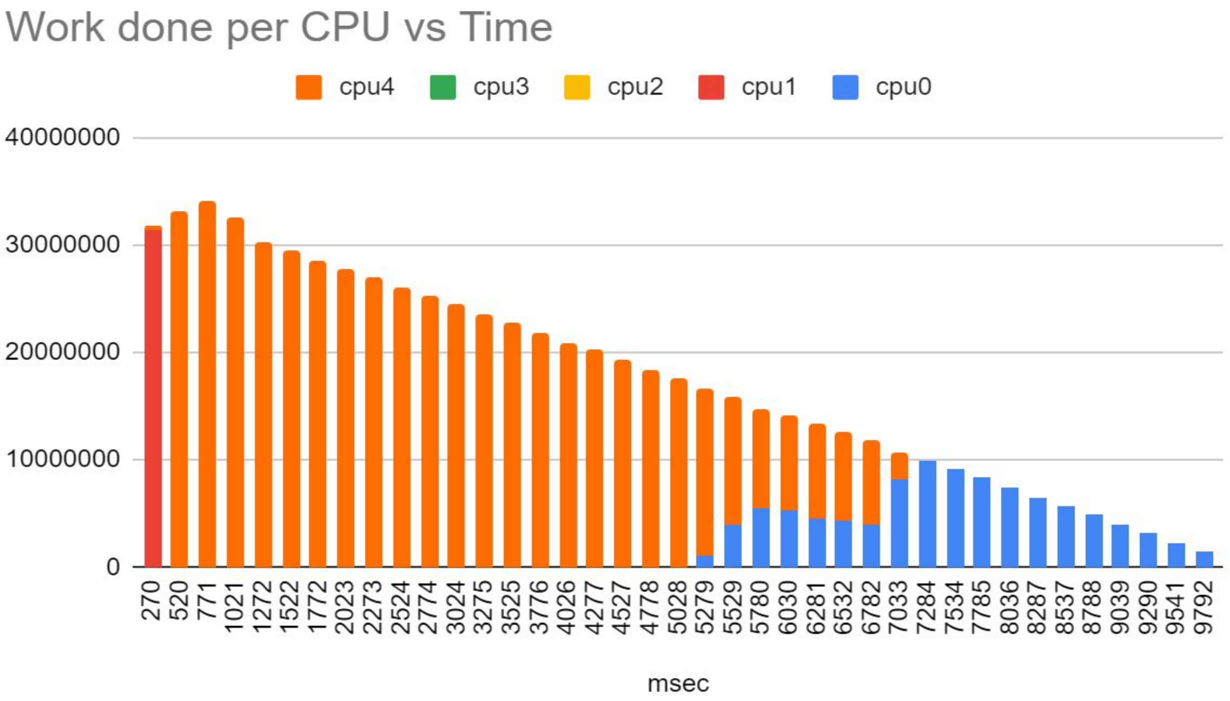
One thread

Requests 100% capacity, ramping down to 1%

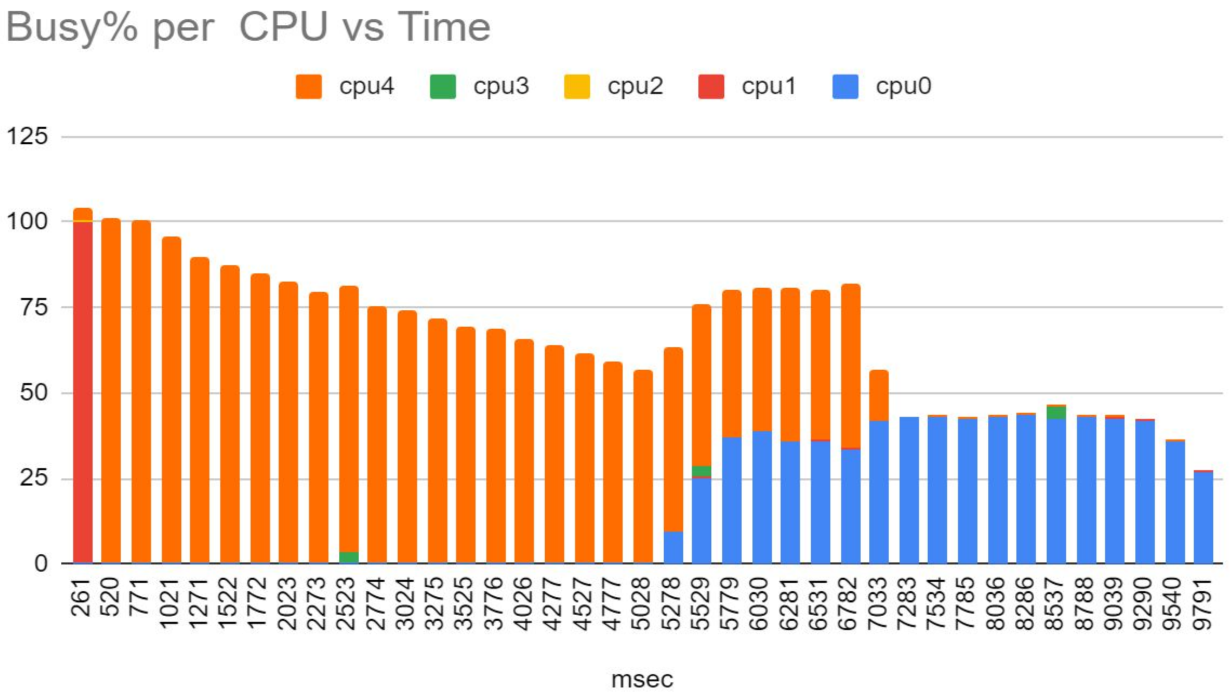
Energy model marks Pcore4 as less efficient.

Expect: high demand to run on Pcore4, migrate to Ecore upon low demand

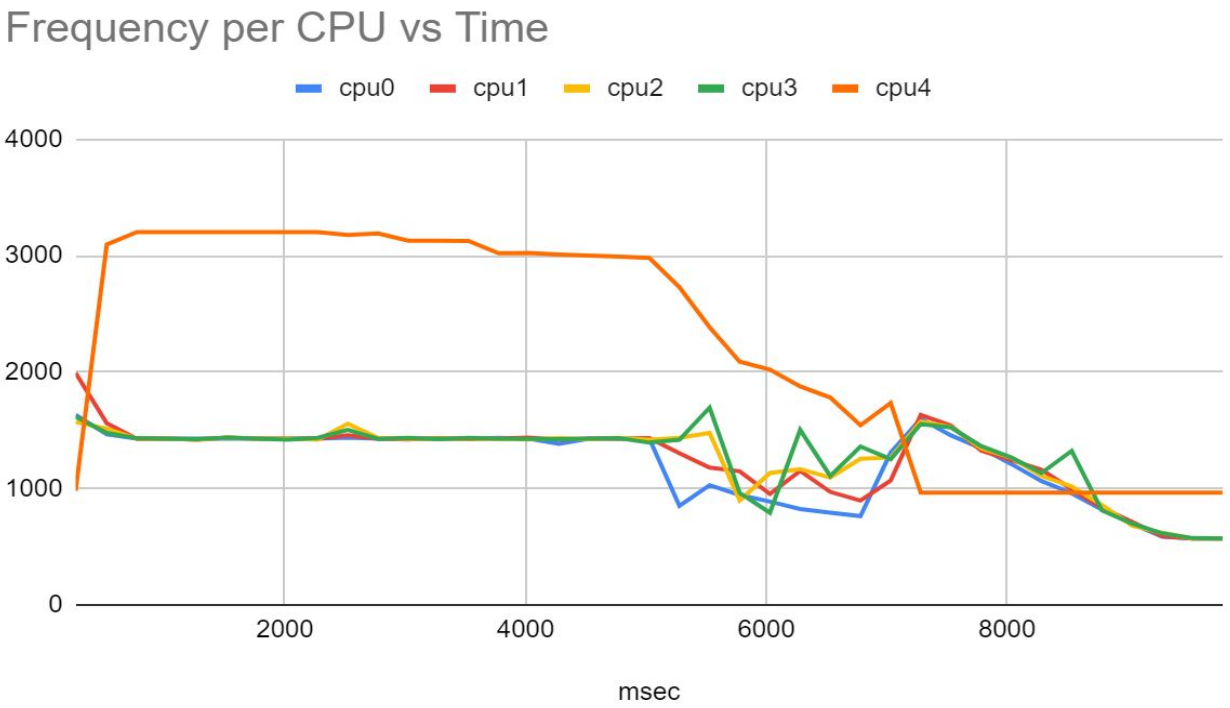
Example Ramp-Down (Big -> Little) - Work Done



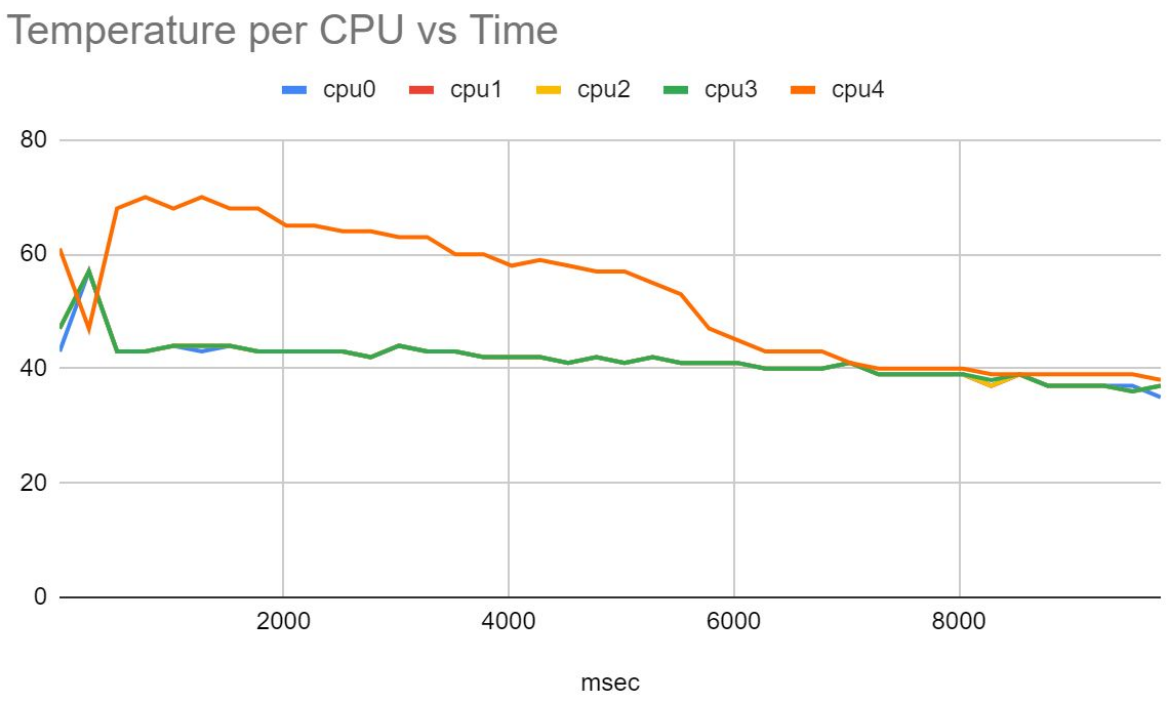
Example Ramp-Down (Big -> Little) %Busy



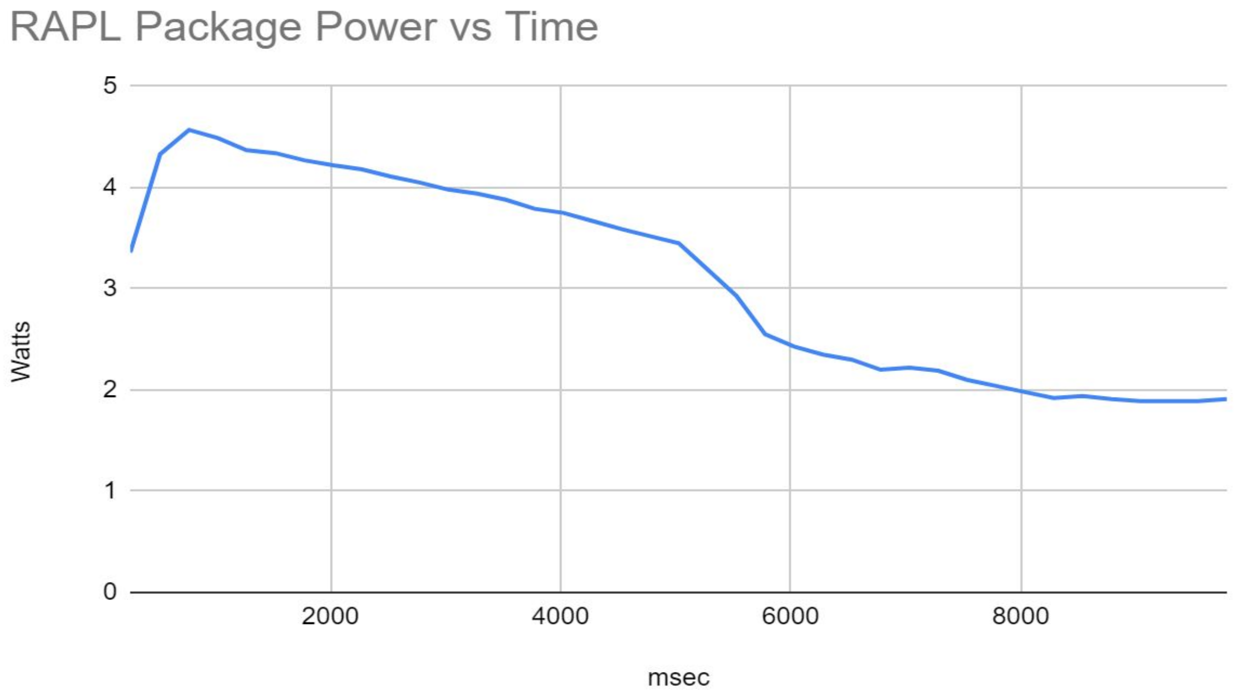
Example Ramp Down - Frequency



Example Ramp Down Temperature



Example Ramp Down - Power



Example Ramp Down: Summary

Summary Report:

- 100 Percent of Requested Throughput Achieved.
- 3.12 Average Watts
- 52 Task Migrations detected

Subjective Observations:

- Small->Big transition could have been faster
- Big -> small transition went meta-stable, but eventually worked

What's Next?

What workloads are "interesting"?

Regression test scenarios?

Is .tsv the ultimate output?

Best way to distribute?