Power Efficiency in HEP
(a case between ARM and x86)

Outline

- Motivations and previous work
- Methodology
  - Available hardware
  - Exporter tools & IPMI validation
- General tests with various workloads
  - C benchmarks & compiler flags
  - ATLAS full G4 simulations
  - HEP-Score containers
- Final focus:
  - Conclusions and future plans
  - HEP-Score

Introduction

- The power consumption of computing is coming under intense scrutiny worldwide, driven both by concerns about the carbon footprint, and by rapidly rising energy costs.

- ARM chips, while widely used in mobile devices due to their power efficiency, are not currently in widespread use as capacity hardware on the Worldwide LHC Computing Grid (WLCG).

- LHC experiments are increasingly able to compile their workloads on the ARM architecture (and … GPUs) to take advantage of various HPC facilities (e.g., ATLAS, CMS).

- To test whether WLCG sites have scenarios where power efficiency can be improved by deploying ARM-based hardware, the energy consumption and execution speed of identical CPU- and RAM-intensive workloads on two almost identical machines were tested: one with an Ampere **arm64** CPU, and the other with a standard AMD **x86_64** CPU.

- The workloads range from compiled C programs to typical HEP workloads (full ATLAS simulations and the most recent HEP-Score containerized jobs developed for Run3).

ScotGrid Glasgow: Emanuele Simili, Gordon Stewart, Samuel Skipsey, Dwayne Spiteri, David Britton

Special Thanks: Domenico Giordano (CERN), Gonzalo Menendez Borge (CERN), Johannes Elmsheuser (CERN)

Available Hardware

We have recently purchased two almost identical machines of comparable price, one with an AMD **x86_64** CPU, the other with an Ampere **arm64** CPU:

x86_64: Single AMD EPYC 7003 series (SuperMicro)

CPU:	AMD EPYC 7643 48C/96T @ 2.3GHz (TDP 300W)
RAM:	256GB (16 x 16GB) DDR4 3200MHz
HDD:	3.84TB Samsung PM9A3 M.2 (2280)

arm64: Single socket Ampere Altra Processor (SuperMicro)

CPU:	ARM Q80-30 80 core 210W TDP processor
RAM:	256GB (16 x 16GB) DDR4 3200MHz
HDD:	3.84TB Samsung PM9A3 M.2 (2280)

And we included in the comparison a standard workernode of our Grid cluster, with 2 AMD **x86_64** CPUs, which is also comparable in price* to the machines above:

2*x86_64: Dual AMD EPYC 7513 series Processors (DELL)

CPU:	2 * AMD EPYC 7513, 32C/64T @2.6GHz (TDP 200W)
RAM:	512GB (16 x 32GB) DDR4 3200MHz
HDD:	3.84TB SSD SATA Read Intensive

* this machine is part of a 2 unit / 4 node chassis.

And we also have a GPU node …

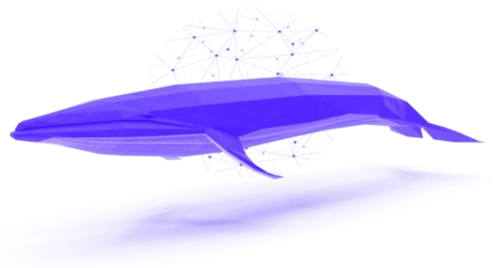
Power Readings

Power readings are achieved by two custom scripts, which collect and export CPU, RAM and IPMI Power metrics:

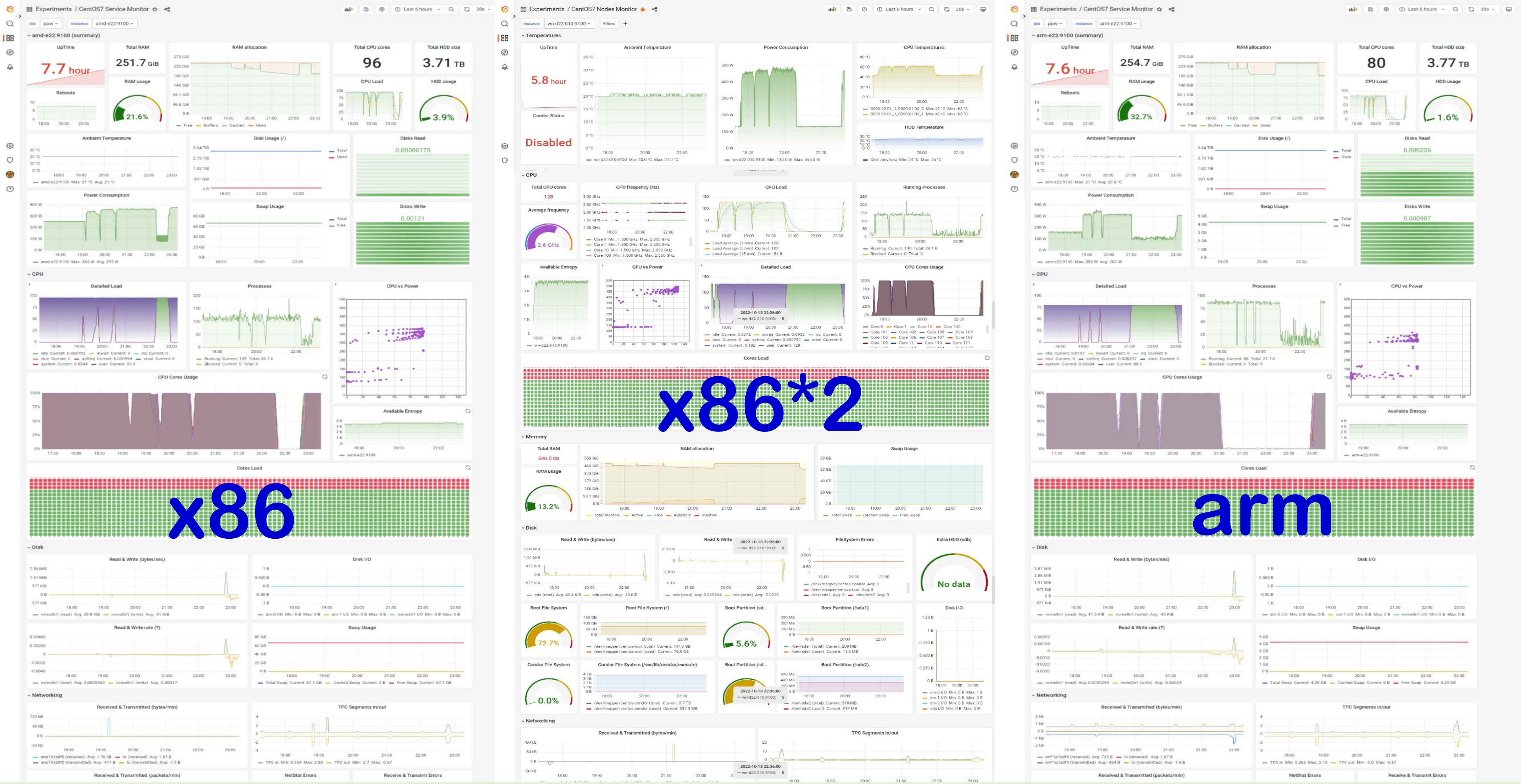
- ❖ The 1st script is a <u>cron job</u> (by **root**) that every 10 seconds exports IPMI power readings with a timestamp to `/tmp/ipmidump.txt` (… because *IPMItool* requires **root** privileges).

- ❖ The 2nd script is executed by the **user** and it takes in the job to be benchmarked. It starts by grabbing the IPMI readings from the dump file, attaches few more info (CPU, RAM) and appends them to a CSV file. After 1 min. sleep (so to measure idle power), it runs the given job, and waits another min. after the job is finished before quitting.

After the job is done, the CSV file is exported locally, processed in ROOT (time profiles plots and power integration), and cumulative results are visualized in Excel.



In addition, all servers are running a *node_exporter* client, which feeds (almost) real time metrics to our *Prometheus* server, which in turn feeds an extensive set of *Grafana* dashboards for easy visualization and monitoring purposes.



Site Monitoring @ **vCHEP2021**:

https://indico.cern.ch/event/948465/contributions/4323666/attachments/2248127/3813306/MonitoringAndAutomation_vCHEP2021.pdf

IPMI validation

As we didn’t have the best tools, automated logging from the external monitors was not an option:

- Instantaneous power was impossible to compare, as the number changed too quickly on the metered plugs and they almost never matched the IPMI readings from the machine.
- We did an integrated measurement of the total energy for a fixed time idle and for a complete job - which unfortunately did not have the same duration on both machines.

Idle test (30 min.)

x86: 0.04766 kWh ~ 0.046 kWh (=0.021+0.025 kWh) → Δ = 0.00166 kWh (= -3.5%% of IPMI reading)
arm: 0.05668 kWh ~ 0.054 kWh (=0.024+0.030 kWh) → Δ = 0.00268 kWh (= -4.7% of IPMI reading)

Job test (x86 ~45 min. ; arm ~30 min.)

x86: 0.25729 kWh ~ 0.263 kWh (=0.128+0.135 kWh) → Δ = 0.00571 kWh (= +2.2% of IPMI reading)
arm: 0.13418 kWh ~ 0.134 kWh (=0.064+0.070 kWh) → Δ = 0.00018 kWh (= +0.1% of IPMI reading)

Measurements of energy consumption over a longer duration of at least 30 minutes yielded results which were within **±5%** of the values recorded via IPMI, giving us confidence in the validity of our IPMI results.

Benchmarks

Custom benchmarks (specific purpose bits):

❖ Idle measurement (*sleep*)

❖ Prime number sieve (C with OMP)

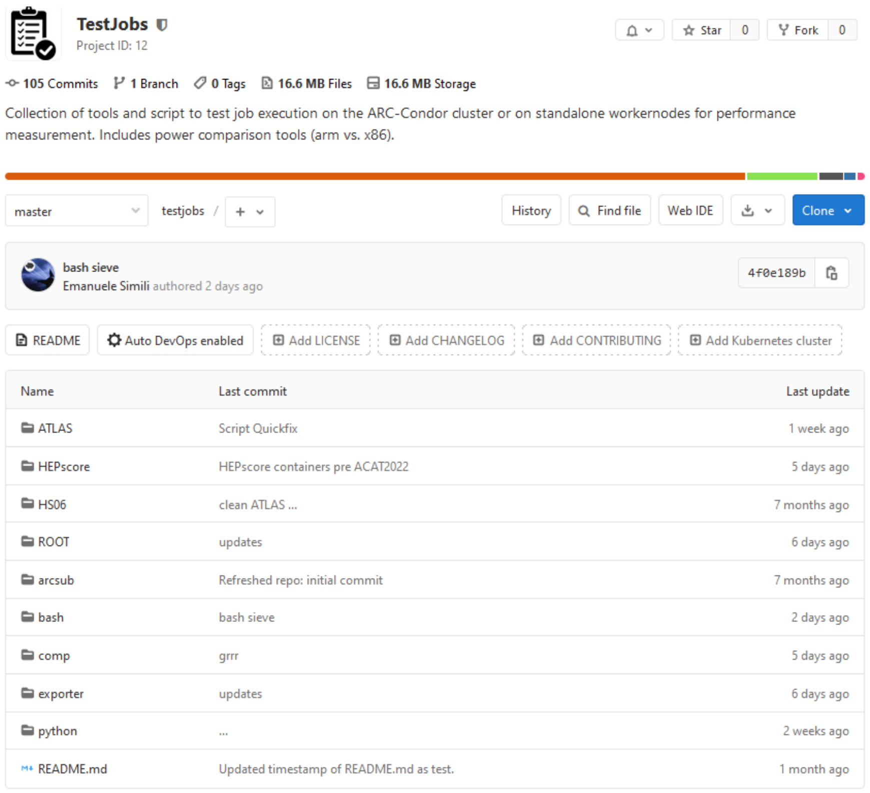
❖ Large Matrix Multiplication (C with OMP) using *int* , *float* and *double*

HEP benchmarks (typical Grid workload):

❖ Full G4MT ATLAS Simulation (sim-digi)

❖ HEP-Score containers (CMS, ATLAS)

We have created a project in our local GitLab repo to collect test-jobs and benchmarks that we have used in various occasion to run tests in our Tier2 cluster.

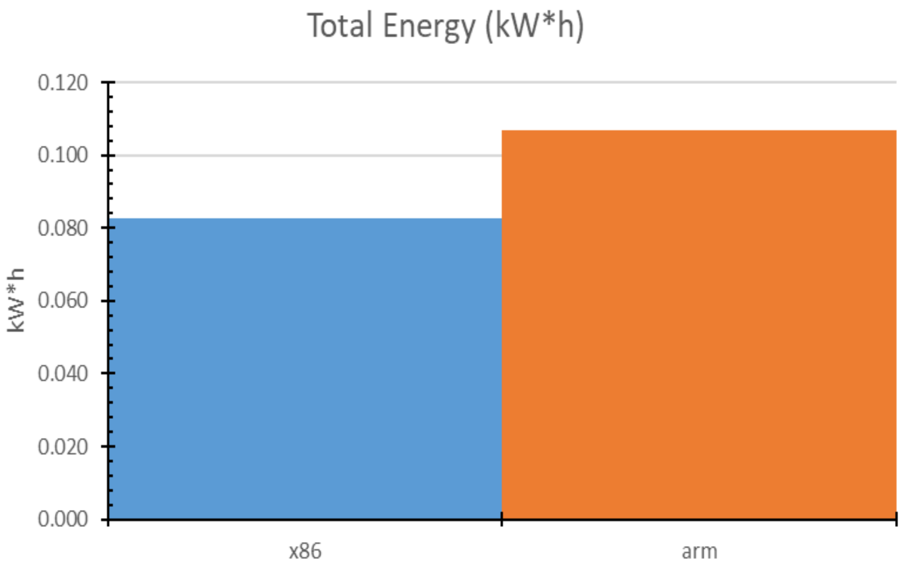


Idle Power

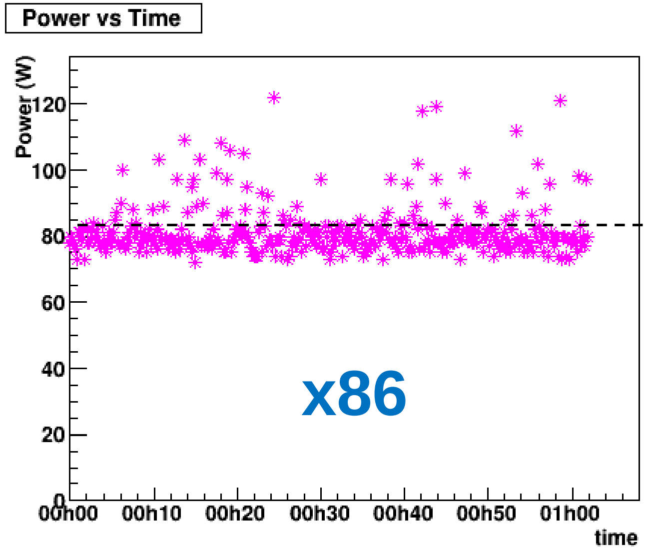
For this measurement we just let the machines idle for 1h, while collecting power metrics. In order to use the IPMI exporter script, we set to execute a *sleep* job:

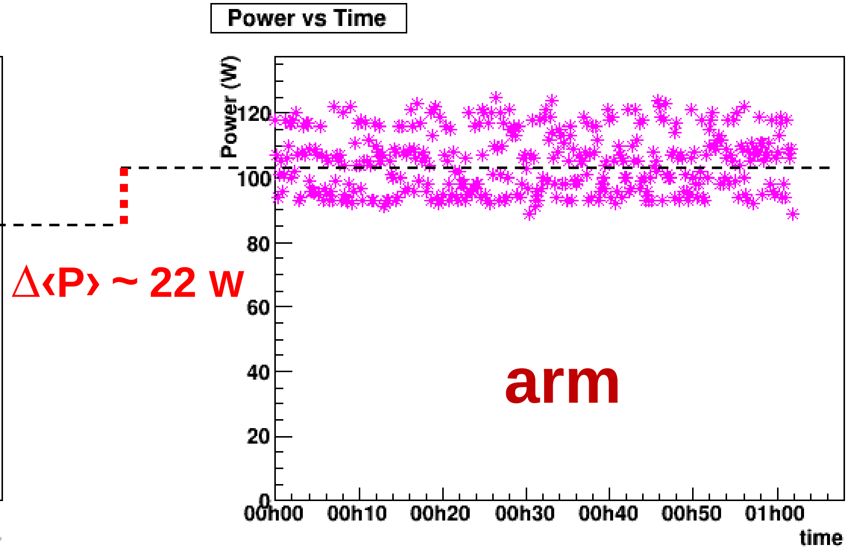
```
$ sleep 3600
```

Power profiles show that the **arm** has a higher baseline, and larger oscillations between power states, leading to a higher average *



lower = better





Machine	threads	Time (s)	Time (h)	Tot. Energy (	Peak Power (W)	Idle (W)
x86	96	3600	01:00:00	0.083	122.0	81.8
arm	80	3600	01:00:00	0.107	125.0	103.8

Key result: the **arm** uses about 30% more energy than the **x86** in idle state, making I/O bound tasks slightly less power efficient on **arm** than on an equivalent **x86** server.

* Note: the two CPUs have a different number of physical cores !

C benchmarks

As a quick and easy set of benchmarks that fully use the CPU, we have created two small C programs with `OpenMP` (Open Multi-Processing): the Eratosthenes' prime number sieve & large matrix multiplication.

- ❖ Eratosthenes' prime number sieve to find primes up to 100M
  - implemented in standard C with `OpenMP`
  - compiled with GCC 11.3 (from CVMFS)

```
$ gcc -fopenmp mprimes.c
```

```
#include <omp.h>
...
#pragma omp parallel for schedule(dynamic) reduction(+ : primes)
    for (num = 1; num <= limit; num++)
    {
...
```

- ❖ Large matrix multiplication using two 50k*50k random matrices (in 3 flavours):
  - 3 basic types: *int* (4 bytes), *float* (4 bytes), *double* (8 bytes)
  - implemented in standard C with `OpenMP`
  - compiled with GCC 11.3 & the `-mcmodel=large` flag
  - plus a few optimization flags … **(*)** see next slide

x86

```
$ gcc -fopenmp -mcmodel=large -Ofast -march=znver3 \
  matmult.c
```

arm

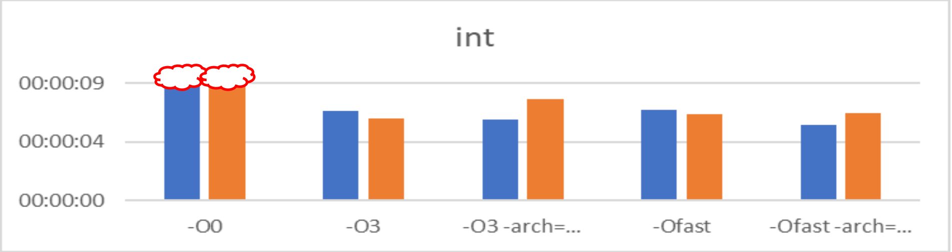
```
$ gcc -fopenmp -mcmodel=large -Ofast -mcpu=ampere1 \
  matmult.c
```

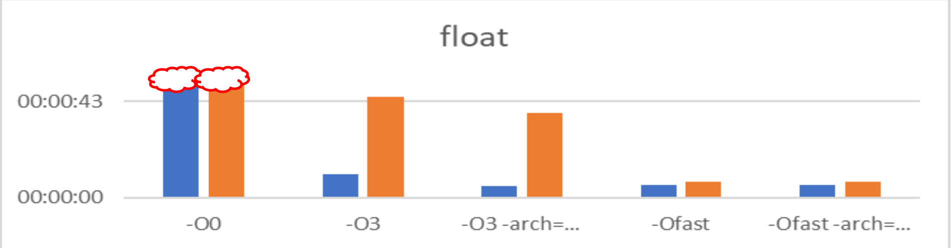
```
#include <omp.h>
...
#define N 50000
...
    #pragma omp parallel for private(i,j,k) shared(A,B,C)
    for (i = 0; i < N; ++i)
    {
        for (j = 0; j < N; ++j)
        {
            for (k = 0; k < N; ++k)
            {
                C[i][j] += A[i][k] * B[k][j];
...
```

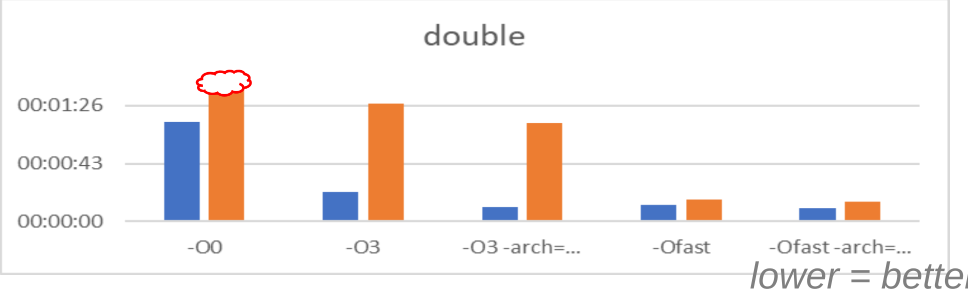
Compiler Flags (*)

Turns out that playing with GCC compiler flags opens a Pandora's box: there are thousands of options, and execution time varies wildly depending on them, especially floating point operations (*float* & *double*)

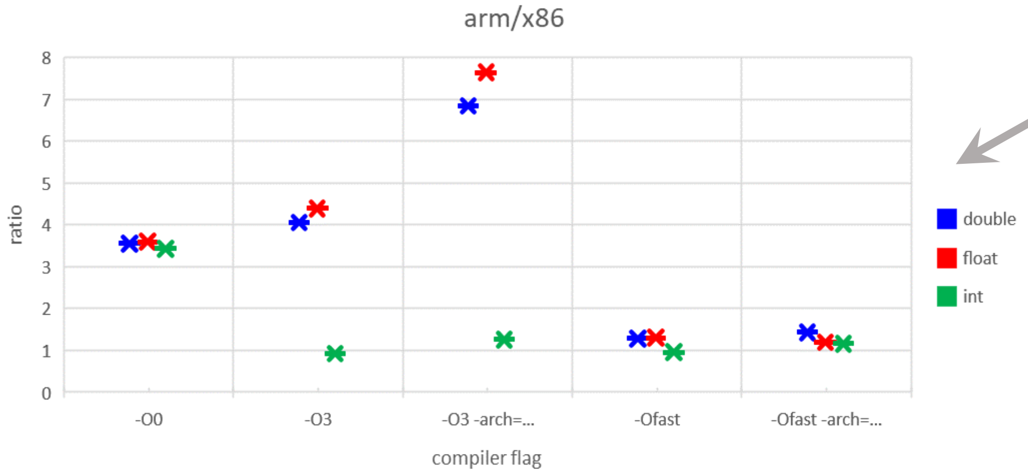
Test	Optimisation	march / mcpu flag	Type	x86	arm	arm/x86
intmatmul.c	-O0		int	00:01:14	00:04:14	3.42
intmatmul.c	-O3		int	00:00:07	00:00:06	0.92
intmatmul.c	-O3 -arch=…	march=znver3 / mcpu=ampere1	int	00:00:06	00:00:07	1.26
intmatmul.c	-Ofast		int	00:00:07	00:00:06	0.95
intmatmul.c	-Ofast -arch=…	march=znver3 / mcpu=ampere1	int	00:00:06	00:00:06	1.16
flomatmul.c	-O0		float	00:01:15	00:04:29	3.59
flomatmul.c	-O3		float	00:00:10	00:00:45	4.39
flomatmul.c	-O3 -arch=…	march=znver3 / mcpu=ampere1	float	00:00:05	00:00:38	7.64
flomatmul.c	-Ofast		float	00:00:06	00:00:07	1.30
flomatmul.c	-Ofast -arch=…	march=znver3 / mcpu=ampere1	float	00:00:06	00:00:07	1.19
doumatmul.c	-O0		double	00:01:14	00:04:22	3.55
doumatmul.c	-O3		double	00:00:22	00:01:28	4.05
doumatmul.c	-O3 -arch=…	march=znver3 / mcpu=ampere1	double	00:00:11	00:01:14	6.84
doumatmul.c	-Ofast		double	00:00:13	00:00:16	1.28
doumatmul.c	-Ofast -arch=…	march=znver3 / mcpu=ampere1	double	00:00:10	00:00:15	1.43







lower = better



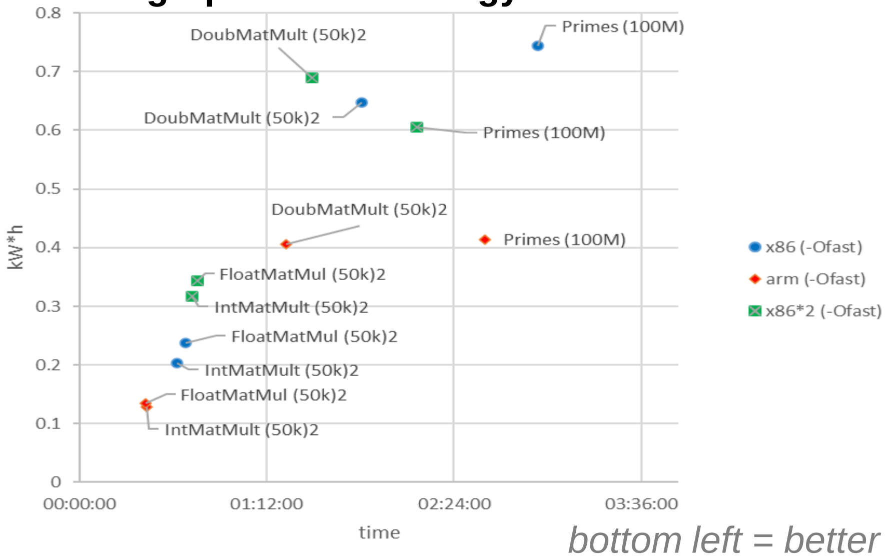
We did a quick study to determine the minimal set of flags that would give a comparable execution times on the two arch. From what we saw, the **arm** becomes competitive with the option `-Ofast` (i.e., disregard strict standards compliance) **!**

Results (C)

We compared the integrated energy consumption over the job duration for the three machines:

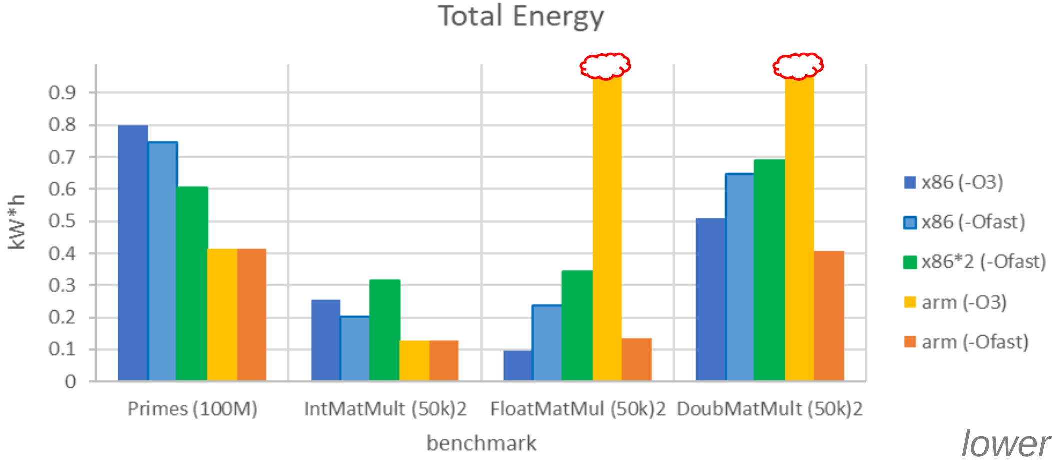
Arch	Threads	Benchmark	Time (hh:mm:ss)	Energy (kW*h)	RAM max(Gb)	idle (W)	Pow. max (W)
x86 (-Ofast)	96	Primes (100M)	02:56:21	0.74488	7.7	84	260
x86 (-Ofast)	96	IntMatMult $(50k)^2$	00:37:44	0.20398	64.3	99	347
x86 (-Ofast)	96	FloatMatMul $(50k)^2$	00:41:04	0.23691	64.3	86	363
x86 (-Ofast)	96	DoubMatMult $(50k)^2$	01:48:29	0.64739	121.2	88	373
x86*2 (-Ofast)	128	Primes (100M)	02:09:56	0.60546	6.6	134	303
x86*2 (-Ofast)	128	IntMatMult $(50k)^2$	00:43:15	0.31687	34.8	134	470
x86*2 (-Ofast)	128	FloatMatMul $(50k)^2$	00:45:36	0.34398	34.8	135	482
x86*2 (-Ofast)	128	DoubMatMult $(50k)^2$	01:29:16	0.68929	63.3	136	492
arm (-Ofast)	80	Primes (100M)	02:36:03	0.41426	7	106	169
arm (-Ofast)	80	IntMatMult $(50k)^2$	00:25:55	0.12782	64	113	331
arm (-Ofast)	80	FloatMatMul $(50k)^2$	00:25:26	0.13543	63.9	116	359
arm (-Ofast)	80	DoubMatMult $(50k)^2$	01:19:18	0.40635	120.1	102	324

2D graph of total Energy & execution time:

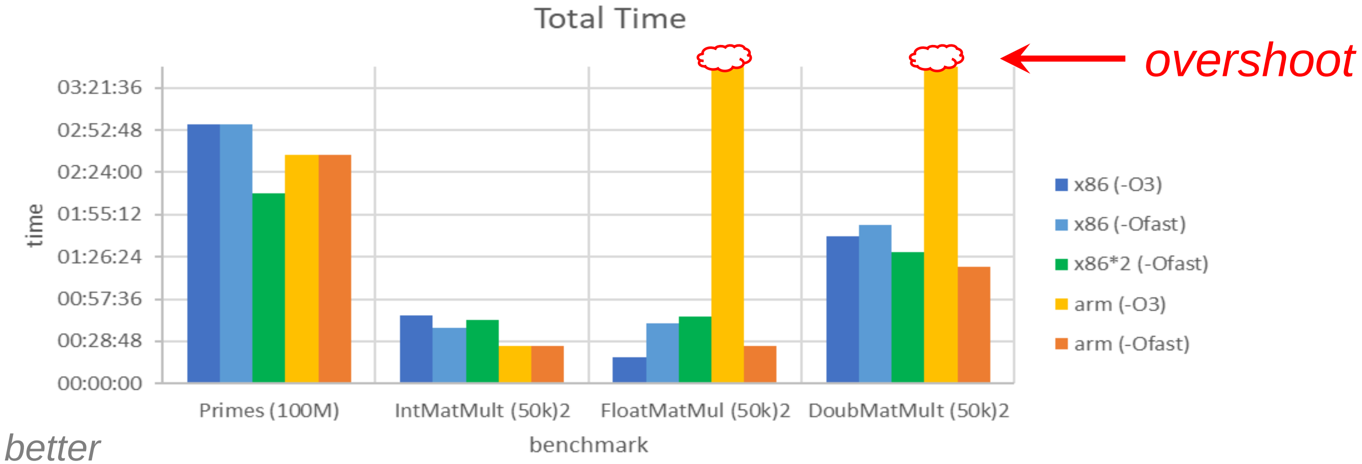


bottom left = better

The histograms below include samples from **x86** and **amd** with a different optimization flag (-O3):



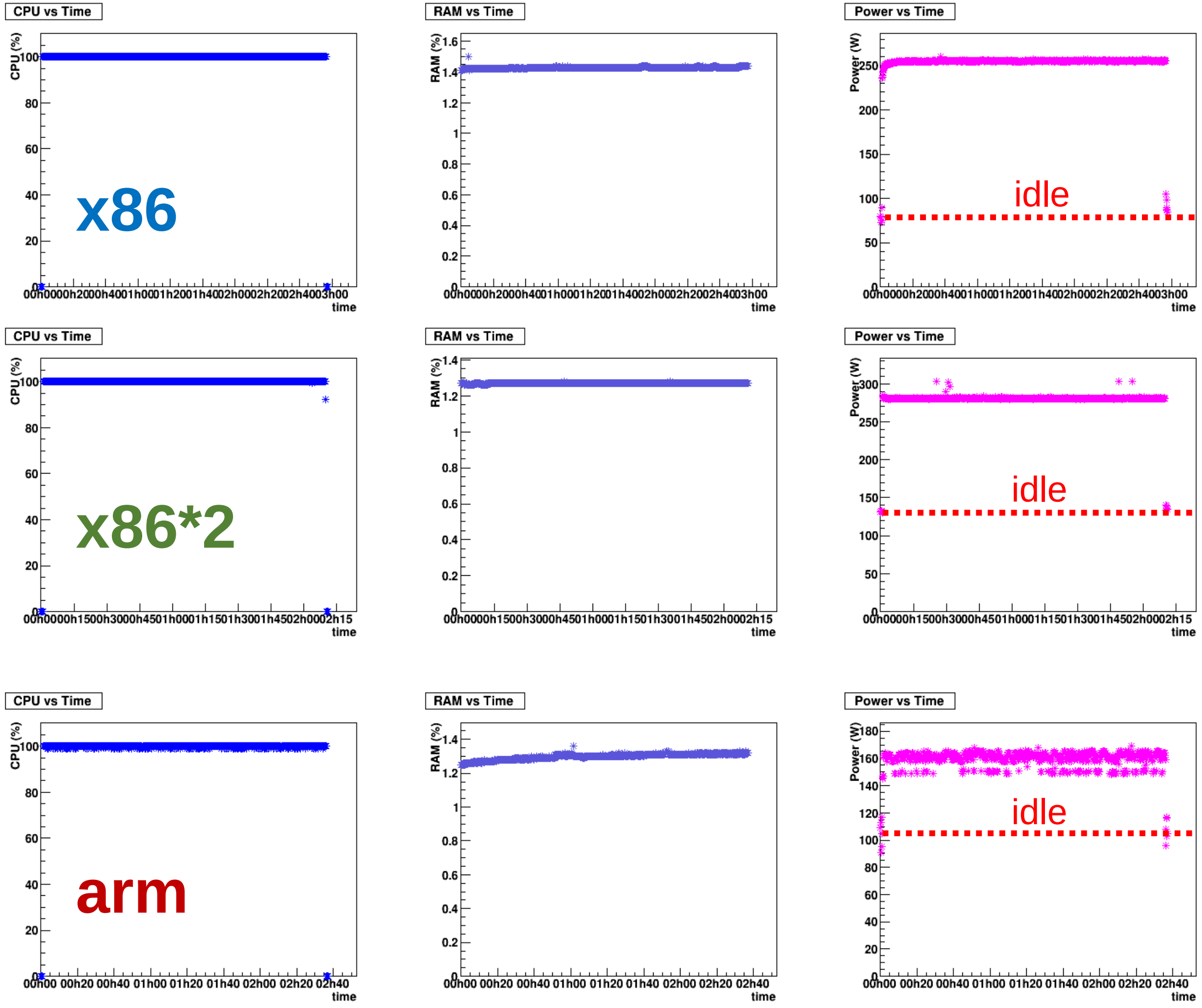
lower = better



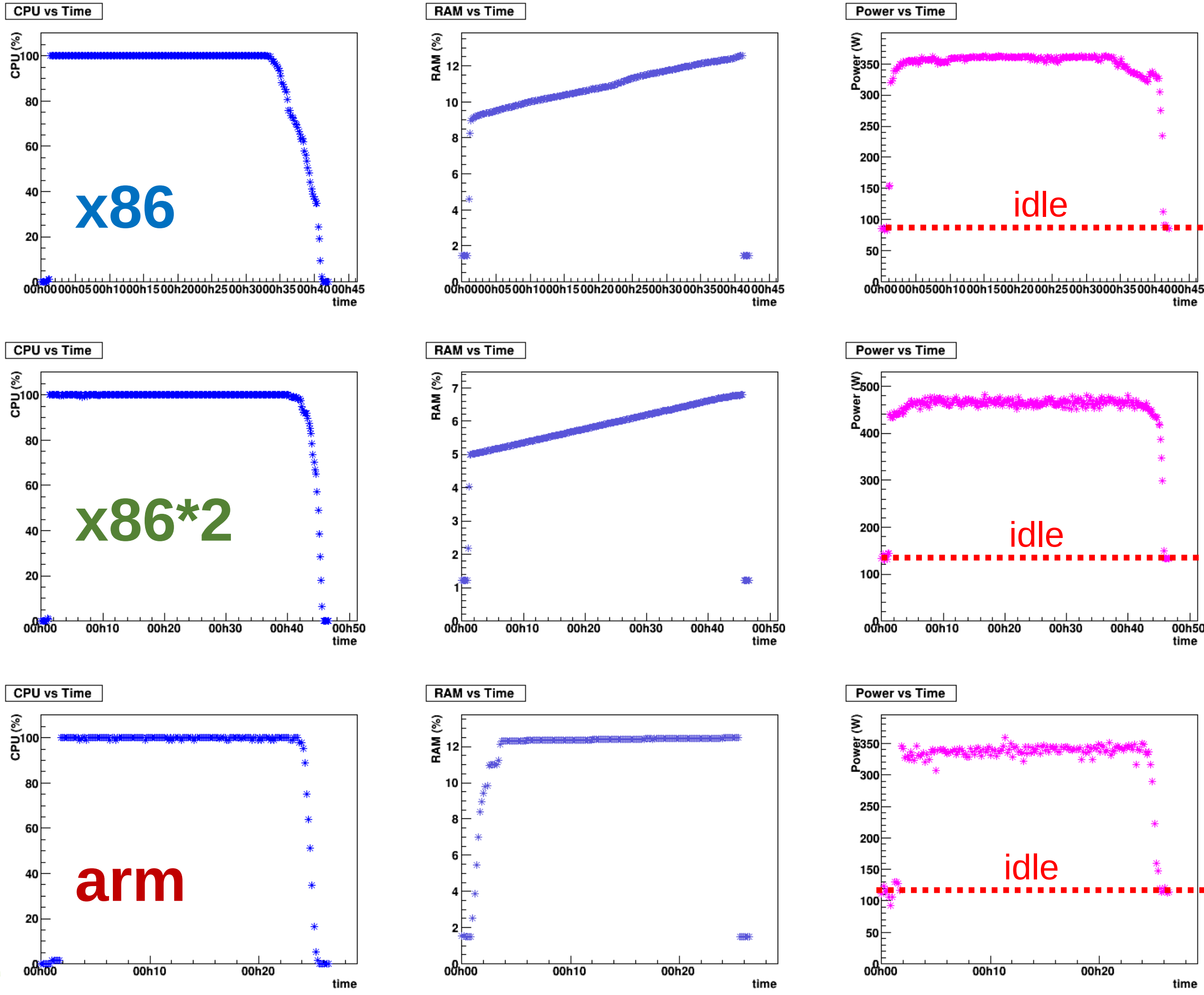
Job Profiles (C)

C benchmarks job profiles (CPU, RAM, Power) vs. Time:

Prime Number sieve (1 to 100M)



Large matrix multiplication (50k^2 *float*)



ATLAS Workload

The chosen version of the software: **Athena 23.0.3**
(builds available for both **x86_64** and **aarch64** on CVMFS)

x86 `Using Athena/23.0.3 [cmake] with platform x86_64-centos7-gcc11-opt at /cvmfs/atlas.cern.ch/repo/sw/software/23.0`

arm `Using Athena/23.0.3 [cmake] with platform aarch64-centos7-gcc11-opt at /cvmfs/atlas.cern.ch/repo/sw/software/23.0`

Setting up the ATLAS framework on CVMFS and running the job:

```
$ export ATLAS_LOCAL_ROOT_BASE=/cvmfs/atlas.cern.ch/repo/ATLASLocalRootBase
$ alias setupATLAS='source ${ATLAS_LOCAL_ROOT_BASE}/user/atlasLocalSetup.sh'
$ setupATLAS

$ asetup Athena,23.0.3
$ ./TTbarSim2022.sh 1000
```

TTbarSim2022.sh

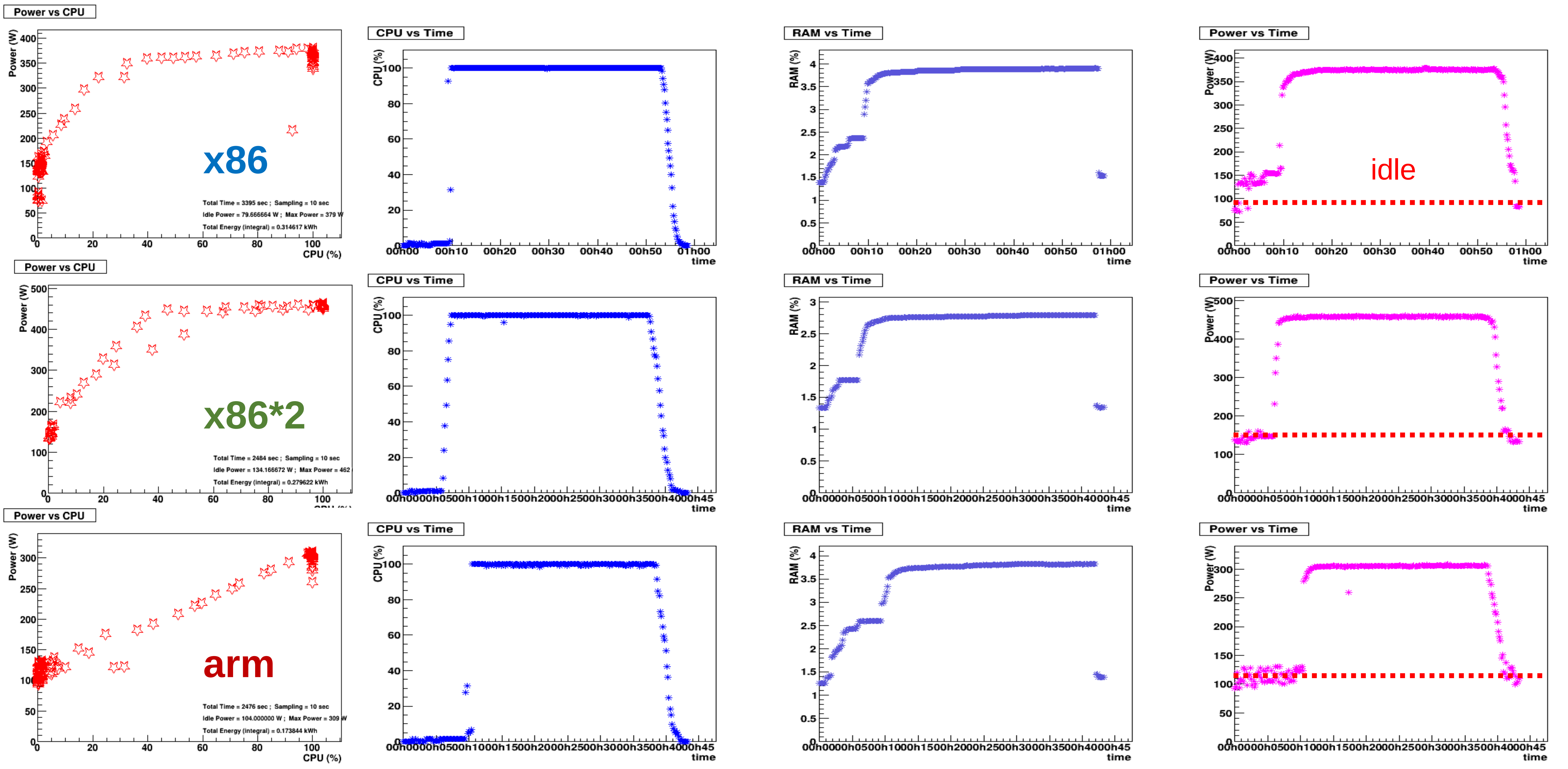
```
#!/bin/sh

export MAXEVENTS=10000
export ATHENA_CORE_NUMBER=$(nproc)
inputdatadir=/cvmfs/atlas.cern.ch/repo/benchmarks/hep-workloads/input-data
inputdata=$inputdatadir/EVNT.13043099._000859.pool.root.1

Sim_tf.py \
      --inputEVNTFile="${inputdata}" \
      --outputHITSFile="TTbar2022.HITS.pool.root" \
      --maxEvents=${MAXEVENTS} \
      --physicsList=FTFP_BERT_ATL \
      --imf=False \
      --randomSeed=6163 \
      --AMIConfig=s3873 \
      --multithreaded=True \
      --jobNumber=1 \
```

Input file:

Geant4 MT full ATLAS detector simulation of a given number of TTbar events (from existing TTbar gen-events file)

We generated samples of 1k and 10k events …

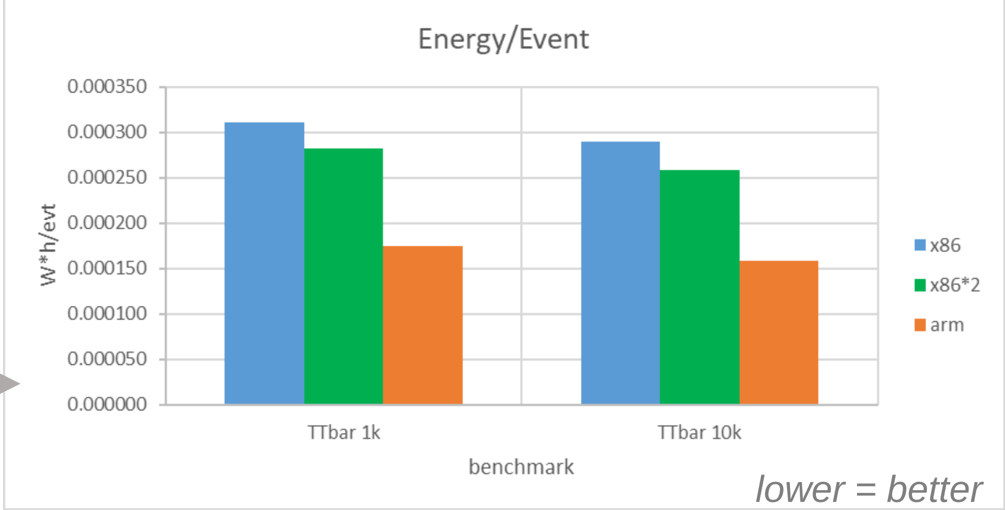
Job Profiles (ATLAS 1k)



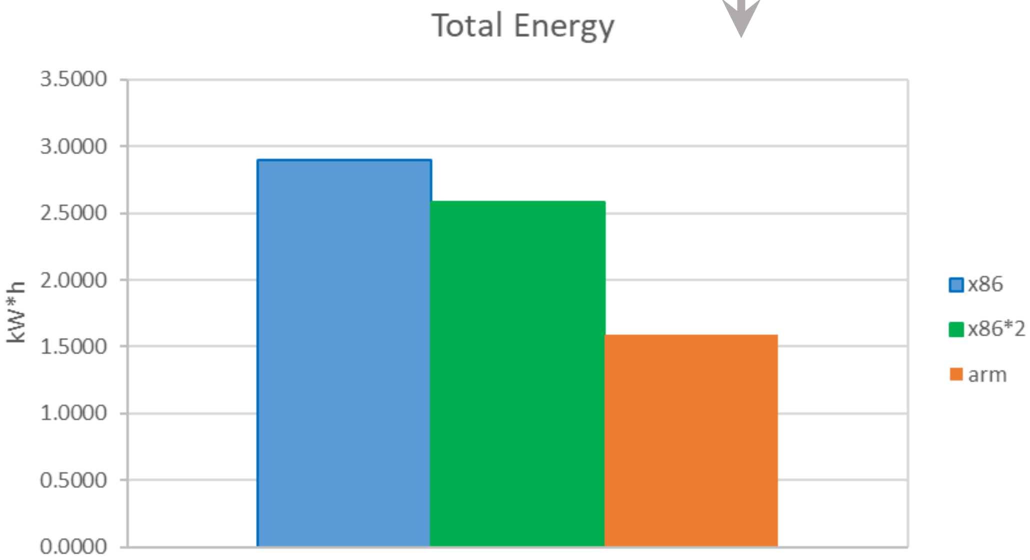
Results (ATLAS 10k)

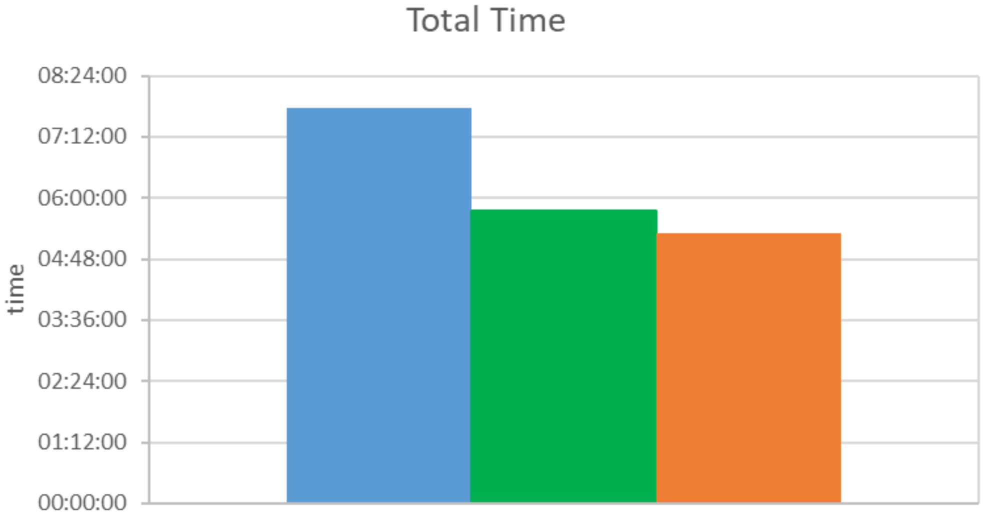
We compared execution time and integrated energy consumption over the job duration for the 3 types of machines available (x86, x86*2 and arm). Each job was executed three times, we took the average execution time and energy consumption, and their standard deviation as an estimate of the uncertainty.

Arch	Threads	ATLAS sim.	N. events	Time (h)	dT (%)	Energy(kW*h)	dE (%)	idle(W)	max(W)	W*h/event
x86	96	TTbar	10'000	07:46:14	0.3%	2.8966	0.6%	85	382	0.000290
x86*2	128	TTbar	10'000	05:44:25	0.4%	2.5843	0.6%	133	463	0.000258
arm	80	TTbar	10'000	05:19:07	2.2%	1.5853	2.5%	110	309	0.000159



The better efficiency of the arm machine compared to both the x86 can be seen in the **Total Energy** plot or in the **Energy/Event** plot.





Key result:
the arm is slightly faster, and much more energy efficient than x86 !

lower = better

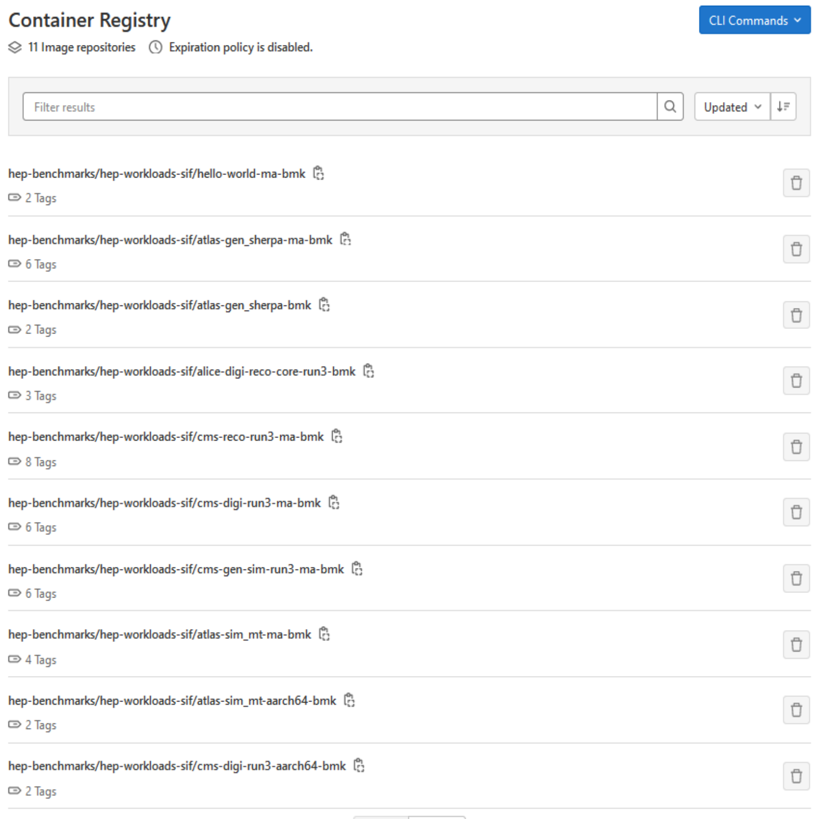
HEP-Score

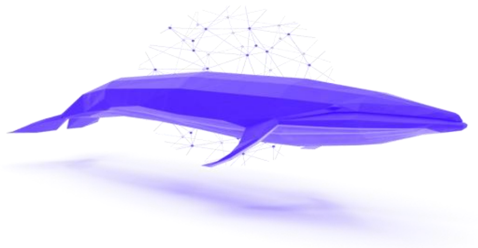
I started interacting with the HEP-Score Task Force earlier this year, because of mutual interest and partial overlap with my research on power efficiency. In particular:

- ❖ I wanted to use HEP-Score as a standard HEP workload to rate different architectures on power efficiency (which is becoming increasingly important for procurement),
- ❖ I helped testing HEP containers on locally available architectures (**x86** & **arm**),
- ❖ We discussed the option to include power readings in the standard output of the HEP-Score suite (peak / idle power & integrated energy consumption for a given workload).

While the full HEP benchmark suite is not yet available for **arm**, an increasing number of experimental workloads are available as standalone containers

https://gitlab.cern.ch/hep-benchmarks/hep-workloads-sif/container_registry





HEP-Score Containers

HEP-Score containers can run on **Singularity** (or **Docker**, which we do not use):

(x86) *Singularity-CE 3.9.9-1.el7* (previous version *3.8.7-1.el7* disappeared from EPEL and replaced with *AppTainer 1.1.0-1.el7*)

(arm) *singularity version 3.8.4-1.el7* (still available in EPEL)

Example execution of a containerised HEP-Score job:

```
$ mkdir –p /tmp/test/results
$ chmod a+rw /tmp/test/
$ singularity run -B /tmp/test:/results oras://registry.cern.ch/hep-workloads/cms-gen-sim-run3-bmk:latest
```

We used 5 HEP-Score containers from the container_registry (prev. slide):

gitlab-registry.cern.ch/hep-benchmarks/hep-workloads-sif/atlas-sim_mt-ma-bmk:v2.0
gitlab-registry.cern.ch/hep-benchmarks/hep-workloads-sif/atlas-gen_sherpa-ma-bmk:ci-v1.0
} ATLAS (2x)

gitlab-registry.cern.ch/hep-benchmarks/hep-workloads-sif/cms-reco-run3-ma-bmk:v1.1
gitlab-registry.cern.ch/hep-benchmarks/hep-workloads-sif/cms-digi-run3-ma-bmk:v1.0
gitlab-registry.cern.ch/hep-benchmarks/hep-workloads-sif/cms-gen-sim-run3-ma-bmk:v1.0
} CMS (3x)

Note: the HEP workloads are designed to scale with the number of available threads, therefore power consumption cannot be directly compared among machines with a different number of cores (/threads), as the machine with more threads would have done more work …

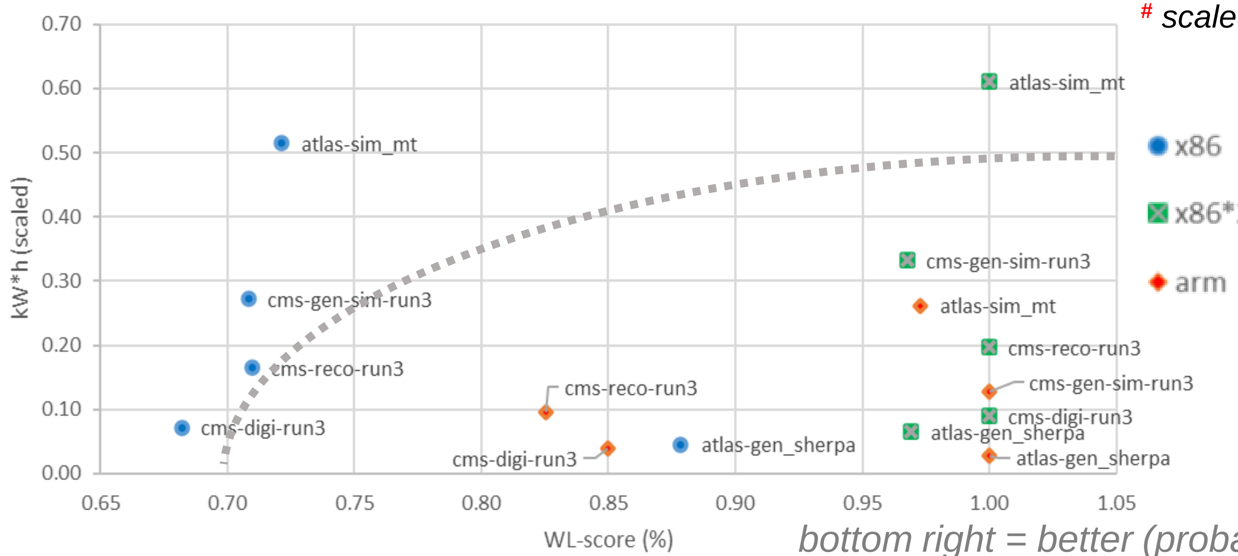
HEP-Score Results

Here the cumulative results of the 5 HEP-Score containers on the 3 machines (**x86**, **x86*2** and **arm**):

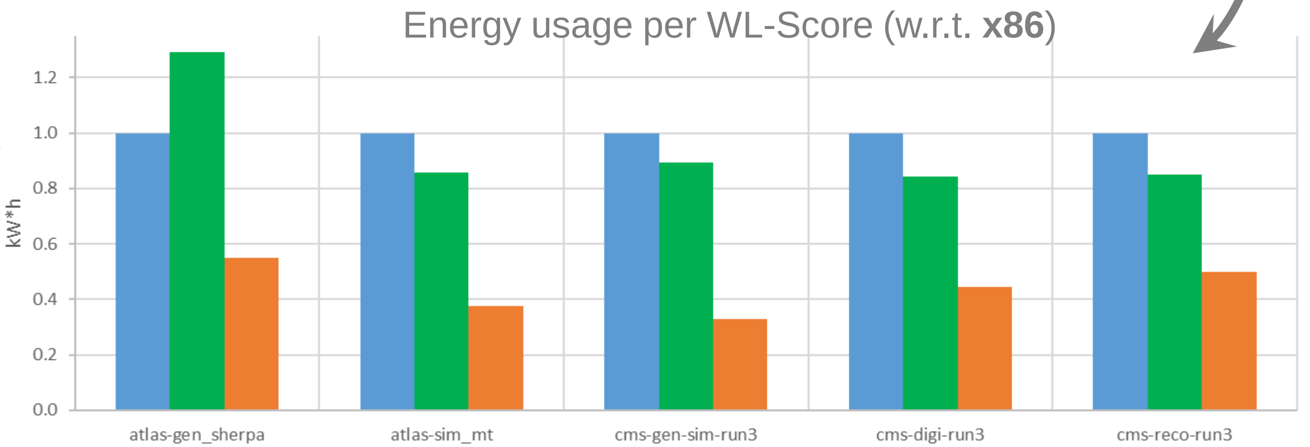
Arch	Threads	Benchmark	Time (hh:mm:ss)	Energy (kW*h)	Energy [#] (kW*h)	RAM max (Gb)	idle (W)	max (W)	WL-Score	WL-Score [@] (%)	E/WLs (wrt *x86*)
x86	96	atlas-gen_sherpa	00:08:21	0.04576	0.04576	154.8	94	390	113.3959	87.9%	1.0000
x86	96	atlas-sim_mt	01:23:49	0.51478	0.51478	54.4	82	383	0.4068	72.1%	1.0000
x86	96	cms-gen-sim-run3	00:44:13	0.27235	0.27235	31.3	98	385	3.7347	70.8%	1.0000
x86	96	cms-digi-run3	00:12:31	0.07224	0.07224	102.1	88	392	15.0538	68.2%	1.0000
x86	96	cms-reco-run3	00:26:53	0.16447	0.16447	119.6	86	389	6.4072	71.0%	1.0000
x86*2	128	atlas-gen_sherpa	00:10:11	0.06509	0.04882	207.9	133	488	125.0605	96.9%	1.2897
x86*2	128	atlas-sim_mt	01:20:58	0.6109	0.45818	74.0	133	473	0.5639	100.0%	0.8561
x86*2	128	cms-gen-sim-run3	00:43:14	0.3319	0.24893	43.2	134	477	5.1025	96.8%	0.8920
x86*2	128	cms-digi-run3	00:12:41	0.08937	0.06703	138.2	133	494	22.0662	100.0%	0.8440
x86*2	128	cms-reco-run3	00:26:03	0.19734	0.14801	160.9	134	483	9.0284	100.0%	0.8515
arm	80	atlas-gen_sherpa	00:06:21	0.02856	0.03427	126.4	108	348	129.0706	100.0%	0.5483
arm	80	atlas-sim_mt	00:56:30	0.26156	0.31387	64.1	104	310	0.5486	97.3%	0.3767
arm	80	cms-gen-sim-run3	00:26:44	0.12691	0.15229	148.8	116	306	5.2721	100.0%	0.3301
arm	80	cms-digi-run3	00:09:01	0.04008	0.04810	183.9	104	318	18.7561	85.0%	0.4453
arm	80	cms-reco-run3	00:20:34	0.09574	0.11489	238.0	102	310	7.4504	82.5%	0.5006

[#] *scaled to **x86** (= 96 threads)*

[@] *renormalised (highest = 100%)*



bottom right = better (probably)

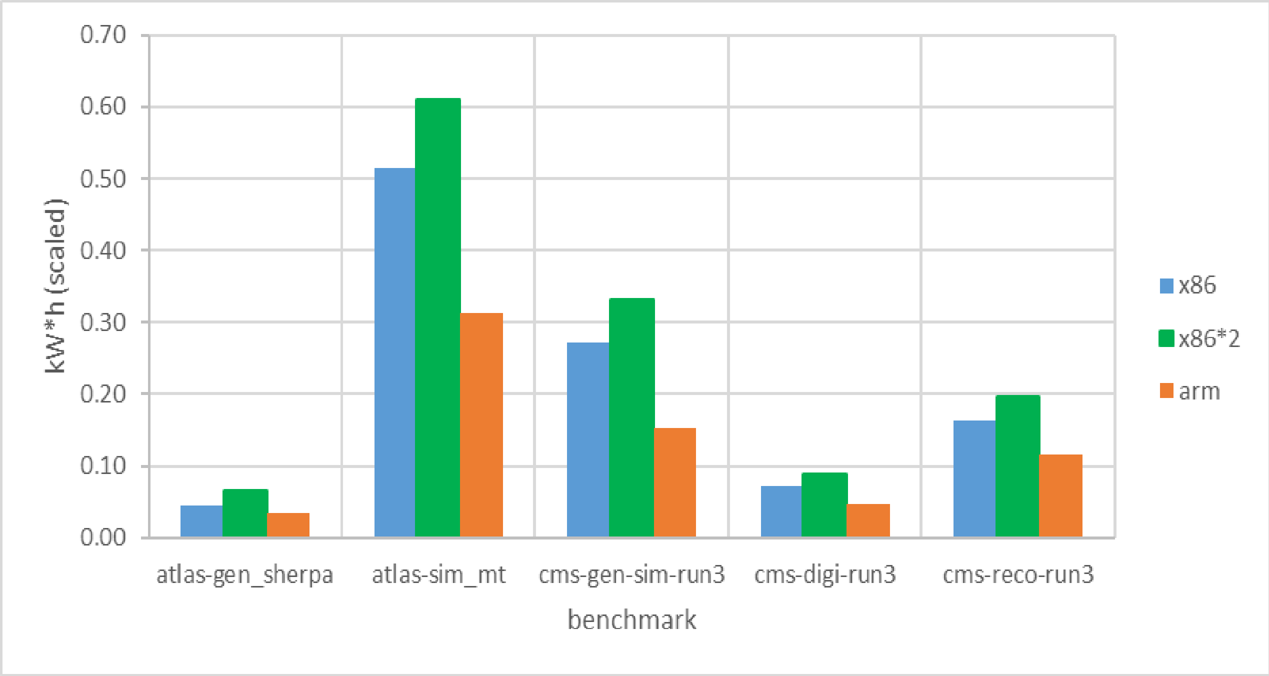


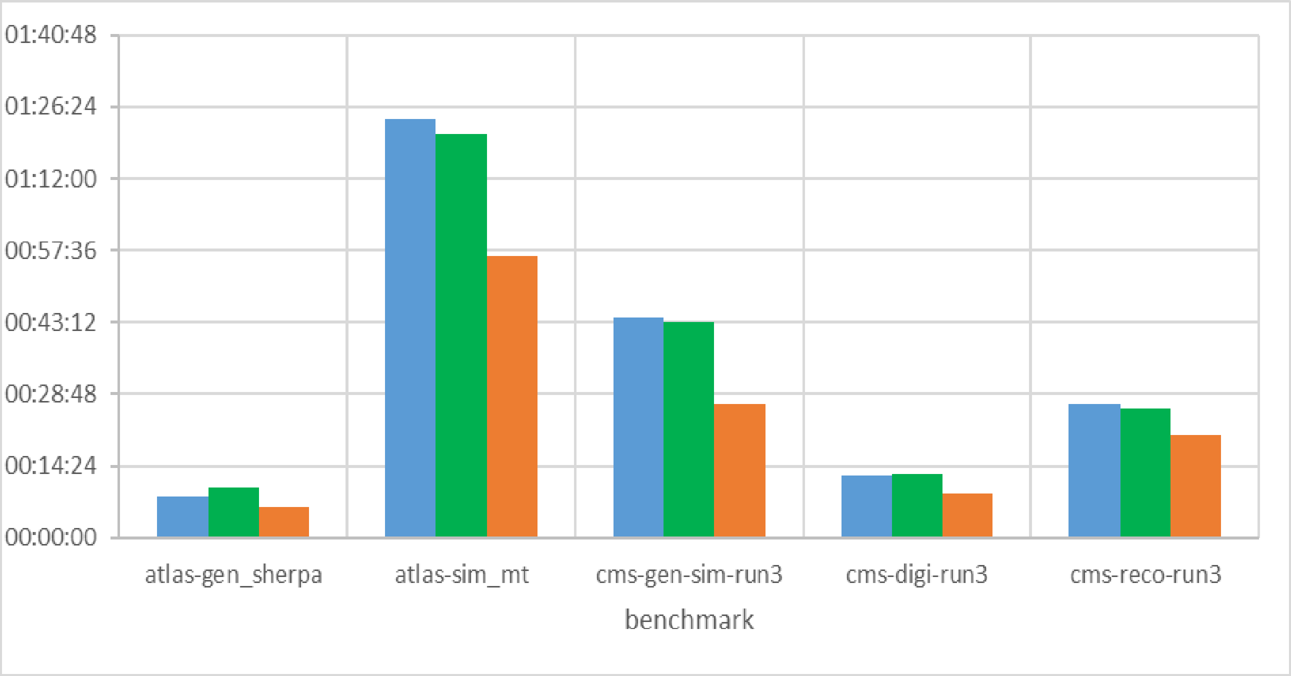
lower = better

HEP-Score Summary

In conclusion, we compared an **arm** and **x86** processor of very similar specs and almost identical cost. We found that on average, the **arm** processor ran ~20% quicker and used ~35% less power per HEP task than the equivalent **x86** *

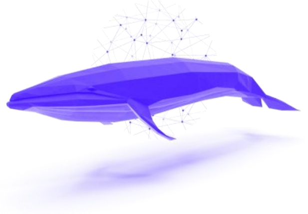
* *averages renormalized to the same amount of work*





Note:

while the HEP-Score geometric average is not yet available (as we did run standalone containers outside the HEP Benchmarking suite), we have been struggling to find an intuitive way to compare data …



Open Issues

- Despite the active development within all LHC experiments, support for architectures other than **x86** is patchy, at best ...

- Same is true for hardware suppliers, at least in our experience: our *DELL* supplier does not sell any **arm** server (yet). We ordered it from a small local supplier and it took over 2 months to arrive …

- The **arm** machine seem to score better in speed and energy efficiency, but we never really looked under the hood: ATLAS physics output was not validated, matrix elements were not checked one by one. So … <u>before committing to a new architecture, we must make sure that results are valid</u>!

- We trust enough our IPMI power readings (± 5%), but for more precise measurement we may want a more precise comparison (e.g., by using a metered PDU with remote power readings on each plug)

- There was no error propagation. We just took the 5% error on IPMI reading as the major source …

- All results presented here were prepared in about 1 month (due to the delay in the **arm** server delivery) … so, they might not be perfect (yet). Stay tuned for the next iteration of this study!

Summary & Outlook

- This study addressed almost all the limitation that affected of our previous one on power efficiency (ref. *GridPP47*), as the benchmark were now performed on two almost identical servers installed locally in a closely controlled environment.

- In almost all categories, we see that for the same price **arm architecture gives better performance in term of speed and power efficiency**.
  We also see that our **arm** server has a higher energy consumption in its idle state than its **x86** counterpart, which makes I/O bound operations slightly less power efficient.

- We wish to continue this study by performing a better estimate of all sources of errors, and by doing some sort of validation of the jobs output - to guarantee that the better performance does not come at the cost of a lower accuracy.

- We also wish to extend our set of benchmark to GPUs. We already have some hardware available and we are working in close contact with the HEP-Score task force (see yesterday's HEP talk by Domenico). This collaboration is likely to continue …

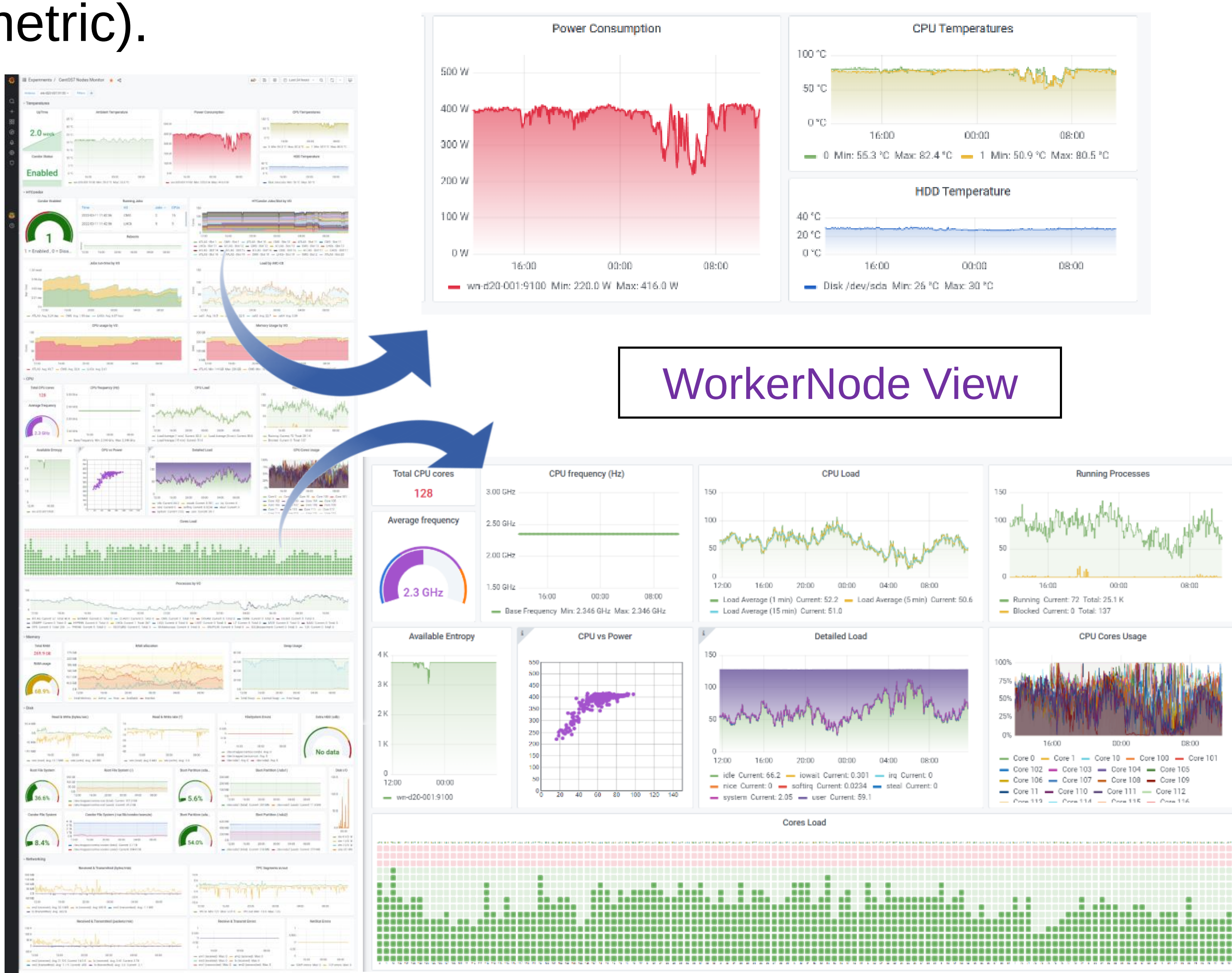
Thanks.

Visualization (local)

Local power readings and resource usage is extracted by *node_exporter* (which can be customized with any metric).

Data is colourfully visualised in a custom *Grafana* dashboard,

and can be exported to CSV format via the *Prometheus* API and analysed in ROOT



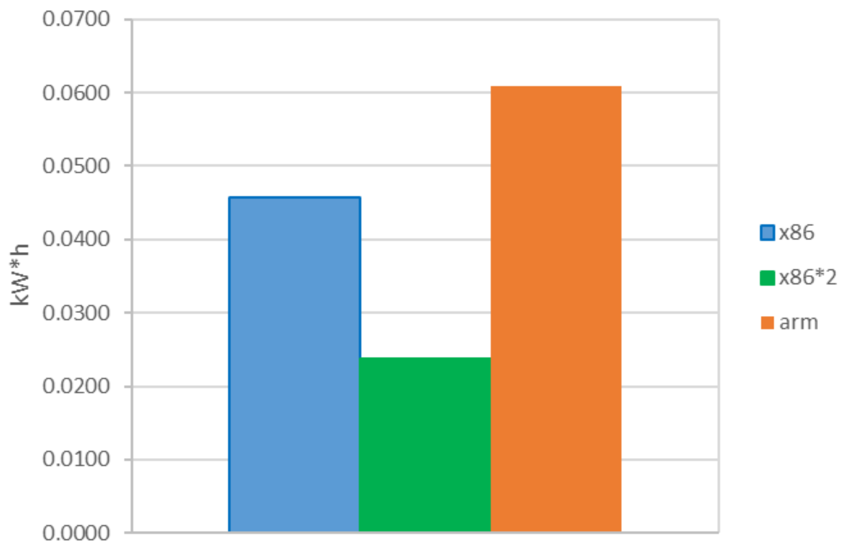
Exporter Script

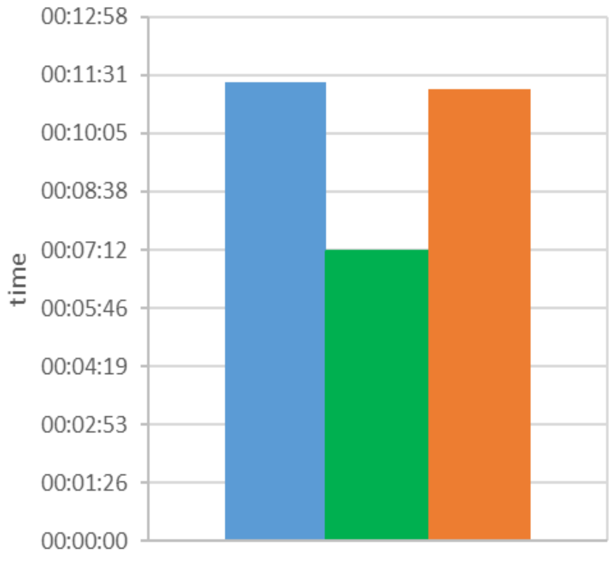
TimeStamp: `date +"%F , %T"`
CPU: `top -bn1 | grep "Cpu(s)" | sed "s/.*, *\([0-9.]*\)%* id.*/\1/" | awk '{print 100 - $1}'`
RAM: `free -t | awk 'FNR == 2 {printf("%.2f"), $3/$2*100}'`
GPU: `nvidia-smi --query-gpu=power.draw --format=csv,noheader,nounits | awk '{s+=$1} END {print s}’`
Power (IPMI): `ipmitool dcmi power reading | grep "Instantaneous power reading:"`

Results (BASH sieve)

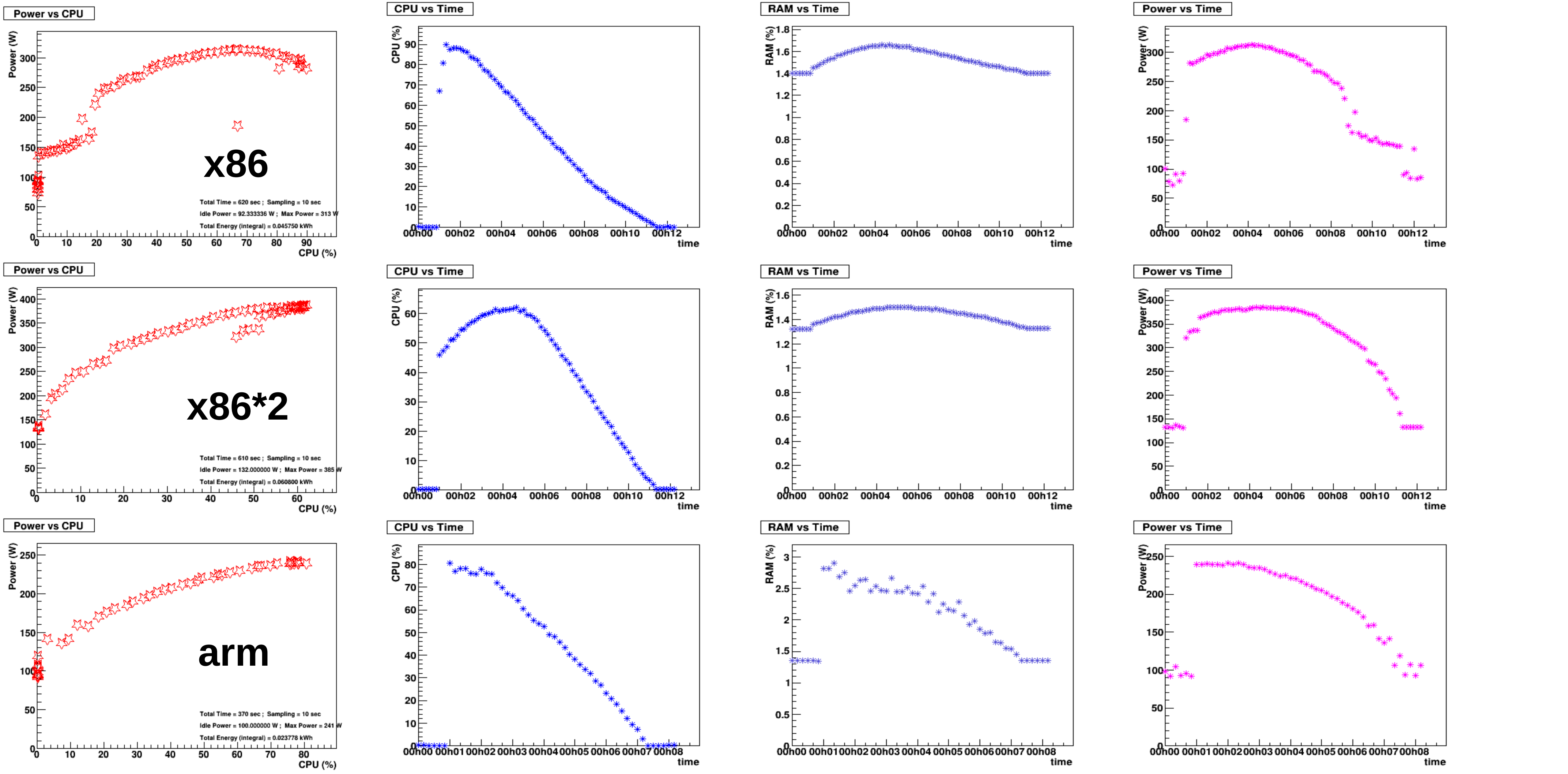
Erathostenes' prime numbers sieve implemented in BASH (thx RosettaCode), with added ad-hoc multithreading …

Arch	Threads	Time (h)	Time(s)	Energy(kW*h)	CPUmin(%)	CPUmax(%)	RAMmin(Gb)	RAMmax(Gb)	idle(W)	max(W)
x86	96	00:11:20	680	0.0458	0.1	89.8	3.6	4.2	92	313
x86*2	128	00:07:10	430	0.0238	0.1	80.6	3.4	7.4	100	241
arm	80	00:11:10	670	0.0608	0.2	62.1	3.4	3.8	132	385





BASH sieve


Power vs CPU
Power (W)
CPU (%)
x86
Total Time = 620 sec ; Sampling = 10 sec
Idle Power = 92.333336 W ; Max Power = 313 W
Total Energy (integral) = 0.045750 kWh
CPU vs Time
CPU (%)
time
RAM vs Time
RAM (%)
time
Power vs Time
Power (W)
time
Power vs CPU
x86*2
Total Time = 610 sec ; Sampling = 10 sec
Idle Power = 132.000000 W ; Max Power = 385 W
Total Energy (integral) = 0.060800 kWh
CPU vs Time
RAM vs Time
Power vs Time
Power vs CPU
arm
Total Time = 370 sec ; Sampling = 10 sec
Idle Power = 100.000000 W ; Max Power = 241 W
Total Energy (integral) = 0.023778 kWh
CPU vs Time
RAM vs Time
Power vs Time

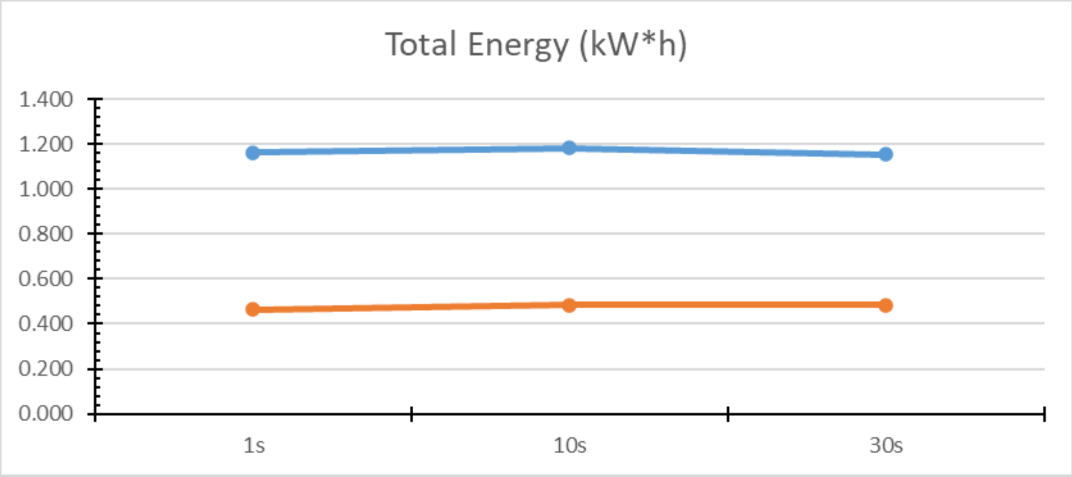
Sampling Frequency

Check that the sampling frequency we chose does not affect the integrated Energy consumption (the effect is less than 2%).

Prime Number sieve (1 to 100M) using compiled C code with OMP (gcc version 4.8.5 20150623)
IPMI sampling interval = 1sec , 10sec, 30sec

Job	Sampling	threads	Time (s)	Time (h)	Tot. Energy (kW*h)	Peak Power (W)	Idle (W)
x86	1s	96	10982	03:03:02	1.162	402.0	90.9
	10s	96	11002	03:03:22	1.183	402.0	84.0
	30s	96	11017	03:03:37	1.154	391.0	85.5
arm	1s	80	8259	02:17:39	0.465	215.0	103.9
	10s	80	8229	02:17:09	0.483	215.0	104.5
	30s	80	8229	02:17:09	0.485	220.0	100.5

dE (x86)	0.0123	1.1%
dE (arm)	0.0087	1.8%



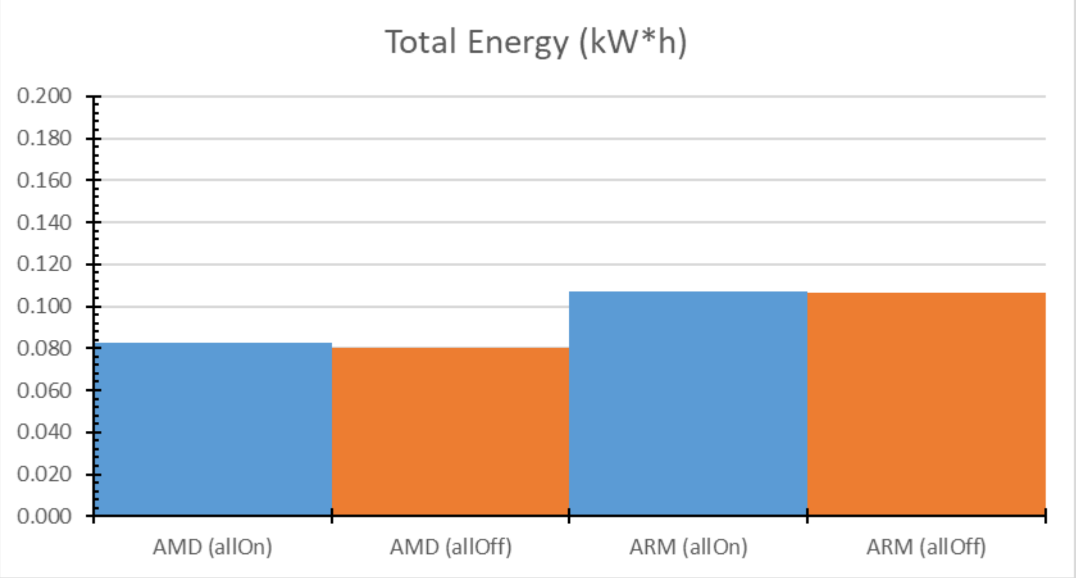
Exporters Effect

Check that the active exporters (*node_exporter*, *PromTail*, *Pakiti*) do not add extra power consumption.
(the effect seems to be about 2% … but it would be better to gather more samples)

Idle Job (1h) with/without exporters (node_exporter, PromTail, Pakiti, cron)

IPMI sampling interval = 10sec

Job	Exporters	threads	Time (s)	Time (h)	Tot. Energ	Peak Powe	Idle (W)
AMD	AMD (allOn)	96	3600	01:00:00	0.083	122.0	81.8
	AMD (allOff)	96	3600	01:00:00	0.080	137.0	76.3
ARM	ARM (allOn)	80	3600	01:00:00	0.107	125.0	103.8
	ARM (allOff)	80	3600	01:00:00	0.107	120.0	105.2



ATLAS stuff

Arch	Threads	Benchmark	N. events	Time (h)	dT (%)	Energy(kW*h)	dE (%)	idle(W)	max(W)	W*h/event
x86	96	TTbar 1k	1'000	00:55:42	3.0%	0.3105	1.2%	83	380	0.000310
x86	96	TTbar 10k	10'000	07:46:14	0.3%	2.8966	0.6%	85	382	0.000290
x86*2	128	TTbar 1k	1'000	00:42:44	0.8%	0.2814	0.6%	132	463	0.000281
x86*2	128	TTbar 10k	10'000	05:44:25	0.4%	2.5843	0.6%	133	463	0.000258
arm	80	TTbar 1k	1'000	00:41:23	3.4%	0.1749	3.2%	101	308	0.000175
arm	80	TTbar 10k	10'000	05:19:07	2.2%	1.5853	2.5%	110	309	0.000159