Few updates on Power Measurement

Dr. Emanuele Simili

Recent Updates

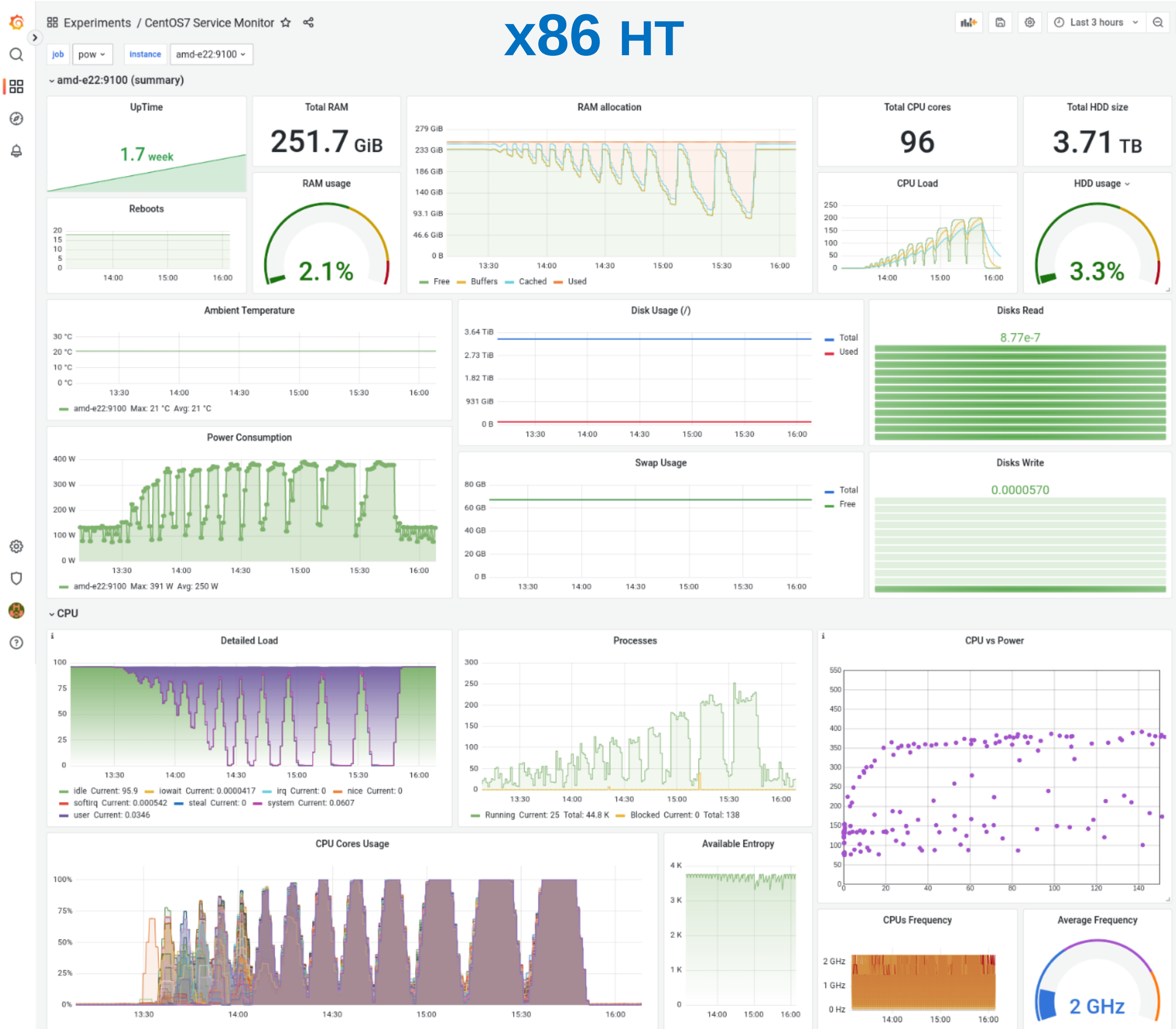
- Completed the thread-scan study on the **x86** (w/out HT) & **arm** using 8 workloads
- Tried a full unofficial run of the **HEP Benchmark Suite** (same 8 workloads)
- Re-done the IPMI validation measurement (IPMItool vs. metered plug)
- Status of the Energy Plug-in for the HEP-Score suite & open issues
- (no) Preparing paper for the ACAT 2022 conference (using old data)
- (no) Tested the latest HEP-Score with Gonzalo's script (run_HEPscore.sh)

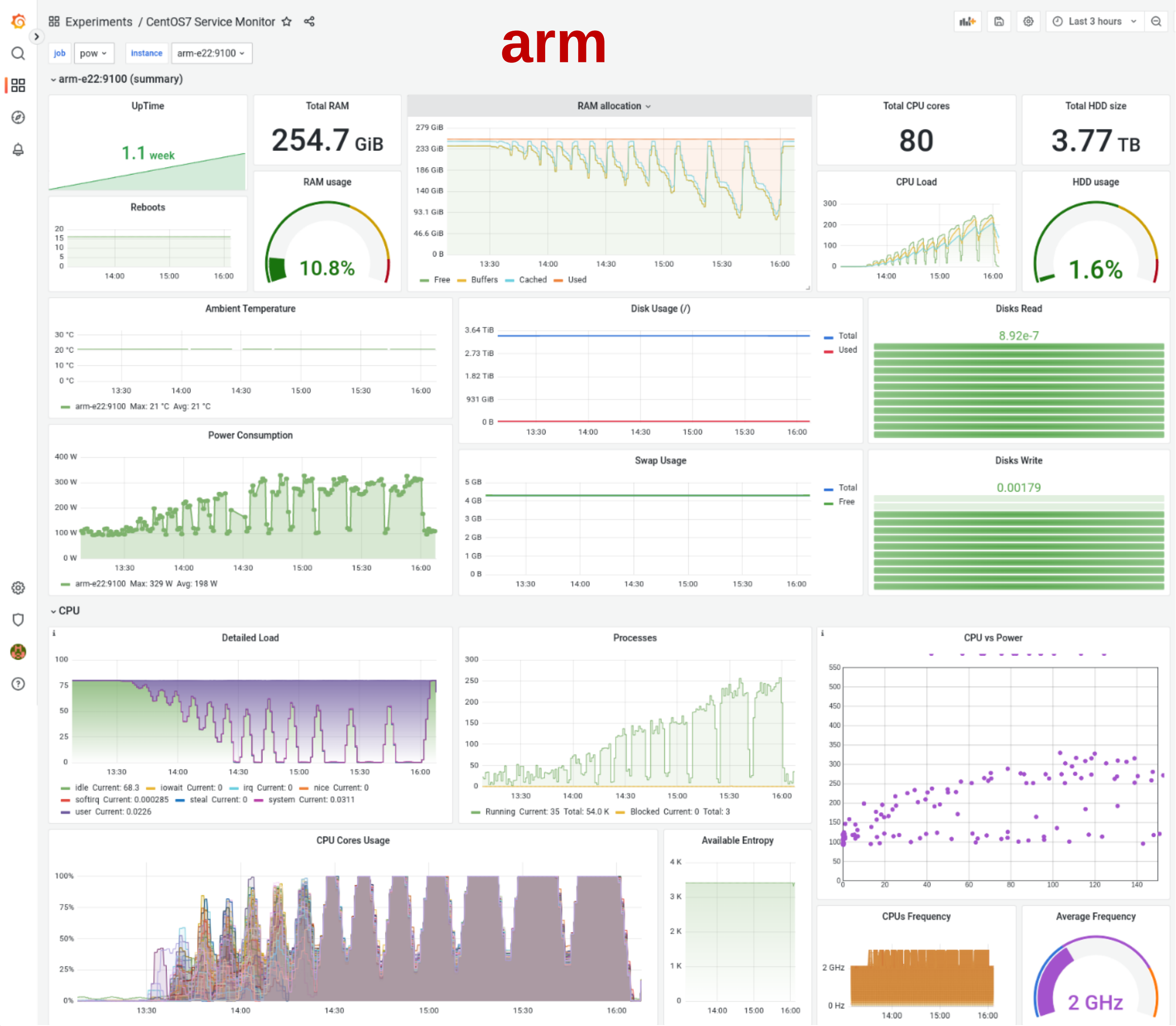
ScotGrid @ Glasgow: Emanuele Simili, Gordon Stewart, Samuel Skipsey, Dwayne Spiteri, David Britton

HEPiX @ CERN: Domenico Giordano, Gonzalo Menendez Borge, Johannes Elmsheuser, etc.

Job Profiles (LHCb)

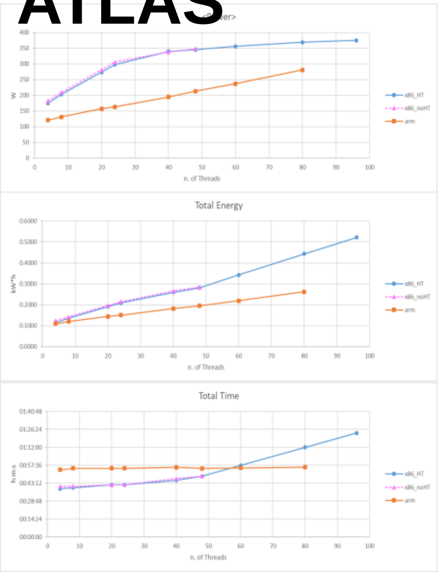
Here an example of runtime profiles for the LHCb workload (one of the 8 containers). The workload was executed ~ 10 times, increasing the number of copies at each run to progressively fill the CPU …



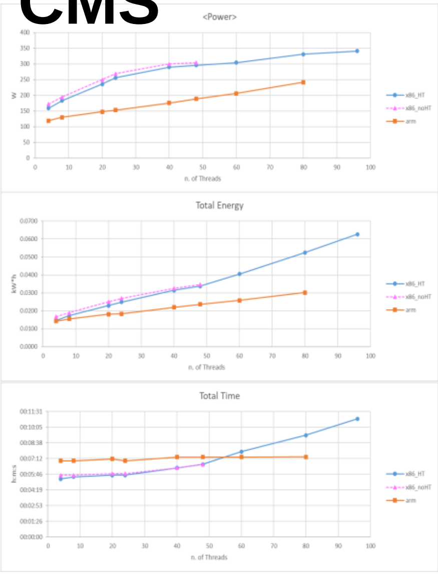


Thread Scan (8x)

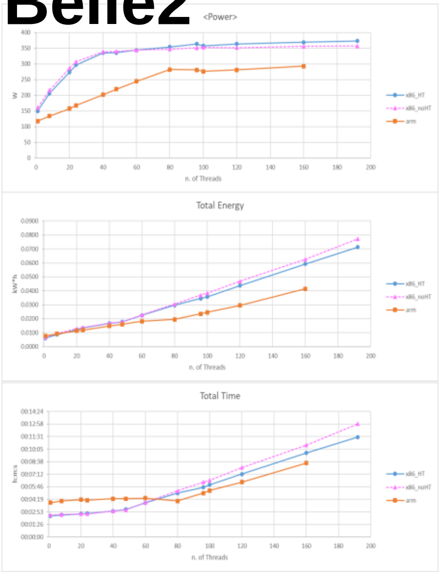
ATLAS



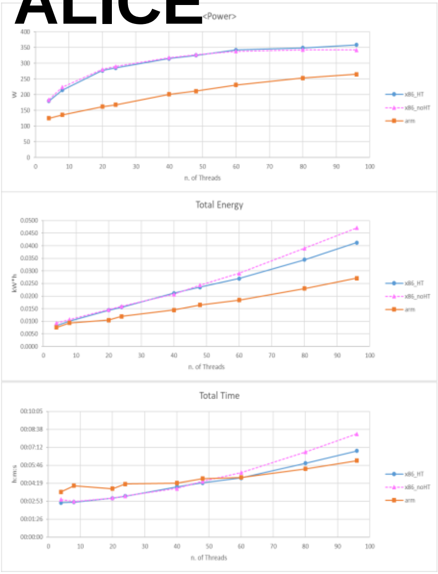
CMS



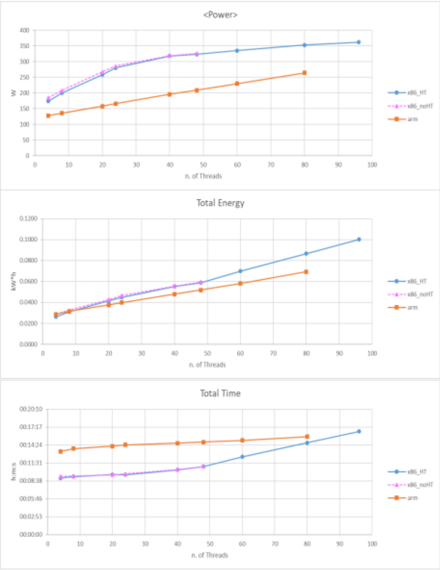
Belle2



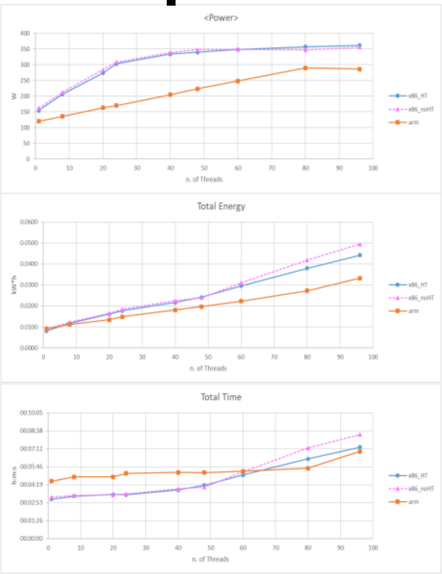
ALICE



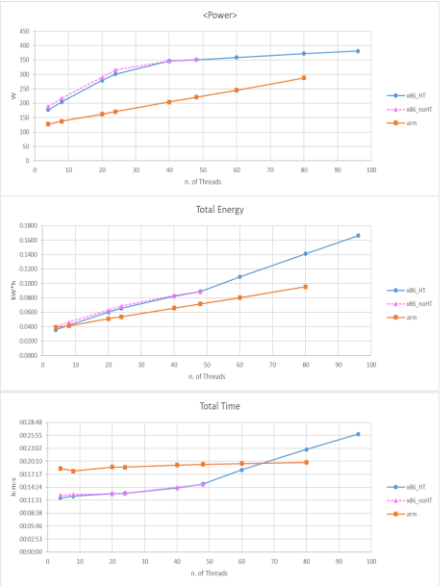
ATLAS reco



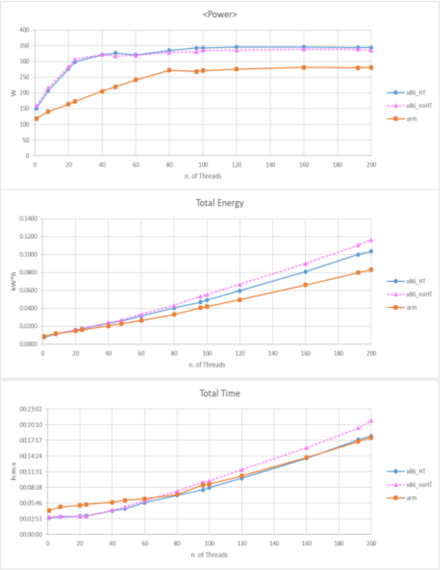
Sherpa



CMS reco



LHCb



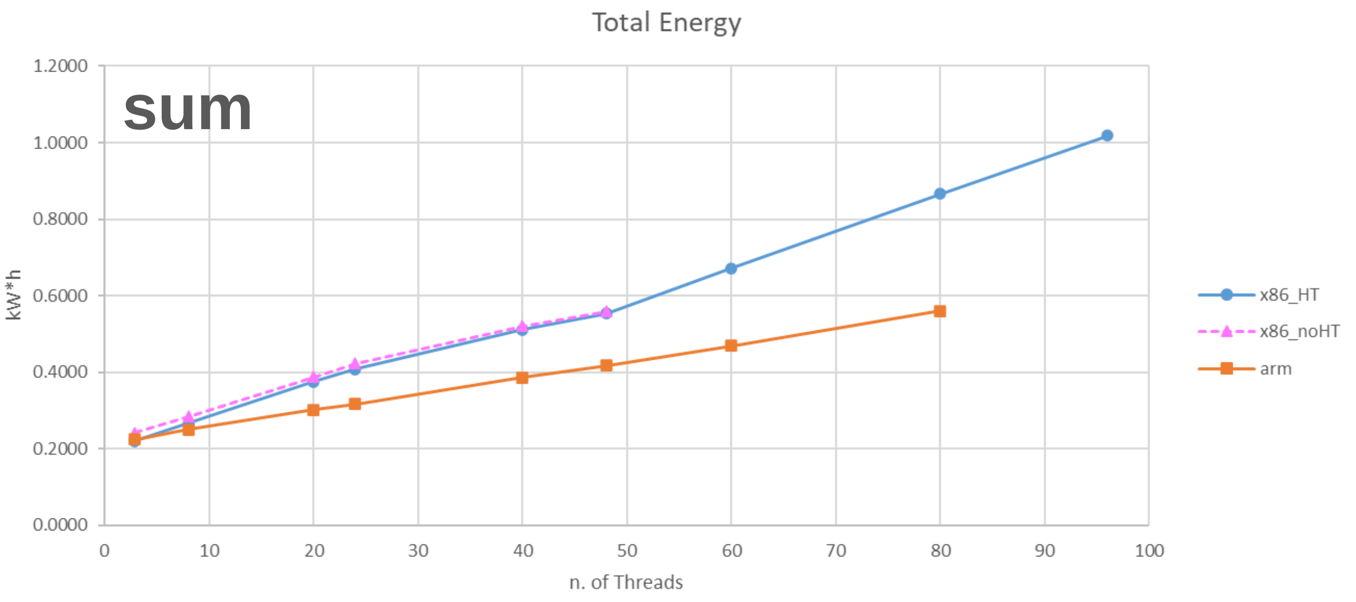
On **ARM**, the Energy (Power) increase linearly with the n. of threads, on **x86** saturates once hyper-threading starts.

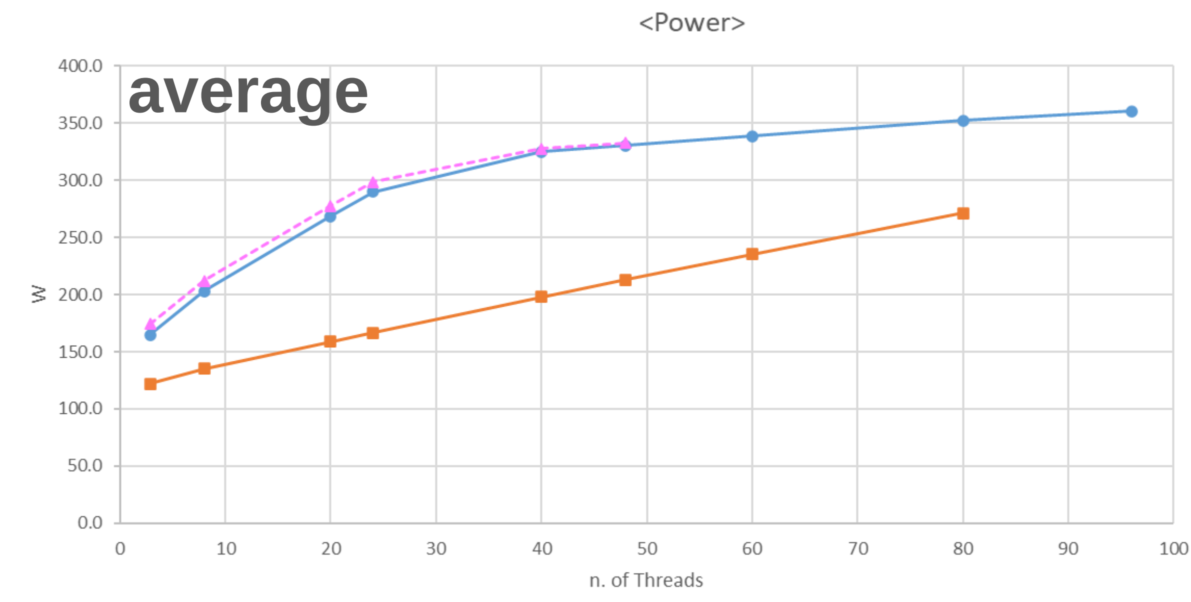
W.r.t. the n. of threads, the execution time is constant on **ARM**, while it increases on **x86** once hyper-threaded.

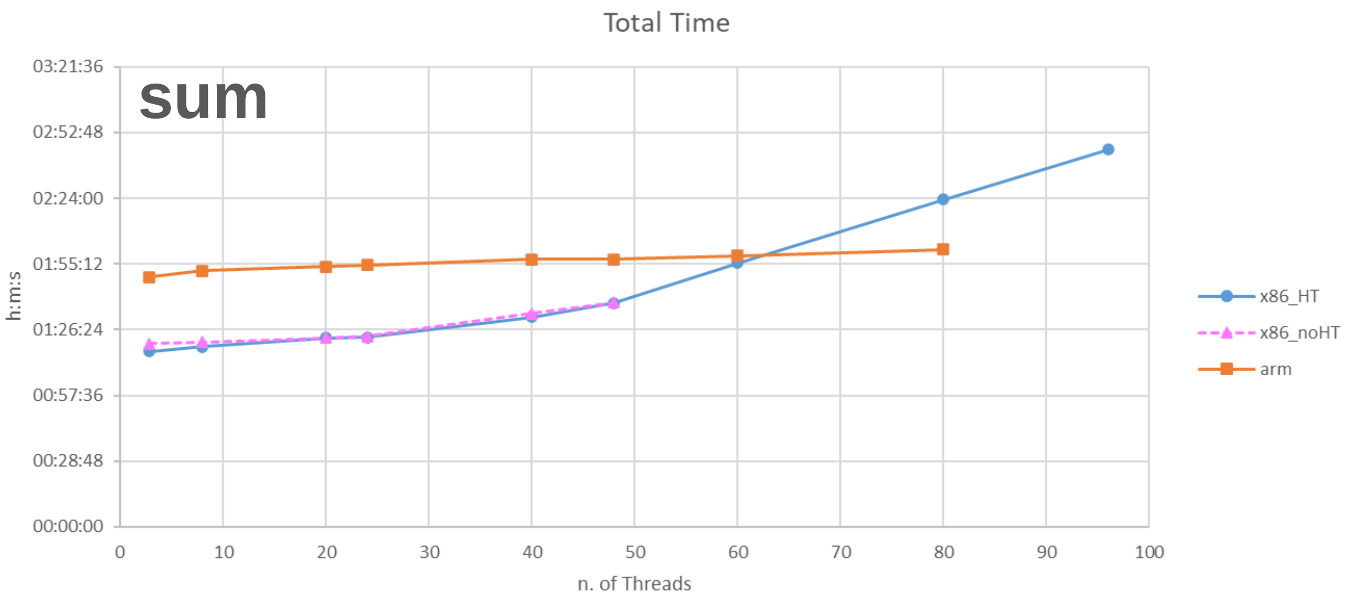
List of containers:

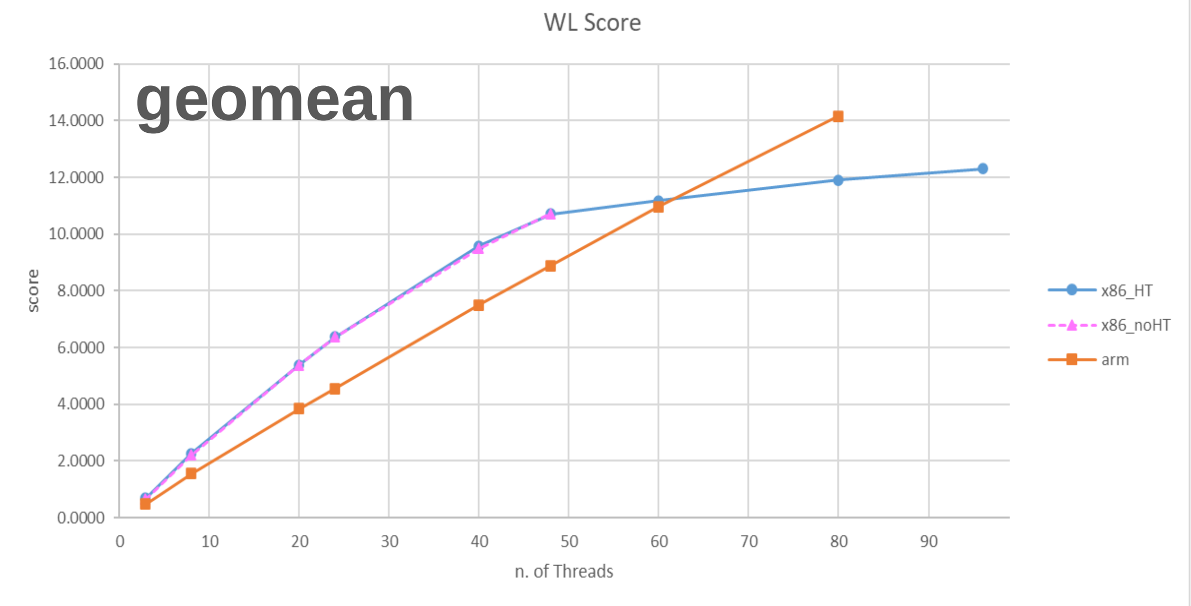
- atlas-sim_mt-ma-bmk
- cms-gen-sim-run3-ma-bmk
- belle2-gen-sim-reco-ma-bmk
- alice-digi-reco-core-run3-ma-bmk
- atlas-gen_sherpa-ma-bmk
- atlas-reco_mt-ma-bmk
- cms-reco-run3-ma-bmk
- lhcb-sim-run3-ma-bmk

Thread Scan (averages)



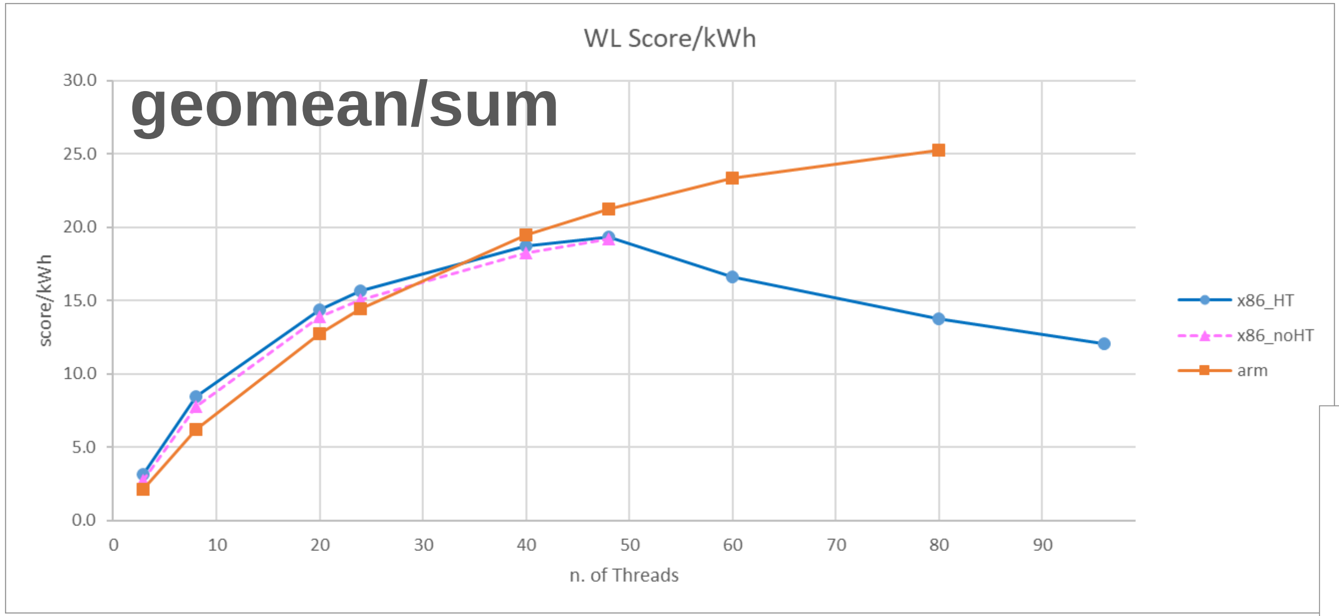






Thread Scan (scores)

Combining the data is not easy …and it is not clear to me what quantity will be more interesting:



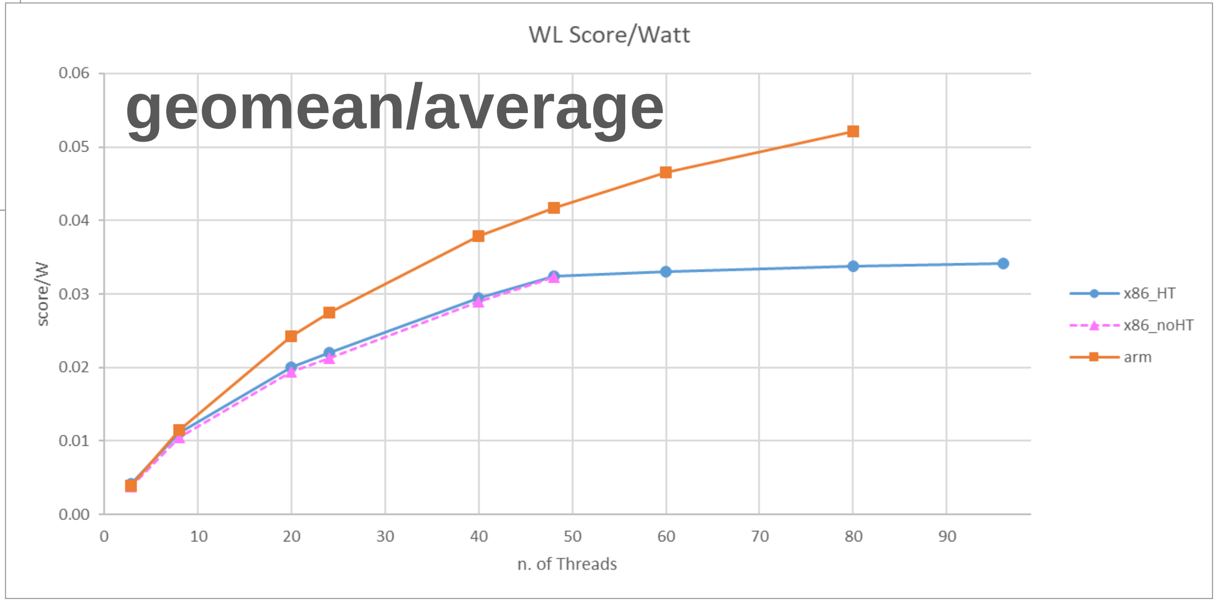
I tend to be more in favor of:

WL-Score (*normalized) / Total Energy

Because Tot. Energy takes into account the job duration as well, while <Power> does not

This is also more fair to the **x86**, which may use more energy but in some case completes the job quicker!

It is trivial that the <Power> of **arm** is lower, due to the lower TDP ...

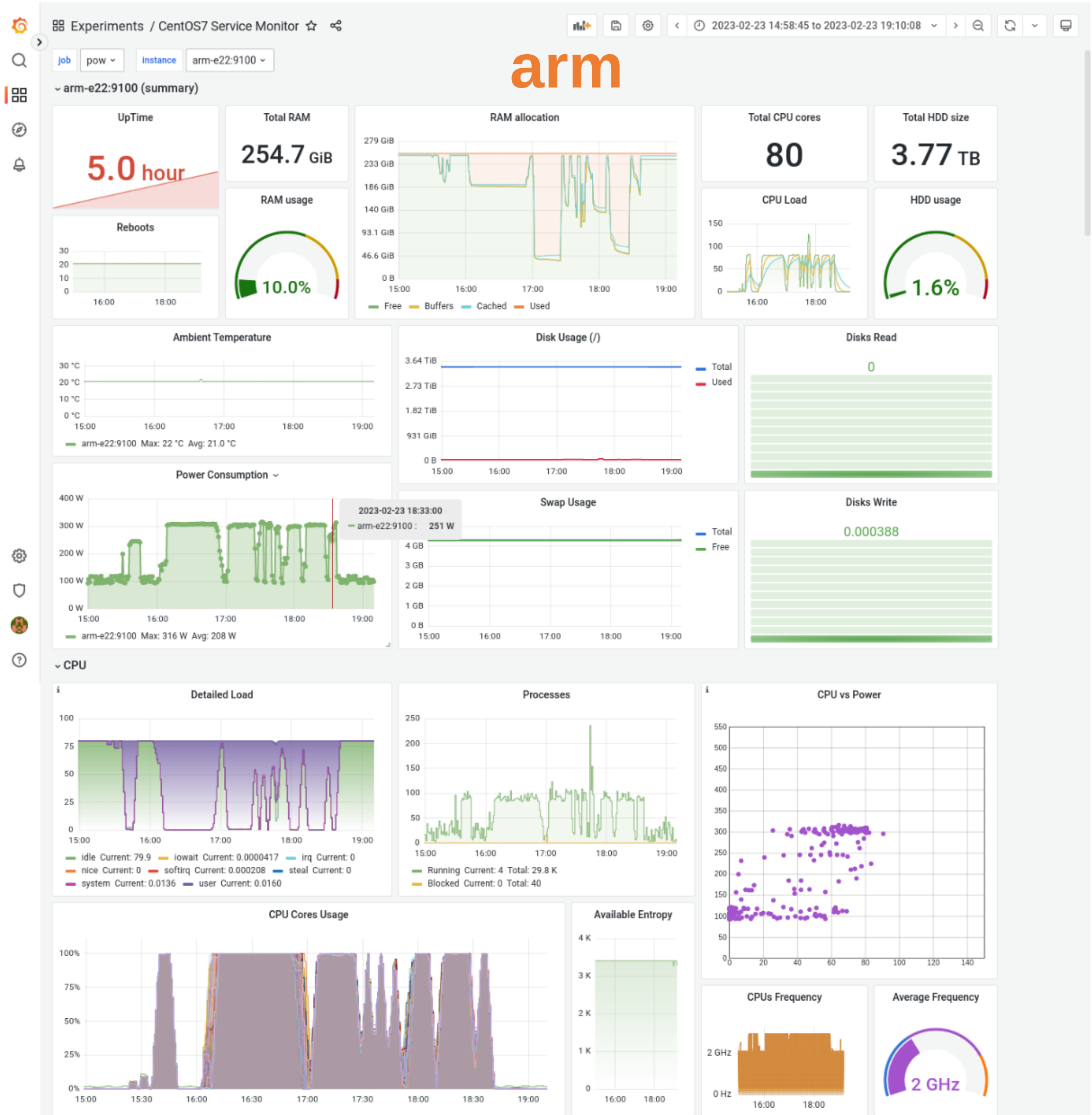


A physical **x86** cores is faster than an **ARM** core, but rapidly lose the advantage once hyper-threading starts.

ARM is always more energy efficient than **x86** !

HEP-Score 2023

Here is a full run of the most recent HEP-Score containers available for **arm** & **x86** (8 in total):



Power Profiles (runtime)

Runtime power profile extracted from the **arm** and the **x86** (with and without Hyper-Threading):

Full list of workloads:
(in order of execution)

WorkLoad	Container	cp	thr/cp	evt/thr	tot evts
ATLAS Sim	atlas-sim_mt-ma-bmk	24	4	20	1920
CMS Gen-Sim	cms-gen-sim-run3-ma-bmk	24	4	100	9600
Belle2 Gen-Sim-Reco	belle2-gen-sim-reco-ma-bmk	96	1	50	4800
ALICE Deigi-Reco	alice-digi-reco-core-run3-ma-bmk	24	4	3	288
ATLAS Gen Sherpa	atlas-gen_sherpa-ma-bmk	96	1	500	48000
ATLAS Reco	atlas-reco_mt-ma-bmk	24	4	100	9600
CMS Reco	cms-reco-run3-ma-bmk	24	4	100	9600
LHCb Sim	lhcb-sim-run3-ma-bmk	96	1	5	480

are these reasonable numbers?

HEP-Score 2023 (8x)

Beside having too many numbers to deal with, my major issue is with the WL-Scores, as they vary over 3-4 orders of magnitudes, making plots impossible without some sort of normalization !

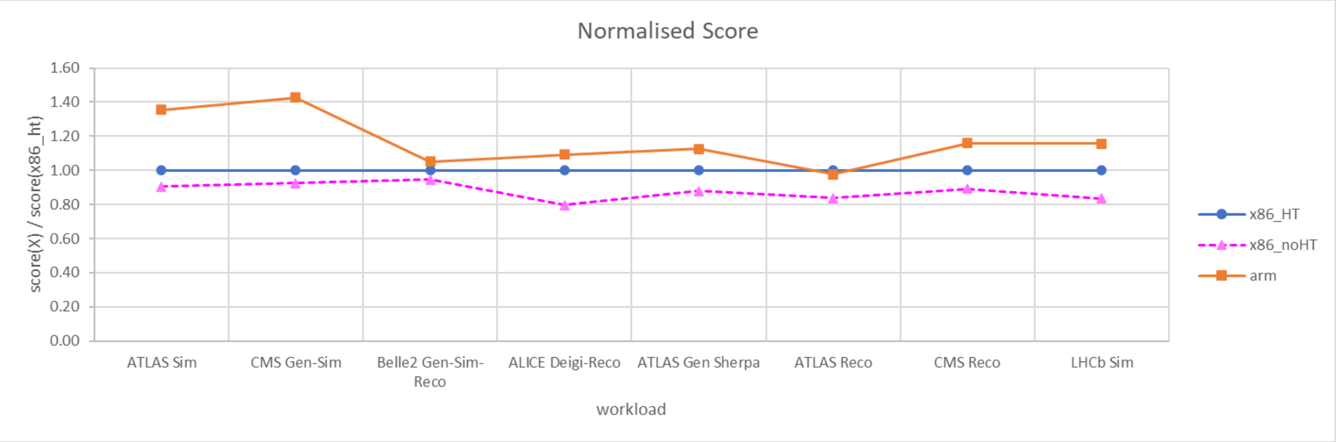
Arch	Max Threads	WorkLoad	cp	thr/cp	evt/thr	tot evts	Time (H:m:s)	Time (s)	Energy(kW*h)	Pow avg (W)	WL score	evt/sec	E/evt (Wh)	Time/cp	Energy/cp	score/x86_ht	(score/x86)/watt	(score/x86)/kWh
x86_HT	96	ATLAS Sim	24	4	20	1920	01:22:16	4936	0.5134	374	0.4081	0.3890	0.2674	205.7	0.0214	1.00	0.003	1.948
x86_HT	96	CMS Gen-Sim	24	4	100	9600	00:43:16	2596	0.2730	379	3.7328	3.6980	0.0284	108.2	0.0114	1.00	0.003	3.663
x86_HT	96	Belle2 Gen-Sim-Reco	96	1	50	4800	00:05:48	348	0.0350	362	20.2441	13.7931	0.0073	3.6	0.0004	1.00	0.003	28.571
x86_HT	96	ALICE Deigi-Reco	24	4	3	288	00:07:02	422	0.0407	348	0.7652	0.6825	0.1415	17.6	0.0017	1.00	0.003	24.546
x86_HT	96	ATLAS Gen Sherpa	96	1	500	48000	00:07:33	453	0.0448	356	113.3022	105.9603	0.0009	4.7	0.0005	1.00	0.003	22.302
x86_HT	96	ATLAS Reco	24	4	100	9600	00:16:34	994	0.0985	357	12.0765	9.6579	0.0103	41.4	0.0041	1.00	0.003	10.156
x86_HT	96	CMS Reco	24	4	100	9600	00:26:04	1564	0.1638	377	6.4225	6.1381	0.0171	65.2	0.0068	1.00	0.003	6.105
x86_HT	96	LHCb Sim	96	1	5	480	00:08:15	495	0.0469	341	2'801.8219	0.9697	0.0978	5.2	0.0005	1.00	0.003	21.304
x86_HT	96					0												
x86_noHT	48	ATLAS Sim	12	4	20	960	00:47:12	2832	0.2755	350	0.3696	0.3390	0.2869	236.0	0.0230	0.91	0.003	3.288
x86_noHT	48	CMS Gen-Sim	12	4	100	4800	00:23:35	1415	0.1400	356	3.4493	3.3922	0.0292	117.9	0.0117	0.92	0.003	6.601
x86_noHT	48	Belle2 Gen-Sim-Reco	48	1	50	2400	00:03:10	190	0.0179	338	19.1577	12.6316	0.0074	4.0	0.0004	0.95	0.003	53.016
x86_noHT	48	ALICE Deigi-Reco	12	4	3	144	00:04:34	274	0.0245	322	0.6100	0.5255	0.1702	22.8	0.0020	0.80	0.002	32.525
x86_noHT	48	ATLAS Gen Sherpa	48	1	500	24000	00:04:11	251	0.0238	342	99.5964	95.6175	0.0010	5.2	0.0005	0.88	0.003	36.872
x86_noHT	48	ATLAS Reco	12	4	100	4800	00:10:57	657	0.0588	322	10.1294	7.3059	0.0123	54.8	0.0049	0.84	0.003	14.255
x86_noHT	48	CMS Reco	12	4	100	4800	00:15:04	904	0.0880	350	5.7184	5.3097	0.0183	75.3	0.0073	0.89	0.003	10.121
x86_noHT	48	LHCb Sim	48	1	5	240	00:05:03	303	0.0264	314	2'339.6395	0.7921	0.1100	6.3	0.0006	0.84	0.003	31.630
x86_noHT	48																	
arm	80	ATLAS Sim	20	4	20	1600	00:55:06	3306	0.2581	281	0.5526	0.4840	0.1613	165.3	0.0129	1.35	0.005	5.246
arm	80	CMS Gen-Sim	20	4	100	8000	00:25:54	1554	0.1256	291	5.3221	5.1480	0.0157	77.7	0.0063	1.43	0.005	11.352
arm	80	Belle2 Gen-Sim-Reco	80	1	50	4000	00:04:42	282	0.0222	283	21.2810	14.1844	0.0055	3.5	0.0003	1.05	0.004	47.395
arm	80	ALICE Deigi-Reco	20	4	3	240	00:05:33	333	0.0233	251	0.8353	0.7207	0.0969	16.7	0.0012	1.09	0.004	46.931
arm	80	ATLAS Gen Sherpa	80	1	500	40000	00:05:42	342	0.0274	289	127.4113	116.9591	0.0007	4.3	0.0003	1.12	0.004	40.996
arm	80	ATLAS Reco	20	4	100	8000	00:15:50	950	0.0690	262	11.7988	8.4211	0.0086	47.5	0.0035	0.98	0.004	14.155
arm	80	CMS Reco	20	4	100	8000	00:19:53	1193	0.0951	287	7.4371	6.7058	0.0119	59.7	0.0048	1.16	0.004	12.183
arm	80	LHCb Sim	80	1	5	400	00:07:12	432	0.0326	271	3'242.8401	0.9259	0.0814	5.4	0.0004	1.16	0.004	35.547
arm	80																	

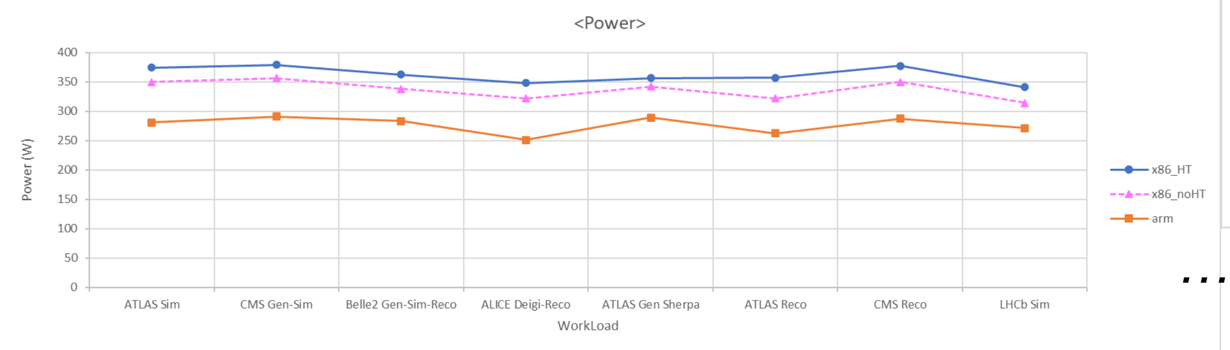
What’s a reasonable normalization? Would it be possible to normalize them within each container?

HEP-Score results

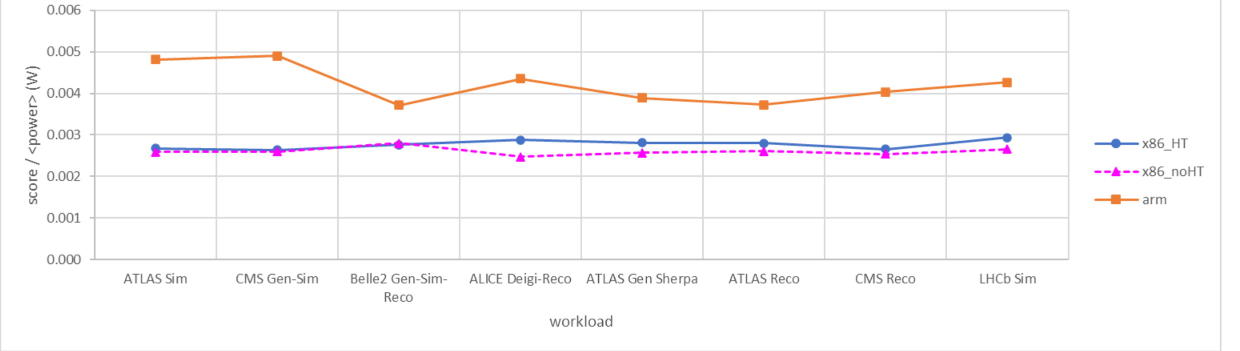
How the data look like for each workload …

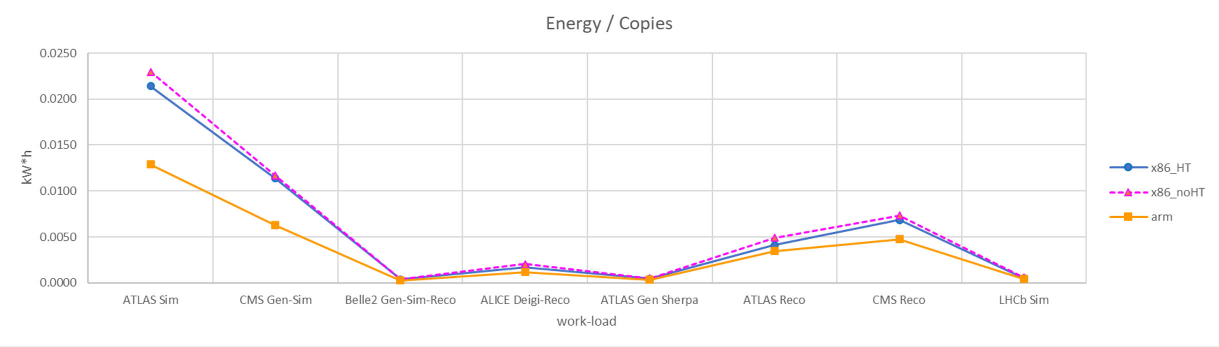
WL-Score normalized to the x86 value (=1)

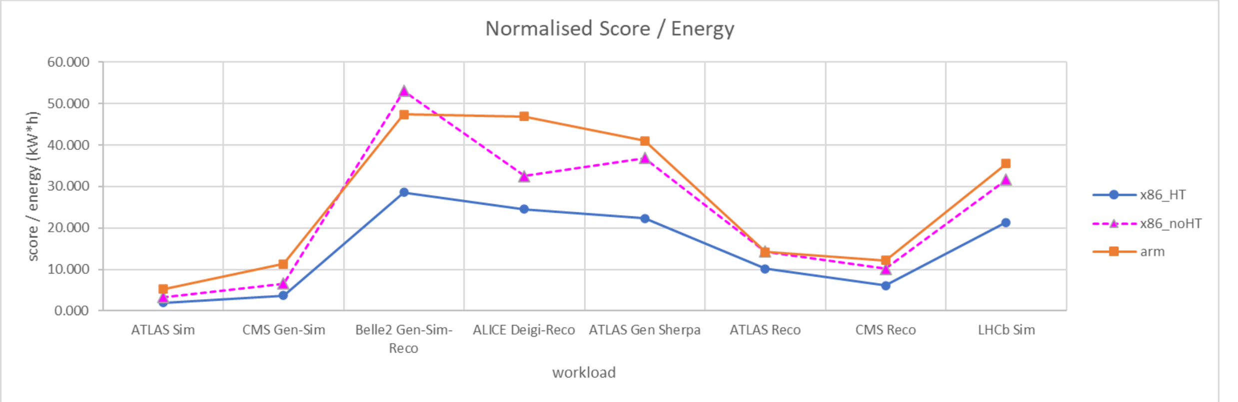




… and derived quantities:



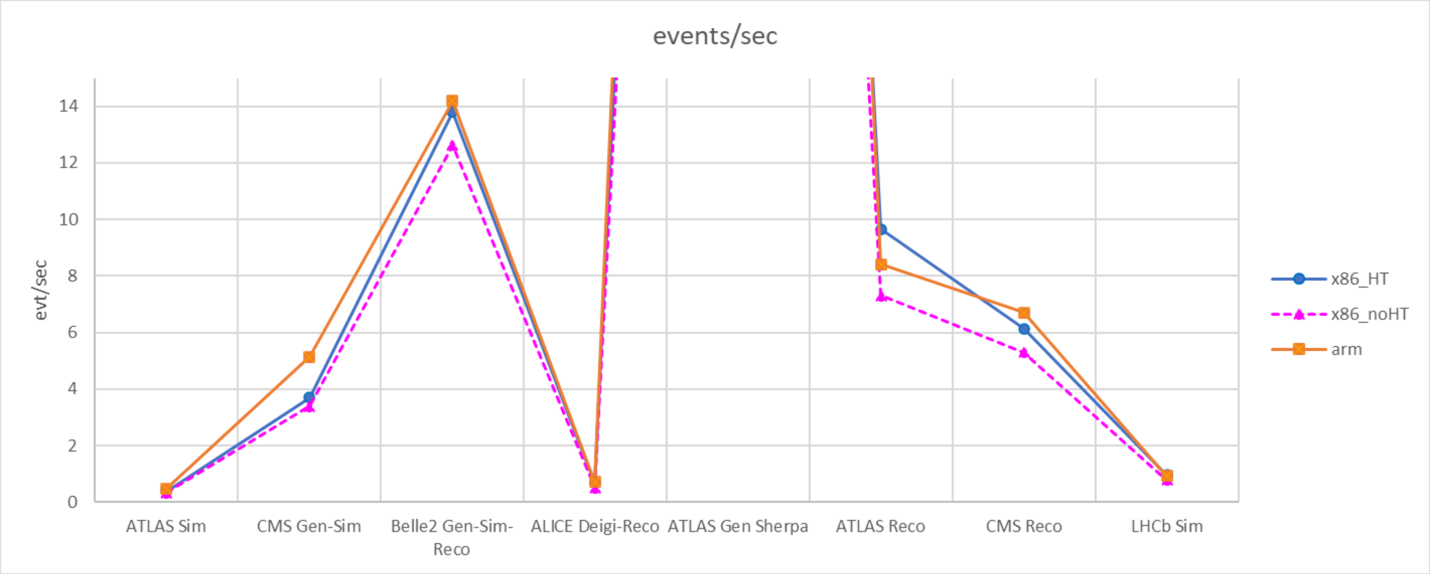


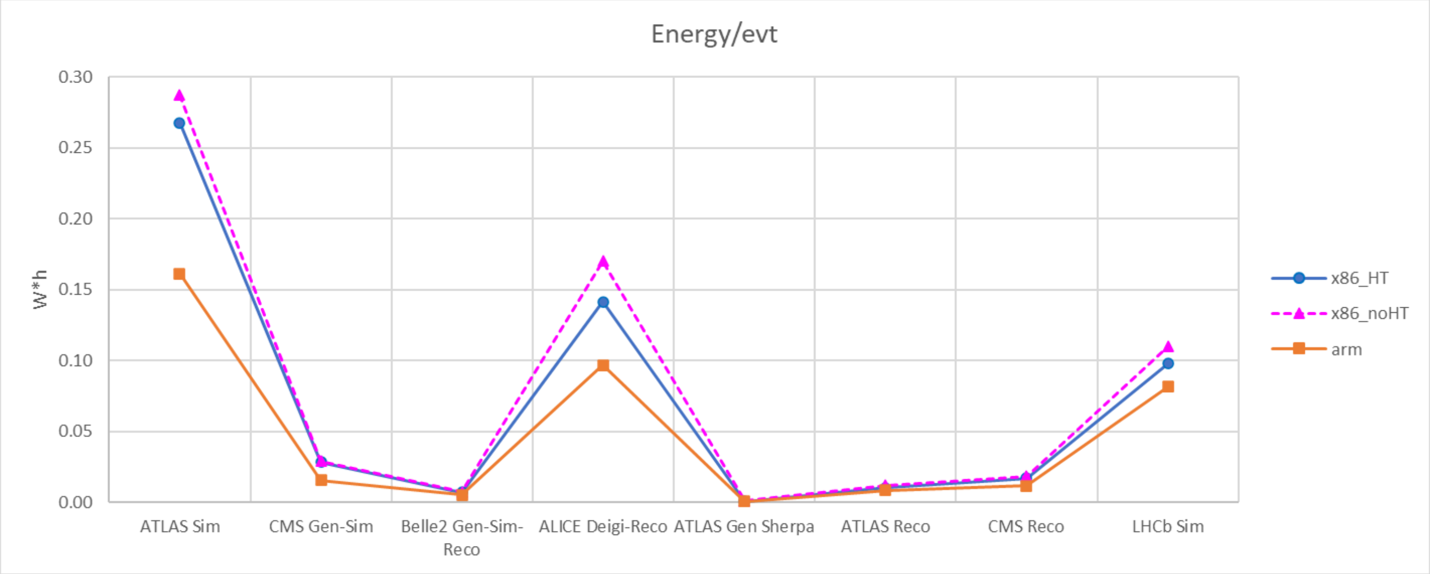


HEP-Score results (2)

Standard plots (such as Events/Second & Energy/Event) are not very nice to look at … due to the very different nature of the workloads and widely different number of events produced:

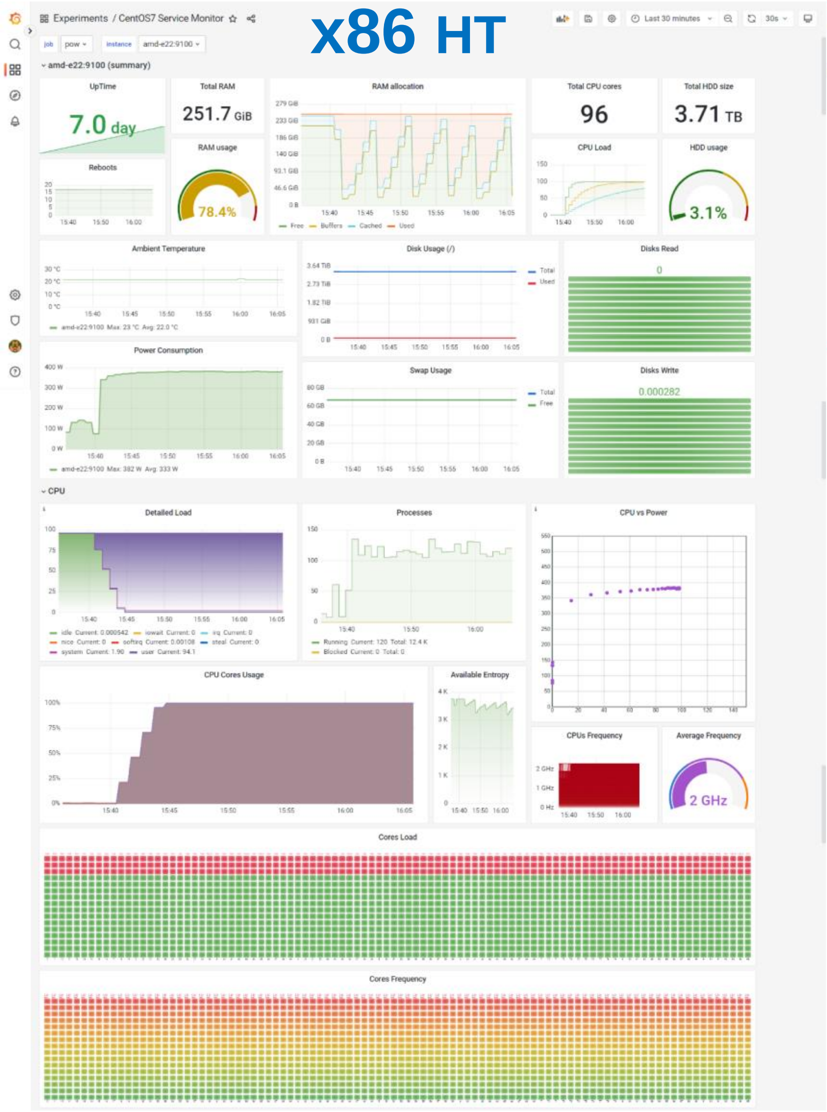
Again, it would be nice to have some sort of normalization, relative to the specific workload (e.g., event generation produces 1 event/sec. , full detector simulation produces 1 event/min. , etc.)

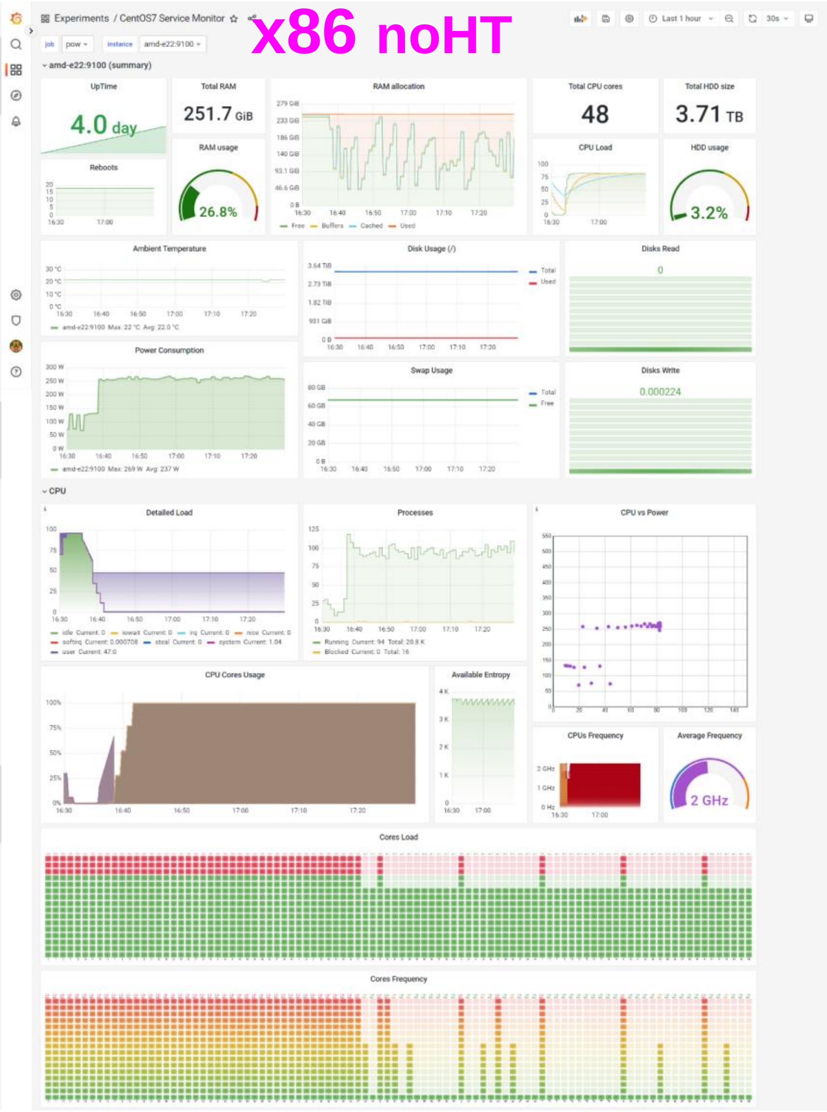


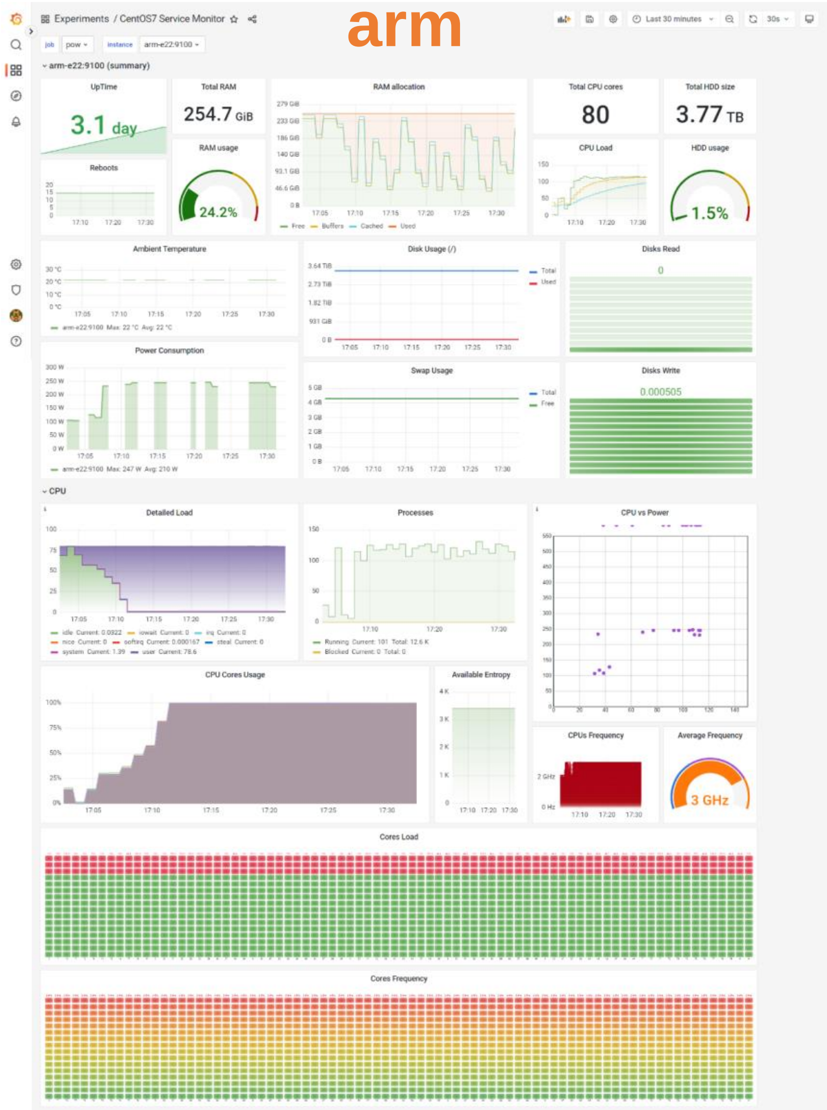


IPMI validation

I did a few runs to validate IPMI reading on **arm** & **x86**, by comparing the output of IPMItools with the reading from a metered plug connected to the servers (1h duration, sleep & stress):



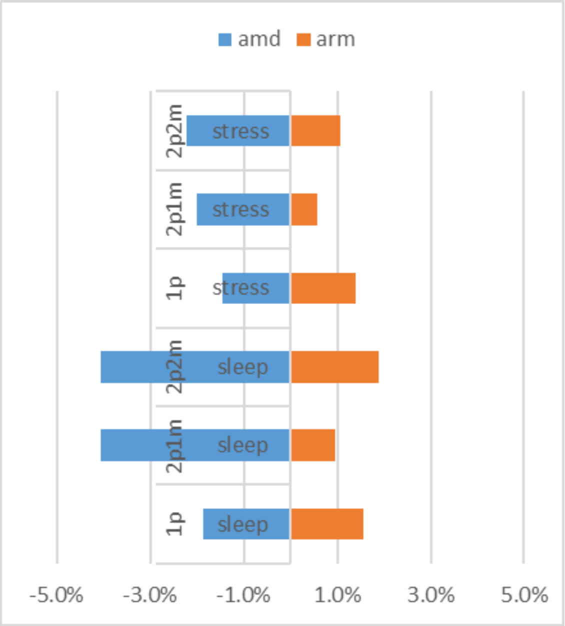




Validation Results

Results are a bit confusing, with the discrepancy changing sign between **arm** & **x86** …
However, the error is small enough, with the highest discrepancy being about 4%.

					IPMI exporter					Manual read (1h)			diff (IPMI - Plug)	
job	machine	HT	power(s)	meter(s)	Time (s)	Energy (kWh)	Min (W)	Max (W)	Avg (W)	Energy (kWh)	Min (W)	Max (W)	Energy	% diff/IPMI
sleep	amd	HT	1	1	3601	0.086	60	148	86	0.088	82	135	-0.002	-1.9%
sleep	amd	HT	2	1	3601	0.093	72	150	93	0.097	87	156	-0.004	-4.1%
sleep	amd	HT	2	2	3600	0.092	72	163	92	0.096	89	154	-0.004	-4.1%
stress	amd	HT	1	1	3600	0.291	68	296	291	0.295	267	300	-0.004	-1.5%
stress	amd	HT	2	1	3630	0.379	76	446	376	0.387	334	396	-0.008	-2.0%
stress	amd	HT	2	2	3600	0.372	77	377	372	0.380	334	388	-0.008	-2.2%
sleep	amd	noHT	1	1	3601	0.134	116	151	134	0.135	133	143	-0.001	-0.7%
stress	amd	noHT	1	1	3600	0.259	131	269	259	0.263	241	272	-0.004	-1.4%
sleep	arm	//	1	1	3600	0.095	93	114	95	0.094	94	109	0.001	1.6%
sleep	arm	//	2	1	3599	0.104	87	131	104	0.103	102	116	0.001	1.0%
sleep	arm	//	2	2	3599	0.104	52	123	104	0.102	102	118	0.002	1.9%
stress	arm	//	1	1	3599	0.236	94	244	236	0.233	197	240	0.003	1.4%
stress	arm	//	2	1	3600	0.241	92	248	241	0.240	105	246	0.001	0.6%
stress	arm	//	2	2	3600	0.239	93	247	239	0.236	207	246	0.003	1.1%



There is already some idea about fitting these data separately for the 2 servers, involving a slope (efficiency of the supplier) and an intercept (power lost) …

… but I need to collect more data, as an interpolation over 2 points is not ideal !

Energy Plug-In

The **energy_plugin** class implements an internal timer, which is used to regularly grab runtime metrics during execution, and an analyser which calculates execution stats from runtime metrics.

Python code uploaded on Git:
(but Jira issue not closed yet …)

HEP dependencies:
- StatefulPlugin **ok**
- Extractor **no**

Called at regular intervals to grab metrics, such as time-stamp, IPMI power reading, CPU usage, etc.

Calculates statistics using the 'trapezoidal sum', which takes care of possibly changing intervals or missing time-stamps.

```
# Skeleton class by Gonzalo, content by Emanuele (v0.4c)

import json
...
from hepbenchmarksuite.plugins.stateful_plugin import StatefulPlugin
#from hepbenchmarksuite.plugins.extractor import Extractor

class EnergyPlugin(StatefulPlugin):
    ...

    # IPMI loop function (runtime): dumps system metrics to a dictionary
    def grab_metrics(self,start_time):
        time_stamp = dt.datetime.utcnow()
        time_key = str(time_stamp.isoformat(timespec='milliseconds')) + "Z"
        ...
        cmd_ipmi = r""" ipmitool dcmi power reading | grep "Instantaneous power reading:" """
        powa = self.get_numbers(self.run_command(cmd_ipmi),0)
        ...

    # IPMI analiser functions (postrun): calculates statistics and averages
    def calculate_statistics(self, measurements: dict) -> Dict[str, float]:
        ...
        for k in sorted(measurements.keys()):
            ...
            deltaSec = (time_stamp - time_prev).total_seconds()
            powAve = (powa + powa0) / 2
            ...
        return self.summary_dict
```

https://gitlab.cern.ch/hep-benchmarks/hep-benchmark-suite/-/blob/fc84702f7e90a1ce850abdd461fe6aaf9e374a1c/hepbenchmarksuite/plugins/energy_plugin.py

Open Issues

The **energy plug-in** is a Python module that runs alongside with the workloads, while extracting CPU and RAM usage, core Frequency, and IPMI (and GPU) power.
When the workload is finished, the plug-in calculates a number of execution statistics and save these, together with the time-stamped runtime data, as a dictionary in a *json* file.

There are a number of issues that became apparent during the first round of implementation and testing, some of these are solved (or so I think), others are still in the air:

❖ Sampling frequency: it does not matter any more!
Using the trapezoidal sum, I see that results with 1 , 5 and 10 sec. sampling frequency are equivalent (< 1%). Finally, I am opting for 5 sec. sampling interval (as 1 sec. is too often and clogs the collector).

❖ No idle collection: still unclear how we will achieve this during the actual run. (#)

❖ I don’t want to run the HEP-Score as **root** !
Therefore I will implement the option of grabbing IPMI values (only) from dump file. This means that the machine will execute a script at boot, and never think about it again!

❖ Energy normalization issue: if the energy is measured alongside the workload, how do we scale it?
The machine with more threads does more work, therefore the total Energy will be more!

❖ Score normalization issue: the score produced by each workload varies over 3-4 orders of magnitude!
Without a proper normalization, it is impossible to plot anything…

End

Available Hardware

We have two almost identical machines of comparable price, one with an AMD **x86_64** CPU (48c/96t), the other with an Ampere **arm64** CPU (80c):

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 80C @ 3GHz (TDP 210W)
RAM:	256GB (16 x 16GB) DDR4 3200MHz
HDD:	3.84TB Samsung PM9A3 M.2 (2280)

The **x86_64** CPU can run in Hyper-Threading regime (with 96 hyperthreaded cores), or without (with 48 physical cores). Hyper-threading does not double performances, but adds 10-20%. Roughly: 1 hyperthreaded core ~ 55-60% of 1 physical core.

The **arm64** CPU has no such feature, therefore it can only run with its 80 physical cores.