**Goal of the paper**: Build a scheduling algorithm to improve energy efficiency on a per request/per thread basis running cloud services (elasticsearch, Cassandra, or Memcached) on heterogeneous multicore systems.

Front-end experiments (Motivation and Opportunities):

**Experiment 1**: Research Question: (1) do we have in the first place light or heavy queries: show CDF? (2) does it make sense to use het cores to map light vs heavy threads on little vs big cores?

ElasticSearch on Juno board experiments (Turn off core sleep states)

1. see ICAC 2013: MSR paper
2. Per request (**per hottest function)** energy and latency measurements on big vs little cores.
   1. Run a single request for a fixed key length on big core at fixed frequency (1.15GHz) \* 10000 times.
      1. Only short size queries (1-4 keyword)
      2. Only large size queries. (12-16 keyword)
   2. Run a single request for a fixed key length on a small core. \* 10000 times.

(something similar to this is what we are expecting: <https://www.overleaf.com/read/jqqqgzxqdnzd>)

Run an app for 10min - poll every 1 sec - we get a energy value X (average)

Run the same app for 10min - poll every 10ms (play with this num) - we get a energy value Y = (Y-X)/X that’s the overhead.

(stupid question: why not build a model to predict power and then we will do easy? Like lucky scheduling)

(Because I have no clue of how to do this) :-D vinicius will do that

TODO:

1. Instrument the code on entry and on exit to read the energy value
2. Change the load gen in python to parameterize the script to issue: (A) short queries (1-4 KW) vs (B) long queries (12-16 KW)
3. Get the latency data from Elasticsearch: “took” time (overall service time) and get latency from the function entry/exit events

Implementation issues:

1. How to measure energy on Juno?

<https://community.arm.com/dev-platforms/w/docs/241/energy-monitoring-on-juno>

for example `/hwmon0/power3\_input' reads `8675' i.e. 8675 uJ and this corresponds to the instantaneous power consumption of the Cortex-A53 cluster

Code to read energy: <https://github.com/ARM-software/devlib/blob/master/src/readenergy/readenergy.c>

2) Turn off sleep states (ONLY IF USES PERF)

Code for deactivating (echo 0 for activating)

for cpu in 0 1 2 3 4 5; do

for state in 0 1 2 ; do

echo 1 > /sys/devices/system/cpu/cpu$cpu/cpuidle/state$state/disable;

done;

done

On the ARM Juno platform, there is a known bug that causes perf to generate garbage values for all cores whenever any core enters an idle state. Since perfor- mance statistics are only needed for the HipsterCo vari- ant, we overcome this by disabling CPUidle [40, 41]. This prevents Linux from entering the cores in an idle state when changes in the mapping cause idle periods longer than 3500 us.

**Experiment** 2: other workloads: Cassandra and Memcached

<https://github.com/yqzhang/ClearSuite>

Cassandra: <https://github.com/yqzhang/ClearSuite/tree/master/Data-Serving>

Memcached: <https://github.com/yqzhang/ClearSuite/tree/master/Data-Caching> (c++ :-)

**[If time permits]** Media Streaming: <https://github.com/yqzhang/ClearSuite/tree/master/Media-Streaming-Nginx>

TailBench: mostly C/C++ apps:

<http://tailbench.csail.mit.edu/>

Back-end experiments:

Just a question: what about a summary paper?

Experiment 3: show the effectiveness of the approach

How it works?

1. Profile and extract hot functions
   1. What tools? And why? Preferrably open-source or well cited.

B) Instrument the code to migrate when you find a hot function: on entry

* 1. <https://github.com/OpenHFT/Java-Thread-Affinity/>
  2. <http://chronicle.software/products/thread-affinity/>

1. Get the call-tree : grandparents to baby.

Experiment 4: how well does it work?

* Metrics etc
* Comparison with State-of-the-art
  + Hipster: we must compare
  + Micro paper: maybe: Adaptative Slow to Fast ??
* Heracles: /home/rnishtala/shared/HERACLES
  + Problems: intel only actuations
* Rubik (hard motherfucker!!!) - can ask nathan
* Last 4 years papers for related work.