EASY-Assignment #3: Energy-Related Optimisation

The topic of this assignment is energy optimisation at software and hardware level. Specifically, we evaluate and optimise the influence of the compiler and the hardware configuration on the energy demand.

Goals of this assignment

- Energy-related optimisation of software
- Energy-related optimisation of the hardware configuration

3.1 Compiler Optimisation

Modern compilers provide a variety of different optimisations, mostly focusing on performance or code size. However, these optimisations also influence the power and energy demand of a program. Goal of this assignment is to analyse the impact of different compilers and optimisations on the power and energy demand.

3.1.1 Compiler Flags

Compile the `julia` program with the following optimisation levels and measure the total energy demand of the resulting binary for one run with the default `julia` parameters and `stdout` redirected to `/dev/null`. The hardware configuration should be comparable for all measurements. Additionally, measure the power demand during execution.

Optimisations flags: `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Ofast`

What differences in energy and power demand do you see? What happens when different compiler implementations are used (`gcc` vs. `clang`).

Visualise your results.

3.1.2 Optimisation Identification

Each optimisation level of a compiler enables a whole set of optimisations passes[1]. The optimisation passes enabled for a specific optimisation level of the `gcc` compiler can be determined with

```
gcc -Q -O<opt_level> --help=optimizers
```

Evaluate the individual influence on the energy demand (you do not need to compare power values) of optimisation passes added at optimisation level `O2` compared to optimisation level `O1` of the `gcc` compiler.

Which optimisation pass is the most important one and for what reason? Do the individual energy differences of the optimisation passes sum up to the energy difference between optimisation level `O1` and `O2` as discovered in assignment 3.1.1?

[1] `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

3.2 CPU Frequency Scaling

You have discussed the PAST algorithm[2] in the lecture. The goal is to implement it for Linux, in user-space.

3.2.1 Implement the Algorithm

The kernel provides the relevant load statistics in the `/proc/stat` pseudo-file. Your algorithm should periodically read this file, derive the current processor utilisation, and compute the optimal processor speed.

You should load the `intel_pstate` driver which controls performance of modern Intel x86 processors in the Linux kernel. Be aware that this driver utilises *hardware pstates*: The hardware can regulate its performance internally, based on information that is not available to the OS. The driver therefore provides an interface in the `sysfs`[3] to configure the maximum and minimum processor speed.

3.2.2 Visualise the Algorithm

Your program should create a log-file where it records the current load and the performance decisions it makes. You should also write a script that visualises this log file.

3.2.3 Test the Algorithm

Write a small load generator to test the governor. Does the frequency scaling algorithm improve the energy demand of the system?

Notes

- Material: the `julia` program
- Deadline: 2019-02-04 12:00

[2] Mark Weiser et al.: *Scheduling for Reduced CPU Energy.* OSDI'94.
[3] `/sys/devices/system/cpu/intel_pstate/{min,max}_perf_pct`