Specialization in Software Engineering

MEI, Universidade do Minho, 2022/2023

João Saraiva

Exercises on Software Energy Monitoring and Analysis

1 Software Energy Measurement via RAPL

To measure energy consumption of a software system there are two alternatives: We can use an external electronic component to measure the energy consumed by an hardware component, that is to say an energy meter, or we may use frameworks provided by chip manufacturers that measure/estimate the energy consumption of their chips.

Hardware manufacturers such as Intel and Qualcomm provide simple, easy to use, software frameworks to estimate the energy consumption of their modern CPUs.

In order to provide support to measure energy consumption of its CPUs, Intel introduced the Running Average Power Limiting (RAPL) interface in its Sandy Bridge architecture, which has been supported in following architectures. RAPL provides two different functionalities. First, it provides energy consumption estimation/measurements at very fine-grained and a high sampling rate. Energy readings are updated roughly every millisecond (1kHz). Energy is estimated using various hardware performance counters, temperature, leakage models and I/O models. Second, it allows capping/limiting the average power consumption of different components inside the processor, which also limits the thermal output of the processor.

The RAPL supports multiple power domains that are physically meaningful for energy management, such as processors, GPU, RAM, etc. Each power domain reports the energy consumption of the domain. Moreover, it allows programmers to limit the power consumption of that domain over a specified time window. RAPL supports the following energy estimation domains:

- *Core:* the energy consumed by all cores and caches.
- *GPU:* the energy consumed by the GPU.
- *DRAM:* this domain estimates the energy consumption of the random access memory (RAM).

- *Package:* It measures the energy consumption of the entire socket. It includes the consumption of all the cores, integrated graphics and also the uncore components (last level caches, memory controller).

It should be noted that not all intel architectures provide energy estimations to all RAPL domains. However, all intel architectures provide the estimation of the overall package consumption.

RAPL not only provides fine-grained and high sampling rate energy estimates, it also gives accurate estimates as shown in several recent research works [HDVH12, KHN+18].

There are frameworks to access the intel's RAPL interface in several programming languages. Next we briefly describe the C interface that we adapt to be re-used in a modular way, so that we can easily instrument fragments/components of C programs and measure their energy consumption[1].

2 RAPL interface in C

The C programming language has an interface to RAPL that started been developed at Intel, which was being updated by Vince Weaver until 2015. This interface is available as a single C program at

https://web.eece.maine.edu/~vweaver/projects/rapl/rapl-read.c

At University of Minho we have updated this interface in two ways: First, we are continuously updating it with intel's new architectures. Second, we have refactored its source code in order to provide a modular tool in C for energy consumption measurement.

Thus, we have defined a module, name `rapl.c` and `rapl.h`, that includes a initialization of the RAPL interface, function with name `rapl_init`, and two functions `rapl_before` and `rapl_after` which can be used to instrument C code fragments or calls to functions.

Next, we show an example of a C program that computes the factorial of a number given as argument of the `main` function. The fragment of the code where the factorial is computed, the for loop, is instrumented with the `rapl_before` and `rapl_after` functions. Before that, we have to initialize RAPL. We have also to include file `rapl.h` where the type signatures of those functions are defined to have access to them.

```
#include <stdio.h>
#include <stdlib.h>
#include "rapl.h"

int main (int argc, char **argv)
{ long fact = 1;
  int n = atoi (argv[1]);
  rapl_init(0);
```

[1]RAPL interfaces for Java and Python are available as the *jRAPL* and *pyRAPL* libraries at https://github.com/kliu20/jRAPL and https://pypi.org/project/pyRAPL/, respectively

```
    rapl_before(stdout,0);
    for(int x=1;x<=n;x++)
        fact=fact*x;
    rapl_after(stdout,0);
    printf("\nfact(%d): %ld \n",n,fact);
    return 0;
}
```

Assuming that this code is included in file with name `factorial.c`, we compile it together with `rapl.c` as follows:

```
gcc -O2 rapl.c factorial.c -o factorial -lm
```

To execute this program while measuring its energy consumption, first we need to have access to Intel's Model Specific Registers (MSP)[2] where the estimated energy is stored by the RAPL interface.

```
sudo modprobe msr
```

In order to access Intel's registers we need to execute it in superuser mode.

```
sudo ./factorial 33
```

2.1 RAPL's Language Ranking Framework

The modular implementation of the C interface of RAPL was the building block to define an energy ranking of programming languages [PCR+17] where the energy, runtime, and memory consumption of 10 programs implemented in 28 programming languages were monitored and analysed.

In order to be language agnostic and to not have to instrument with RAPL calls each of the 280 different programs, we implemented a C program that relies on a system call to execute a given program, given its name. Thus, the such C main function receives as command line argument the name (and path) of the program. It also gets as an additional argument the number of times the program will execute. It is extremely useful to be able to execute the program several times so that we have more accurate measurements, for instance by removing outlier results. Next we present a fragment of such C main function that contains the for loop that executes *ntimes*, each of the times it uses the *system* call to run the given program.

```
  for (i = 0 ; i < ntimes ; i++)
    {   sleep(1);
        fprintf(fp,"%s , ",argv[1]);
        rapl_before(fp,core);

#ifdef RUNTIME
        begin = clock();
```

[2]MSR are control registers that are part of Intel x86 instruction. They are used to monitor program execution, measure computer performance, and configure CPU features.

```
        gettimeofday(&tvb, 0);
#endif

        system(command);

#ifdef RUNTIME
        end = clock();
        gettimeofday(&tva, 0);
        //      time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
        time_spent = (tva.tv_sec-tvb.tv_sec)*1000000 + tva.tv_usec-tvb.tv_u
#endif

        rapl_after(fp,core);

#ifdef RUNTIME
        fprintf(fp," %G \n",time_spent);
#endif
    }
```

As we can see in this fragment, the system call is surrounded by calls both to RAPL and execution time functions. Thus, it measures energy and time consumption of every execution of the given program. The for loop starts with a sleep of 1 second to "guarantee" that the temperature of the chip does not introduce noise in RAPL measurements.

This function writes every measurement in a file (file pointer fp) in a csv format. Each line corresponds to one (of the *ntimes*) program execution where each item corresponds to an estimated energy domain or measured execution time. This file can be easily imported and manipulated by other tools, such as spreadsheet systems.

3 Exercises

Exercise 1: Consider the two provided implementations in C of the Fibonacci function.

1. Compile and execute the two functions within the framework of the ranking of programming languages. Answer the following questions:

   (a) Which is the difference in the *package energy consumption* reported by RAPL between the lowest and highest energy consumption between executions?

   (b) Which is the difference in runtime and energy consumption between both implementations? How did you compute it?

2. Instrument both C implementations with calls to RAPL library functions to directly compute the energy consumed when they are executed. Answer the following questions:

   (a) Which is the difference in terms of runtime and energy consumption reported by the language ranking framework and the direct instrumentation of the C

code? Is the overhead induced by using system calls in the ranking framework relevant?

Exercise 2: Consider that we wish to monitor and analyse the performance of sorting algorithms. Moreover, we would like to compare similar implementations of the same sorting algorithms in different programming languages.

1. Consider the implementation of two different sorting algorithms implemented in C and in the language your are more familiar with. In order to have such implementations you may consider to implement such implementation yourself, to find them in a open source repository, or to "ask" an AI source code generative model to produce them.

2. In order to have equivalent running programs sorting the same input data, consider to define a main function in all implementations that sort a given input. You should consider two methodologies to produce such (large) inputs: define a function that generates at runtime the input to be sorted, or to define such input as a (global) static variable. Because sorting algorithms may have different performance when executed with small or large inputs, you should consider three types of inputs to sort: small, medium and large.

3. Execute all implementations you defined within the energy PL ranking framework.

4. Write scripts to execute all programs (possibly organized in different folders according to their language) and to compile a single csv file that includes all csv files produced by running on program.

5. Which is the fastest sorting program? which is the greenest sorting program? Is the difference in terms of runtime similar to the difference in energy?

Exercise 3: The energy PL ranking framework forces a delay (sleep) between executions so that the temperature of the chip does not influence the energy measurements. In linux, the temperature of the chip (CPU) can be measured by the `lm-sensors` package, available on github at

https://github.com/lm-sensors/lm-sensors

1. Update the `main` function of the ranking framework so that sensors are used to ensure that there is no influence on measurements between consecutive executions of a program.

2. Consider again the sorting implementations of the previous exercise. Re-run all implementations using the updated version of the ranking framework. Are there significant changes in the measurements?

Exercise 4: When we need to limit the energy consumption of a device, we may use a mechanism to guarantee that the consumption is below some power cap. For example, this happens when we use a dimmable lamp, and we reduce its consumption by decreasing the button that controls it. RAPL CAP provides a C interface for getting/setting power caps with RAPL.

https://github.com/powercap/raplcap

Thus, we may execute a program under a given power cap.

1. Update the `main` function of the ranking framework in order to be possible to define the power cap under which the programs need to be executed.

2. Consider the Fibonacci implementations of Exercise 1. Re-run the implementations using different power caps. How did the power cap influenced the overall energy consumption and runtime of the Fibonacci implemenations?

3. Consider again the sorting implementations of Exercise 2. Re-run the implementations using different power caps. How did the power cap influenced the overall energy consumption and runtime of the sorting algorithms implementations?

4 How to install

Refer to this section if any troubles arise installing the tools required for the exercises.

4.1 Sensors

This library supplies tools for measuring the temperature of hardware components. In most machines, installing and testing if it works properly should be quite simple:

```
#installing the library
sudo apt install lm-sensors

#measuring sensors
sensors
```

It should be usable directly in C code, but it simpler to just run the terminal executable. We recommend using the *sensors* terminal executable in the exercises, and integrating it with your build in one of two ways:

- Adapt your $RAPL/main.c$ file to execute the code to be measured only once. Create a shell script where you run the *main.c* file N times with whatever needs to be measured. Use the *sensors* command inside said script, handle its output and sleep while the temperature is too high.

- Adapt your $RAPL/main.c$ file to use $system(sensors)$ to run *sensors* from within C, handle its output, and sleep while the temperature is too high.

4.2 Powercap

The Powercap library is required by *raplcap*. The tool *cmake* needs to be previously installed on the machine.

https://github.com/powercap/powercap.

These instructions are taken from the library's repository:

```
#download the library
git clone https://github.com/powercap/powercap.git
cd powercap

#compile the library
mkdir _build
cd _build
cmake ..
make

#install the library. You might need elevated user previleges
make install
```

If it is installed correctly, you should be able to compile any C file with the following include line:

```
#include <powercap/powercap.h>
```

4.3 Raplcap

This library enables the definition of upper bounds of energy consumption for an Intel CPU. The installation steps are very similar to *powercap*.

```
#download the library
git clone https://github.com/powercap/raplcap.git
cd raplcap

#compile the library
mkdir _build
cd _build
cmake ..
make

#install the library. You might need elevated user previleges
make install
```

You will need additional flags for compilation with this library. This is explained in https://github.com/powercap/raplcap#linking. You can find out which flags are needed by running the following command, after which we recommend adding either the command itself or the resulting flags to your makefile or shell script:

```
pkg-config --libs --static raplcap-msr
```

If it is installed correctly, you should be able to compile any C file with the following include line:

```
#include <raplcap/raplcap.h>
```

For usage examples of this library, refer to https://github.com/powercap/raplcap#usage. **Note that the provided example has a typo: where it reads** $d =$

$raplcap\_get\_num\_die(rc, 0)$; **it should read** $d = raplcap\_get\_num\_die(\&rc, 0)$;**, i.e. the variable** rc **is missing a &.**

References

[HDVH12] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17, jan 2012.

[KHN+18] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. RAPL in action: Experiences in using rapl for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), mar 2018.

[PCR+17] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, pages 256–267, New York, NY, USA, 2017. ACM.