pyRAPL

Release 0.2.0

INRIA, University of Lille

Dec 19, 2019

CONTENTS

1 Quickstart **1**
- 1.1 Installation . . . 1
- 1.2 Basic usage . . . 1
  - 1.2.1 Decorate a function to measure its energy consumption . . . 1
  - 1.2.2 Configure the decorator specifying the device to monitor . . . 2
  - 1.2.3 Running the test multiple times . . . 2
  - 1.2.4 Configure the output of the decorator . . . 2
  - 1.2.5 Measure the energy consumption of a piece of code . . . 3
  - 1.2.6 Measure the energy consumption of a block . . . 3

2 API **5**
- 2.1 Enumeration . . . 5
- 2.2 Functions . . . 5
- 2.3 Decorator . . . 5
- 2.4 Class . . . 6
- 2.5 Exception . . . 7

3 Outputs API **9**
- 3.1 Abstract Class . . . 9
- 3.2 Class . . . 10

4 About **13**

5 Miscellaneous **15**
- 5.1 Mailing list and contact . . . 15
- 5.2 Contributing . . . 15

Python Module Index **17**

Index **19**

CHAPTER

ONE

QUICKSTART

1.1 Installation

You can install **pyRAPL** with pip : `pip install pyRAPL`

1.2 Basic usage

Here are some basic usages of **pyRAPL**. Please note that the reported energy consumption is not only the energy consumption of the code you are running. This includes the *global energy consumption* of all the process running on the machine during this period, thus including the operating system and other applications. That is why we recommend to eliminate any extra programs that may alter the energy consumption of the machine hosting experiments and to keep only the code under measurement (*i.e.*, no extra applications, such as graphical interface, background running task. . .). This will give the closest measure to the real energy consumption of the measured code.

Here are some basic usages of pyRAPL. Please understand that the measured energy consumption is not only the energy consumption of the code you are running. It's the **global energy consumption** of all the process running on the machine during this period. This includes also the operating system and other applications. That's why we recommend eliminating any extra programs that may alter the energy consumption of the machine where we run the experiments and keep **only** the code we want to measure its energy consumption (no extra applications such as graphical interface, background running task . . .). This will give the closest measure to the real energy consumption of the measured code.

1.2.1 Decorate a function to measure its energy consumption

To measure the energy consumed by the machine during the execution of the function `foo()` run the following code:

```
import pyRAPL

pyRAPL.setup()

@pyRAPL.measureit
def foo():
  # Instructions to be evaluated.

foo()
```

This will print the recorded energy consumption of all the monitorable devices of the machine during the execution of function `fun`.

1.2.2 Configure the decorator specifying the device to monitor

You can easly configure which device and which socket to monitor using the parameters of the `pyRAPL.setup` function. For example, the following example only monitors the CPU power consumption on the CPU socket `1`. By default, **pyRAPL** monitors all the available devices of the CPU sockets:

```
import pyRAPL

pyRAPL.setup(devices=[pyRAPL.Device.PKG], socket_ids=[1])

@pyRAPL.measureit
def foo():
  # Instructions to be evaluated.

foo()
```

You can append the device `pyRAPL.Device.DRAM` to the `devices` parameter list to monitor RAM device too.

1.2.3 Running the test multiple times

For short functions, you can configure the number of runs and it will calculate the mean energy consumption of all runs. As an example if you want to run the evaluation 100 times

```
import pyRAPL

pyRAPL.setup()

@pyRAPL.measureit(number=100)
def foo():
    # Instructions to be evaluated.

for _ in range(100):
    foo()
```

1.2.4 Configure the output of the decorator

If you want to handle data with different output than the standard one, you can configure the decorator with an `Output` instance from the `pyRAPL.outputs` module.

As an example if you want to write the recorded energy consumption in a csv file

```
import pyRAPL

pyRAPL.setup()

csv_output = pyRAPL.outputs.CSVOutput('result.csv')

@pyRAPL.measureit(output=csv_output)
def foo():
  # Some stuff ...

for _ in range(100):
  foo()

csv_output.save()
```

This will produce a csv file of 100 lines. Each line containing the energy consumption recorded during one execution of the function *fun*. Other predefined Output classes exist to export data to *MongoDB* and *Panda* dataframe. You can also create your own Output class (see the documentation)

1.2.5 Measure the energy consumption of a piece of code

To measure the energy consumed by the machine during the execution of a given piece of code, run the following code:

```
import pyRAPL

pyRAPL.setup()
measure = pyRAPL.Measurement('bar')
measure.begin()

# ...
# Instructions to be evaluated.
# ...

measure.end()
```

You can also access the result of the measurements using the property : `measure.result` which returns a Result instance.

You can also use an output to handle this results, for example with the csv output : `measure.export(csv_output)`

1.2.6 Measure the energy consumption of a block

pyRAPL allows also to measure a block of instructions using the Keyword `with` as the example below:

```
import pyRAPL
pyRAPL.setup()

with pyRAPL.Measurement('bar'):
  # ...
  # Instructions to be evaluated.
  # ...
```

This will print in the console the energy consumption of the block. To handle the measures instead of just printing them you can use any Output class that you pass to the Measurement object

```
import pyRAPL
pyRAPL.setup()

dataoutput= pyRAPL.outputs.DataFrameOutput()
with pyRAPL.Measurement('bar',output=dataoutput):

  # ...
  # Instructions to be evaluated.
  # ...

dataoutput.data.head()
```

CHAPTER

TWO

API

2.1 Enumeration

class pyRAPL.**Device**

Device that can be monitored by pyRAPL

Device.PKG : to monitor the CPU energy consumption

Device.DRAM : to monitor the RAM energy consumption

2.2 Functions

pyRAPL.**setup**(*devices=None*, *socket_ids=None*)

Configure which device and CPU socket should be monitored by pyRAPL

This function must be called before using any other pyRAPL functions

Parameters

- **devices** (Optional[List[Device]]) – list of monitored devices if None, all the available devices on the machine will be monitored
- **socket_ids** (Optional[List[int]]) – list of monitored sockets, if None, all the available socket on the machine will be monitored

Raises

- ***PyRAPLCantRecordEnergyConsumption*** – if the sensor can't get energy information about the given device in parameter
- ***PyRAPLBadSocketIdException*** – if the given socket in parameter doesn't exist

2.3 Decorator

@pyRAPL.**measureit**(*_func=None*, *, *output=None*, *number=1*)

Measure the energy consumption of monitored devices during the execution of the decorated function (if multiple runs it will measure the mean energy)

Parameters

- **output** (Optional[Output]) – output instance that will receive the power consummation data

- **number** (`int`) – number of iteration in the loop in case you need multiple runs or the code is too fast to be measured

2.4 Class

class `pyRAPL.`**Measurement**(*label*, *output=None*)

measure the energy consumption of devices on a bounded period

Beginning and end of this period are given by calling `begin()` and `end()` methods

Parameters

- **label** (`str`) – measurement label
- **output** (`Optional[Output]`) – default output to export the recorded energy consumption. If None, the PrintOutput will be used

begin()

Start energy consumption recording

end()

End energy consumption recording

export(*output=None*)

Export the energy consumption measures to a given output

Parameters **output** (`Optional[Output]`) – output that will handle the measure, if None, the default output will be used

property result

Access to the measurement data

Return type `Result`

class `pyRAPL.`**Result**(*label*, *timestamp*, *duration*, *pkg=None*, *dram=None*)

A data class to represent the energy measures

Variables

- **label** (*`str`*) – measurement label
- **timestamp** (*`float`*) – measurement's beginning time (expressed in seconds since the epoch)
- **duration** (*`float`*) – measurement's duration (in micro seconds)
- **pkg** (*`Optional[List[float]]`*) – list of the CPU energy consumption -expressed in micro Joules- (one value for each socket) if None, no CPU energy consumption was recorded
- **dram** (*`Optional[List[float]]`*) – list of the RAM energy consumption -expressed in seconds- (one value for each socket) if None, no RAM energy consumption was recorded

2.5 Exception

exception pyRAPL.**PyRAPLException**

Parent class of all PyRAPL exception

exception pyRAPL.**PyRAPLCantRecordEnergyConsumption**(*device*)

Exception raised when starting recording energy consumption for a device but no energy consumption metric is available for this device

Variables **device** (Device) – device that couldn't be monitored (if None, Any device on the machine could be monitored)

exception pyRAPL.**PyRAPLBadSocketIdException**(*socket_id*)

Exception raised when trying to initialise PyRAPL on a socket that doesn't exist on the machine

Variables **socket_id** (*int*) – socket that doesn't exist

CHAPTER

THREE

OUTPUTS API

This module contains class that will be used by the `measure` decorator or the `Measurement.export` method to export recorded measurement

example with the `measure decorator`:

```
output_instance = pyRAPL.outputs.XXXOutput(...)

@pyRAPL.measure(output=output_instance)
def foo():
    ...
```

example with the `Measurement.export` function:

```
measure = pyRAPL.Measurement('label')
...
output_instance = pyRAPL.outputs.XXXOutput(...)
measure.export(output_instance)
```

You can define your one output by inherit from the `Output` class and implements the `add` method. This method will receive the measured energy consumption data as a `Result` instance and must handle it.

For example, the `PrintOutput.add` method will print the `Result` instance.

3.1 Abstract Class

class pyRAPL.outputs.**Output**

Abstract class that represent an output handler for the *Measurement* class

add(*result*)

Handle the object *Result*

Parameters **result** (`Result`) – data to handle

class pyRAPL.outputs.**BufferedOutput**

Use a buffer to batch the output process

The method `add` add data to the buffer and the method `save` outputs each data in the buffer. After that, the buffer is flushed

Implement the abstract method `_output_buffer` to define how to output buffered data

_output_buffer()

Abstract method

Output all the data contained in the buffer

Parameters **data** – data to output

add(*result*)
Add the given data to the buffer

Parameters **result** – data that must be added to the buffer

property buffer
Return the buffer content

Return type `List[Result]`

Returns a list of all the `Result` instances contained in the buffer

save()
Output each data in the buffer and empty the buffer

3.2 Class

class pyRAPL.outputs.**PrintOutput**(*raw=False*)
Output that print data on standard output

Parameters **raw** (`bool`) – if True, print the raw result class to standard output. Otherwise, print a fancier representation of result

add(*result*)
print result on standard output

Parameters **result** (`Result`) – data to print

class pyRAPL.outputs.**CSVOutput**(*filename*, *separator=','*, *append=True*)
Write the recorded measure in csv format on a file

if the file already exists, the result will be append to the end of the file, otherwise it will create a new file.

This instance act as a buffer. The method `add` add data to the buffer and the method `save` append each data in the buffer at the end of the csv file. After that, the buffer is flushed

Parameters

- **filename** (`str`) – file's name were the result will be written
- **separator** (`str`) – character used to separate columns in the csv file
- **append** (`bool`) – Turn it to False to delete file if it already exist.

class pyRAPL.outputs.**MongoOutput**(*uri*, *database*, *collection*)
Store the recorded measure in a MongoDB database

This instance act as a buffer. The method `add` add data to the buffer and the method `save` store each data in the buffer in the MongoDB database. After that, the buffer is flushed

Parameters

- **uri** (`str`) – uri used to connect to the mongoDB instance
- **database** (`str`) – database name to store the data
- **collection** (`str`) – collection name to store the data

class pyRAPL.outputs.**DataFrameOutput**
Append recorded data to a pandas Dataframe

add(*result*)

Append recorded data to the pandas Dataframe

Parameters **result** – data to add to the dataframe

property data

Return the dataframe that contains the recorded data

Return type `DataFrame`

Returns the dataframe

CHAPTER

FOUR

ABOUT

pyRAPL is a software toolkit to measure the energy footprint of a host machine along the execution of a piece of Python code.

pyRAPL uses the Intel "*Running Average Power Limit*" (RAPL) technology that estimates power consumption of a CPU. This technology is available on Intel CPU since the Sandy Bridge generation.

More specifically, pyRAPL can measure the energy consumption of the following CPU domains:

- CPU socket package
- DRAM (for server architectures)
- GPU (for client architectures)

CHAPTER

FIVE

MISCELLANEOUS

PyRAPL is an open-source project developed by the Spirals research group (University of Lille and Inria) that take part of the Powerapi initiative.

5.1 Mailing list and contact

You can contact the developer team with this address : powerapi-staff@inria.fr

You can follow the latest news and asks questions by subscribing to our mailing list

5.2 Contributing

If you would like to contribute code you can do so via GitHub by forking the repository and sending a pull request.

When submitting code, please make every effort to follow existing coding conventions and style in order to keep the code as readable as possible.

PYTHON MODULE INDEX

p

pyRAPL.outputs, 9

INDEX

Symbols

`_output_buffer()` (*pyRAPL.outputs.BufferedOutput method*), 9

A

`add()` (*pyRAPL.outputs.BufferedOutput method*), 10
`add()` (*pyRAPL.outputs.DataFrameOutput method*), 10
`add()` (*pyRAPL.outputs.Output method*), 9
`add()` (*pyRAPL.outputs.PrintOutput method*), 10

B

`begin()` (*pyRAPL.Measurement method*), 6
`buffer()` (*pyRAPL.outputs.BufferedOutput property*), 10
`BufferedOutput` (*class in pyRAPL.outputs*), 9

C

`CSVOutput` (*class in pyRAPL.outputs*), 10

D

`data()` (*pyRAPL.outputs.DataFrameOutput property*), 11
`DataFrameOutput` (*class in pyRAPL.outputs*), 10
`Device` (*class in pyRAPL*), 5

E

`end()` (*pyRAPL.Measurement method*), 6
`export()` (*pyRAPL.Measurement method*), 6

M

`measureit()` (*in module pyRAPL*), 5
`Measurement` (*class in pyRAPL*), 6
`MongoOutput` (*class in pyRAPL.outputs*), 10

O

`Output` (*class in pyRAPL.outputs*), 9

P

`PrintOutput` (*class in pyRAPL.outputs*), 10
`pyRAPL.outputs` (*module*), 9
`PyRAPLBadSocketIdException`, 7
`PyRAPLCantRecordEnergyConsumption`, 7
`PyRAPLException`, 7

R

`Result` (*class in pyRAPL*), 6
`result()` (*pyRAPL.Measurement property*), 6

S

`save()` (*pyRAPL.outputs.BufferedOutput method*), 10
`setup()` (*in module pyRAPL*), 5