International Conference on
Energy-Aware High Performance Computing

# Automatic Detection of Power Bottlenecks in Parallel Scientific Applications

Maria Barreda[1], Sandra Catalán[1],
Manuel F. Dolz[2], Rafael Mayo[1],
Enrique S. Quintana-Ortí[1]

1 Universitat Jaume I

2 Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

September 2nd – 3rd, 2013, Dresden (Germany)

# Motivation

- High performance computing:
  - Optimization of algorithms applied to solve complex problems
- Technological advance ⇒ improve performance:
  - Higher number of cores per socket (processor)
- Large number of processors and cores ⇒ High energy consumption
- Tools to analyze performance and power in order to detect code inefficiencies and reduce energy consumption

# Motivation

A framework for power-performance analysis of parallel applications

⇓

Detect energy inefficiencies in the code

⇓

Can be tackled to reduce the energy consumption

# Motivation

A framework for power-performance analysis of parallel applications

$\Downarrow$

Detect energy inefficiencies in the code

$\Downarrow$

Can be tackled to reduce the energy consumption

# Motivation

A framework for power-performance analysis of parallel applications

$\Downarrow$

Detect energy inefficiencies in the code

$\Downarrow$

Can be tackled to reduce the energy consumption

# Outline

# Introduction

- **Framework for power and energy analysis**

Parallel applications + Power profiling + C-states profiling

$\Downarrow$

Environment to identify sources of power inefficiency

- **Extension of the power profiling framework (pmlib)**

Inspection tool

$\Downarrow$

Automatic detection of power sinks

# Introduction

- **Framework for power and energy analysis**

  Parallel applications + Power profiling + C-states profiling

  $\Downarrow$

  Environment to identify sources of power inefficiency

- **Extension of the power profiling framework (pmlib)**

  Inspection tool

  $\Downarrow$

  Automatic detection of power sinks

# Introduction

**Inspection tool:**

- Discrepancies between the application activity and the CPU C-states
- Multithreaded Python module
- High reliability and flexibility
- Visual trace and analytical report

# Introduction

**Additional contribution:**

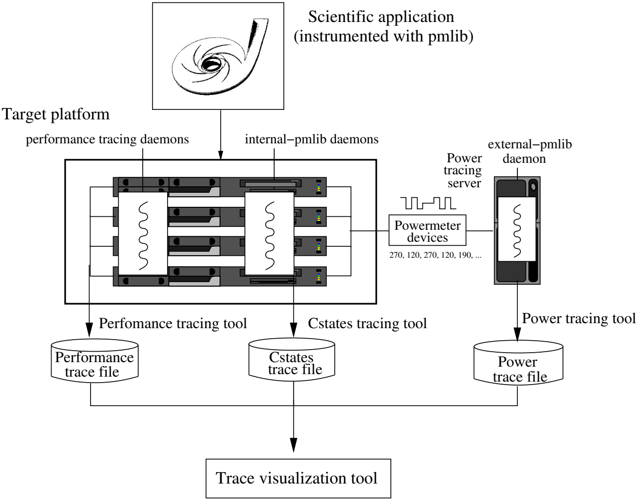
- Evaluate two power sampling/modeling approaches
  - A DC wattmeter using a DAS from National Instruments
  - The RAPL in an Intel Xeon E5-2620 processor
- Advantages and drawbacks:
  - Accuracy
  - Sampling rates
  - Overhead introduced during the execution

# Introduction

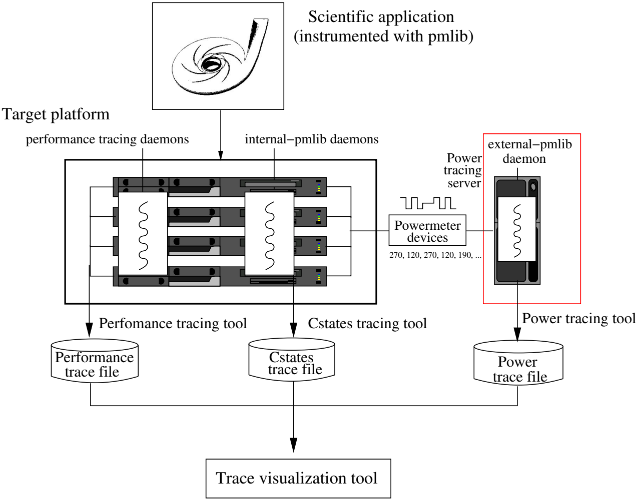
**Additional contribution:**

- Evaluate two power sampling/modeling approaches
  - A DC wattmeter using a DAS from National Instruments
  - The RAPL in an Intel Xeon E5-2620 processor
- Advantages and drawbacks:
  - Accuracy
  - Sampling rates
  - Overhead introduced during the execution

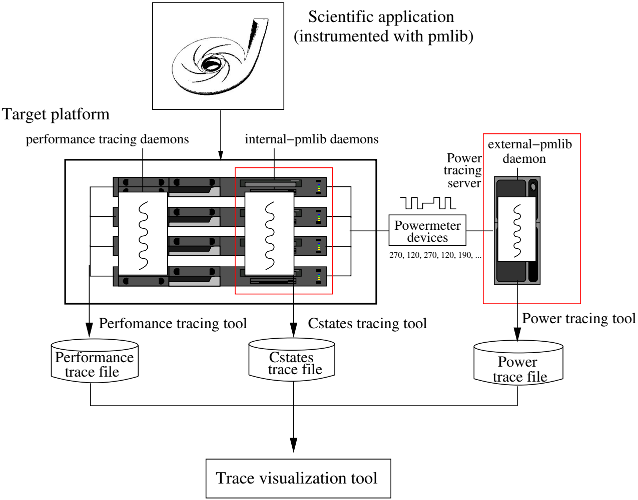
# Performance-Power Monitoring using `pmlib`



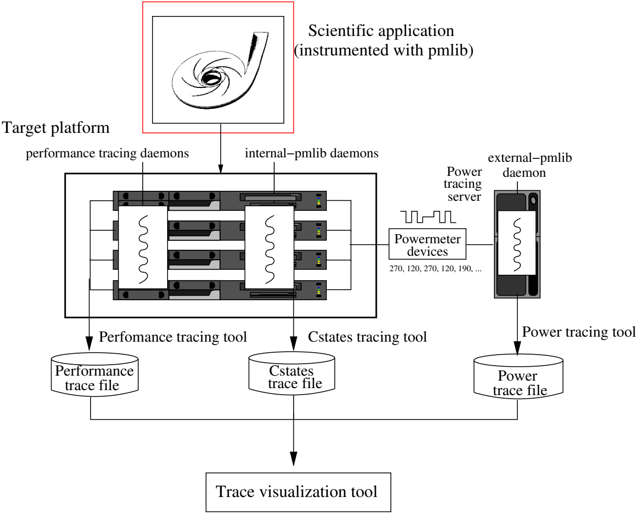
# Performance-Power Monitoring using `pmlib`



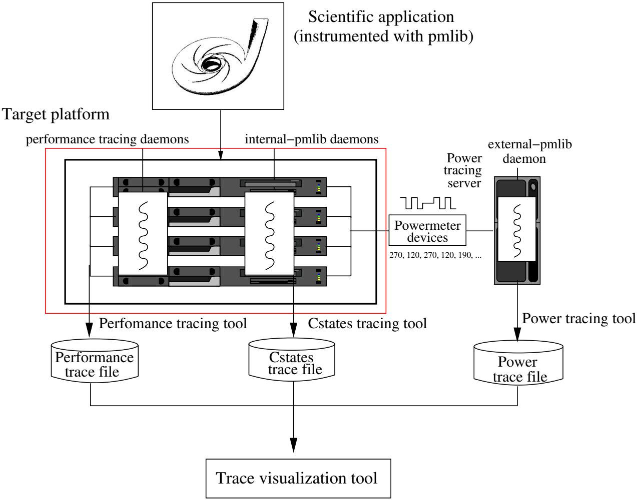
# Performance-Power Monitoring using `pmlib`



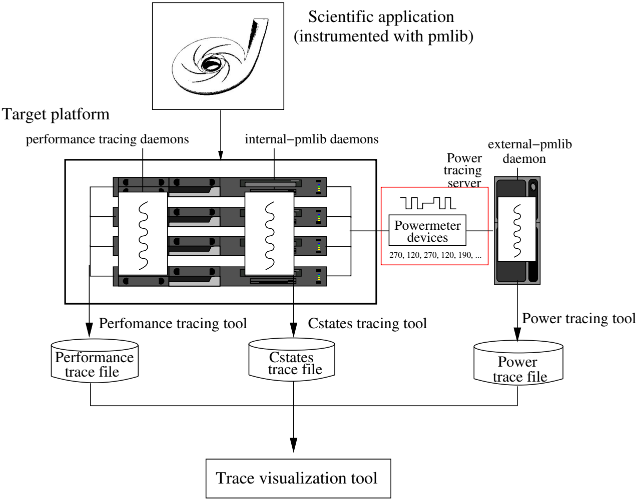
# Performance-Power Monitoring using pmlib



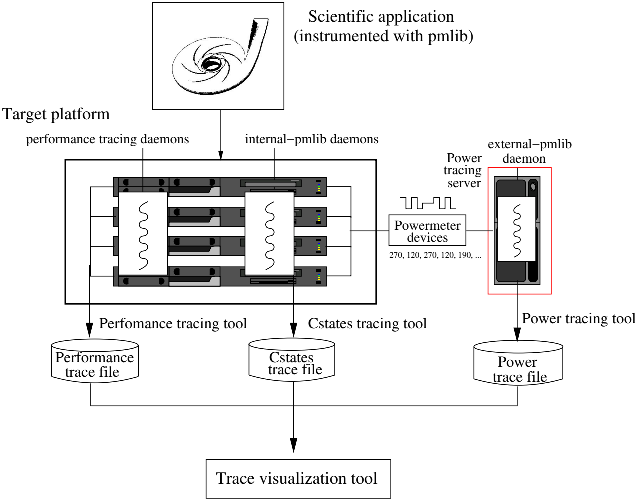
# Performance-Power Monitoring using `pmlib`



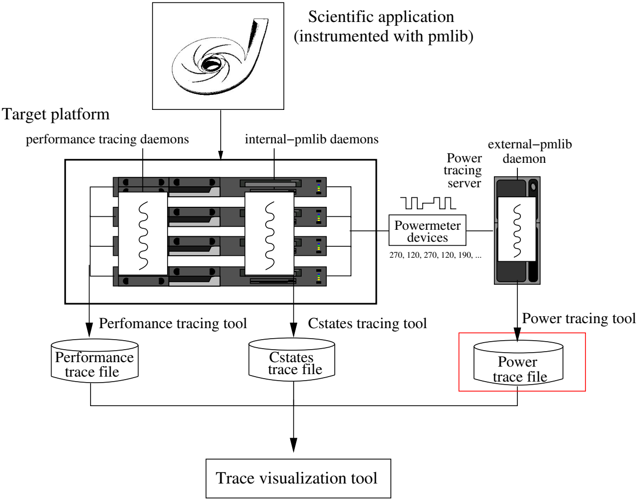
# Performance-Power Monitoring using pmlib



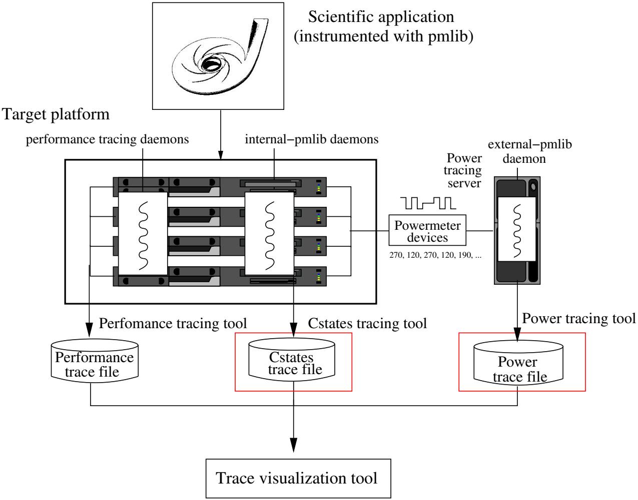
# Performance-Power Monitoring using pmlib



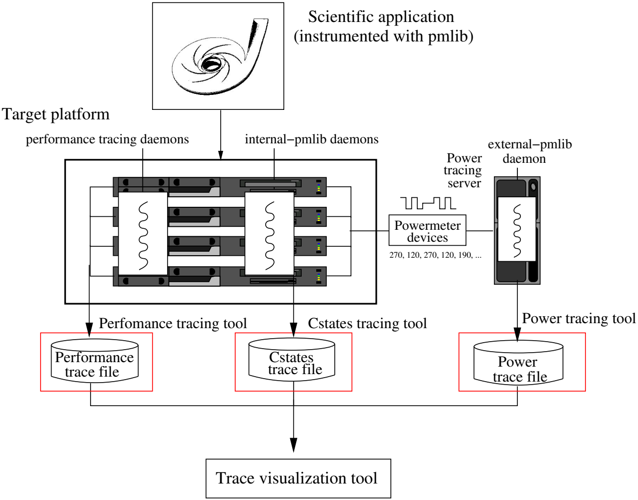
# Performance-Power Monitoring using pmlib



# Performance-Power Monitoring using `pmlib`



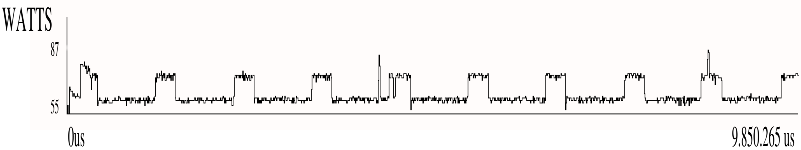
# Performance-Power Monitoring using `pmlib`



# Performance-Power Monitoring using pmlib

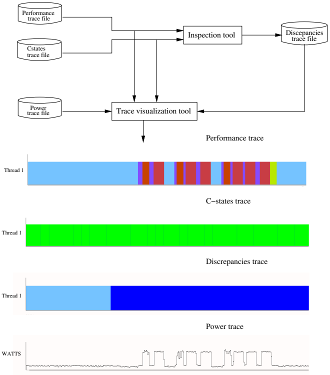
Performance trace


Power trace



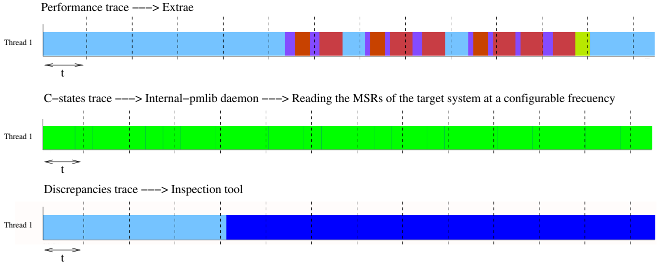
# Overview

**Parallel analyser to detect power bottlenecks**




- Automates and accelerates the inspection process
- Process more reliable
- Flexible analyzer
  - Task type that is "useful" work
  - Lenght of the analysis interval
  - Discrepancy threshold

# Operation and implementation



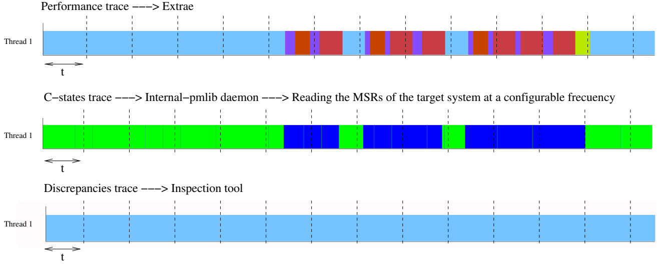

## Implementation:

- Python
- Intervals of length t ⇒ Inserted into a task pool
- Multithreaded analyzer ⇒ Python Pool class

## Result

- Analytical ⇒ (c, $t_i$, $t_f$, %divergence)
- Graphical ⇒ Paraver

# Operation and implementation




## Implementation:

- Python
- Intervals of length t ⇒ Inserted into a task pool
- Multithreaded analyzer ⇒ Python Pool class

## Result

- Analytical ⇒ (c, $t_i$, $t_f$, %divergence)
- Graphical ⇒ Paraver

# Examples

- **Ilupack**
- **LU factorization using libflame**

# Ilupack

- Concurrent solution of **sparse linear systems**
- Multilevel preconditioners for general and Hermitian positive definite problems
- **Parallelization** $\Rightarrow$ Task partitioning of the sparsity graph

$\Downarrow$

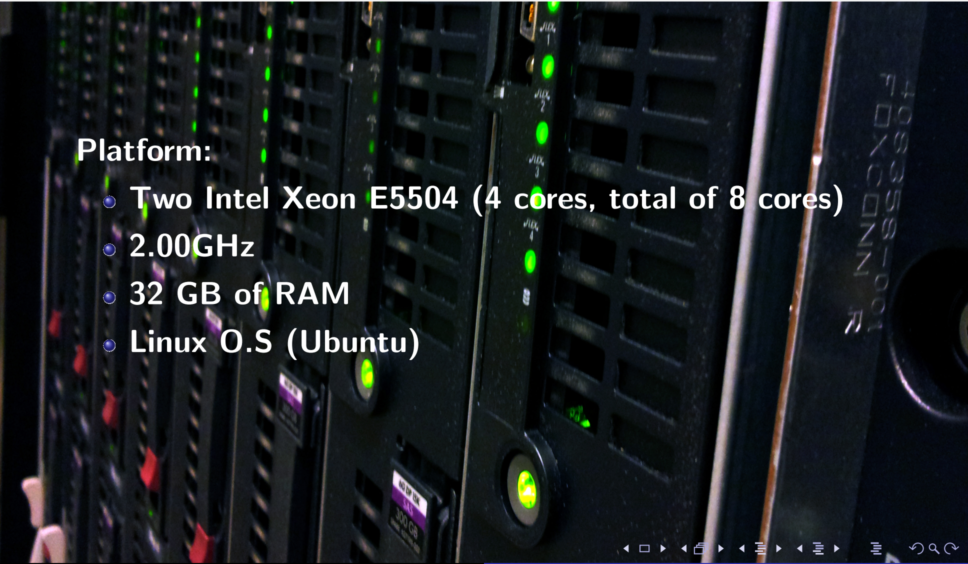
Task acyclic graph capturing dependencies

$\Downarrow$

Tasks mapped to threads on-demand at runtime

- An idle thread **polls** the queue till a ready task becomes available

# Ilupack

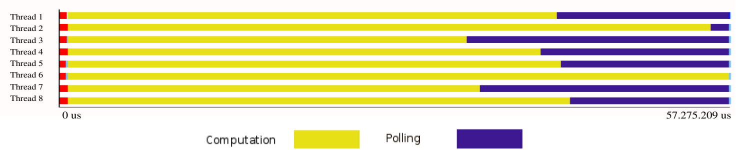


**Platform:**

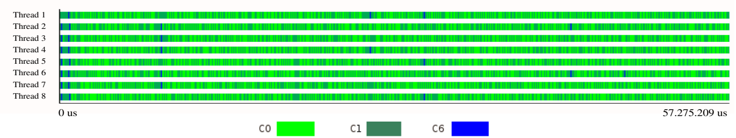
- **Two Intel Xeon E5504 (4 cores, total of 8 cores)**
- **2.00GHz**
- **32 GB of RAM**
- **Linux O.S (Ubuntu)**

# Ilupack

Performance trace




C-states trace




Discrepancies trace

# LU Factorization

- LU factorization with partial pivoting of a **dense matrix**
- **FLA_LU** routine of **libflame** library
- Parallelized with the **Supermatrix** runtime
- **Hybrid CPU_GPU version:**
  - Computation on CPU ⇒ Intel MKL
  - Computation on GPU ⇒ NVIDIA CUBLAS and CUDA
  - Runtime controls the data transfers

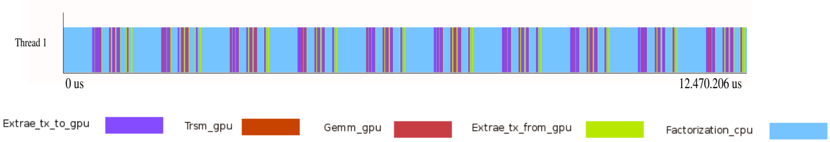
# LU Factorization

**Hybrid Platform:**

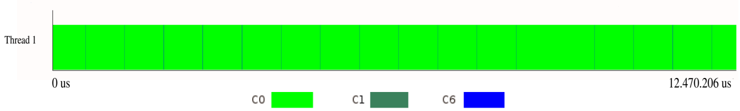
- **Intel Xeon i7-3770**
- **16 GB of RAM**
- **NVIDIA Tesla C2050 ("Fermi")**

# LU Factorization

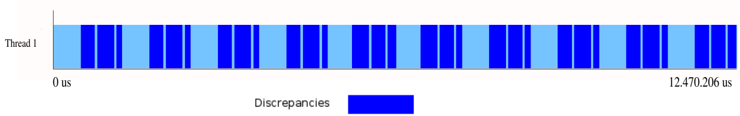
Performance trace




C-states trace




Discrepancies trace



# Impact of power sinks

## Statistical information

| | Computation | Polling | C0 | C1 | C6 | Discrepancies |
|---|---|---|---|---|---|---|
| THREAD 1 | 72.00% | 25.56% | 99.33% | 0.29% | 0.39% | 27.49% |
| THREAD 2 | 96.45% | 2.50% | 99.25% | 0.26% | 0.50% | 4.77% |
| THREAD 3 | 59.90% | 39.14% | 99.53% | 0.10% | 0.37% | 40.59% |
| THREAD 4 | 70.81% | 28.13% | 99.48% | 0.10% | 0.42% | 30.11% |
| THREAD 5 | 74.00% | 25.14% | 99.29% | 0.90% | 0.61% | 26.61% |
| THREAD 6 | 99.18% | 0.00% | 99.34% | 0.22% | 0.45% | 0.00% |
| THREAD 7 | 61.52% | 37.17% | 99.53% | 0.12% | 0.35% | 38.84% |
| THREAD 8 | 75.03% | 23.69% | 99.27% | 0.10% | 0.64% | 25.74% |

$\Downarrow$

Estimation of the costs of the power sinks

# Impact of power sinks

## Statistical information

| | Computation | Polling | C0 | C1 | C6 | Discrepancies |
|---|---|---|---|---|---|---|
| THREAD 1 | 72.00% | 25.56% | 99.33% | 0.29% | 0.39% | 27.49% |
| THREAD 2 | 96.45% | 2.50% | 99.25% | 0.26% | 0.50% | 4.77% |
| THREAD 3 | 59.90% | 39.14% | 99.53% | 0.10% | 0.37% | 40.59% |
| THREAD 4 | 70.81% | 28.13% | 99.48% | 0.10% | 0.42% | 30.11% |
| THREAD 5 | 74.00% | 25.14% | 99.29% | 0.90% | 0.61% | 26.61% |
| THREAD 6 | 99.18% | 0.00% | 99.34% | 0.22% | 0.45% | 0.00% |
| THREAD 7 | 61.52% | 37.17% | 99.53% | 0.12% | 0.35% | 38.84% |
| THREAD 8 | 75.03% | 23.69% | 99.27% | 0.10% | 0.64% | 25.74% |

$\Downarrow$

Estimation of the costs of the power sinks

# Energy-costs due to hotspots

- Time that cores are doing "useless" work ⇒ Wasting power
- Overall energy cost:
  - Power rate of a core in a power-saving sleep state
  - Power consumption in the trace corresponding to the "guilty" core(s)

⇓

Design a test that mimics the power sink:
ILUPACK: busy-wait
LU: CUDA invocations

- Potential savings:
  - (Power("guilty" core) - Power(power-saving state)) * total duration power sinks

# Energy-costs due to hotspots

- Time that cores are doing “useless” work ⇒ Wasting power
- Overall energy cost:
  - Power rate of a core in a power-saving sleep state
  - Power consumption in the trace corresponding to the “guilty” core(s)

⇓

Design a test that mimics the power sink:
ILUPACK: busy-wait
LU: CUDA invocations

- Potential savings:
  - (Power(“guilty” core) - Power(power-saving state)) * total duration power sinks

# Energy-costs due to hotspots

- Time that cores are doing "useless" work ⇒ Wasting power
- Overall energy cost:
  - Power rate of a core in a power-saving sleep state
  - Power consumption in the trace corresponding to the "guilty" core(s)

⇓

Design a test that mimics the power sink:
ILUPACK: busy-wait
LU: CUDA invocations

- Potential savings:
  - (Power("guilty" core) - Power(power-saving state)) * total duration power sinks

# Energy-costs due to hotspots

- Time that cores are doing "useless" work ⇒ Wasting power
- Overall energy cost:
  - Power rate of a core in a power-saving sleep state
  - Power consumption in the trace corresponding to the "guilty" core(s)

⇓

Design a test that mimics the power sink:
ILUPACK: busy-wait
LU: CUDA invocations

- Potential savings:
  - (Power("guilty" core) - Power(power-saving state)) * total duration power sinks

# Energy-costs due to hotspots

- Time that cores are doing "useless" work ⇒ Wasting power
- Overall energy cost:
  - Power rate of a core in a power-saving sleep state
  - Power consumption in the trace corresponding to the "guilty" core(s)

⇓

Design a test that mimics the power sink:

ILUPACK: busy-wait

LU: CUDA invocations

- Potential savings:
  - (Power("guilty" core) - Power(power-saving state)) * total duration power sinks

# Energy-costs due to hotspots

- Time that cores are doing "useless" work ⇒ Wasting power
- Overall energy cost:
  - Power rate of a core in a power-saving sleep state
  - Power consumption in the trace corresponding to the "guilty" core(s)

⇓

Design a test that mimics the power sink:
ILUPACK: busy-wait
LU: CUDA invocations

- Potential savings:
  - (Power("guilty" core) - Power(power-saving state)) * total duration power sinks

# Energy-costs due to hotspots

- Time that cores are doing "useless" work ⇒ Wasting power
- Overall energy cost:
  - Power rate of a core in a power-saving sleep state
  - Power consumption in the trace corresponding to the "guilty" core(s)

⇓

Design a test that mimics the power sink:
ILUPACK: busy-wait
LU: CUDA invocations

- Potential savings:
  - (Power("guilty" core) - Power(power-saving state)) * total duration power sinks

# Energy-costs due to hotspots

- Time that cores are doing "useless" work ⇒ Wasting power
- Overall energy cost:
  - Power rate of a core in a power-saving sleep state
  - Power consumption in the trace corresponding to the "guilty" core(s)

⇓

Design a test that mimics the power sink:
ILUPACK: busy-wait
LU: CUDA invocations

- Potential savings:
  - (Power("guilty" core) - Power(power-saving state)) * total duration power sinks

# Power Sampling Interfaces

**Comparison in the Intel Xeon E5-2620**

**RAPL model:**

- Power estimates per CPU socket
- Access the MSRs periodically
- Internal-pmlib daemon

$\Downarrow$

**Overhead?**

**DAS form National Instruments:**

- 32 channels
- 7,000 samples/sec
- External-pmlib daemon

$\Downarrow$

**No overhead**

# Experiments

**Access the MSRs ⇒ Overhead?**

**Synthetic test**

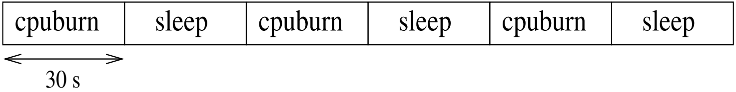
**Power profiles**

- Synchronization of the measures
- The rates from the NI are higher than those from the RAPL

# Experiments

**Access the MSRs ⇒ Overhead?**

## Synthetic test



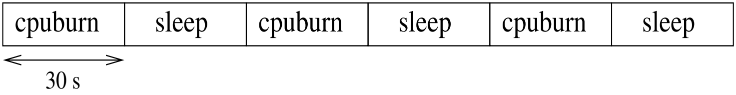

## Power profiles

Synchronization of the measures
The rates from the NI are higher than those from the RAPL

# Experiments

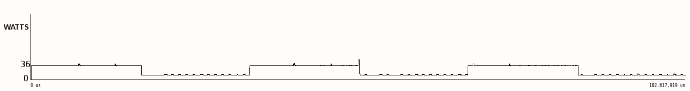
**Access the MSRs ⇒ Overhead?**

**Synthetic test**




**Power profiles**

RAPL



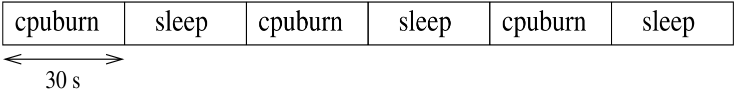

Synchronization of the measures

The rates from the NI are higher than those from the RAPL

# Experiments

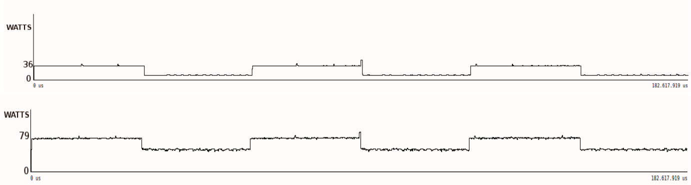
**Access the MSRs ⇒ Overhead?**

## Synthetic test




## Power profiles

RAPL
NI

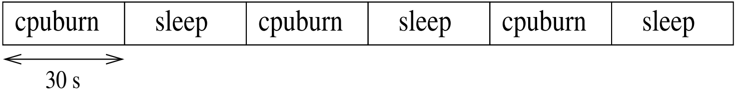



- Synchronization of the measures
- The rates from the NI are higher than those from the RAPL

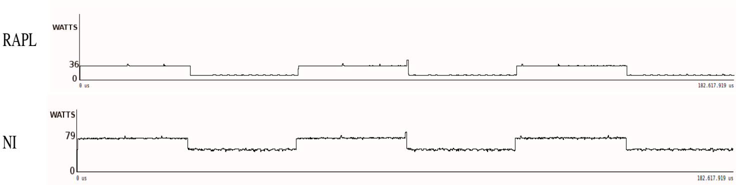
# Experiments

**Access the MSRs ⇒ Overhead?**

## Synthetic test




## Power profiles

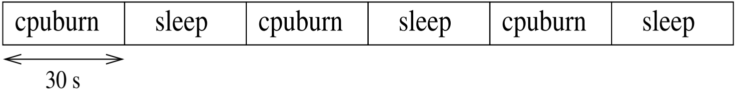



- Synchronization of the measures
- The rates from the NI are higher than those from the RAPL

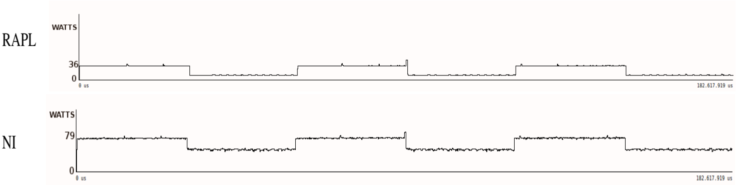
# Experiments

**Access the MSRs ⇒ Overhead?**

## Synthetic test




## Power profiles



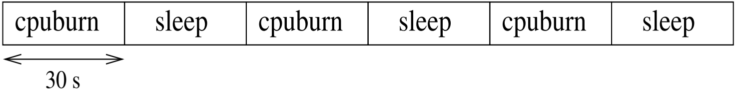

Synchronization of the measures

The rates from the NI are higher than those from the RAPL

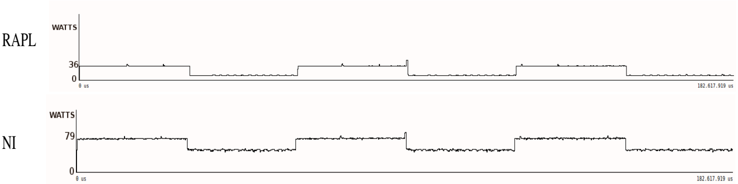
# Experiments

**Access the MSRs ⇒ Overhead?**

## Synthetic test




## Power profiles




Synchronization of the measures
The rates from the NI are higher than those from the RAPL

# Overhead of the internal-pmlib

## Power measurements with both daemons in simultaneous operation

| Source | freq. | idle (W) max | idle (W) avg | 1 × cpuburn (W) max | 1 × cpuburn (W) avg | 4 × cpuburn (W) max | 4 × cpuburn (W) avg |
|---|---|---|---|---|---|---|---|
| RAPL+ MSR | 1 | 9.0 | 8.7 | 24.8 | 24.0 | 49.6 | 49.2 |
| | 10 | 10.8 | 8.9 | 27.6 | 24.4 | 50.3 | 49.2 |
| | 100 | 21.0 | 9.2 | 34.0 | 24.5 | 55.3 | 49.6 |
| NI DAS | 1 | 69.1 | 43.7 | 77.6 | 62.3 | 102.9 | 92.6 |
| | 10 | 69.7 | 43.6 | 79.7 | 62.4 | 103.3 | 92.3 |
| | 100 | 68.8 | 43.2 | 78.1 | 62.7 | 102.8 | 92.5 |

A small overhead ⇒ Depending on the sampling rate

## Power measurements from NI with only that daemon in operation

| Source | idle (W) max | idle (W) avg | 1 × cpuburn (W) max | 1 × cpuburn (W) avg | 4 × cpuburn (W) max | 4 × cpuburn (W) avg |
|---|---|---|---|---|---|---|
| NI | 70.6 | 43.7 | 77.5 | 63.0 | 103.3 | 92.2 |

The overhead is negligible

# Conclusions

- **Extension of pmlib:**
  - Automatic detection of power bottlenecks
  - Compares the performance and the core C-state traces
  - Analysis in parallel ⇒ Python Pool class
  - The user can configure the process
- **The power estimation using RAPL model vs DAS system:**
  - Recording the power in the same platform using RAPL

    ⇓

    It introduces a certain overhead
- **Future work:**
  - Extend pmlib to recover information form a variety of power/temperature sensors/models
  - Accomodate other performance tracers
  - Analyse the sources of the overhead of the internal-pmlib software

# Thanks for your attention!

*Questions?*