# Energy-Aware Programming Techniques

Dominik Lohmann

Universität Hamburg
Fakultät für Informatik, Mathematik und Naturwissenschaften
Fachbereich Informatik

64-174 Seminar Energy-Efficient Programming

2014-12-03

# Outline

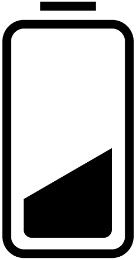
1 Introduction
   Motivation
   Energy-Awareness
   Performance

2 Computational Efficiency

3 Energy-to-Solution

4 Energy-Awareness in Practice

# Motivation



**Q:** How many of you had to charge their phone today?

# Energy-Awareness

What is energy-aware programming?

- Focus on efficiency: $\frac{\textit{Performance achieved}}{\textit{Maximum performance achievable}}$
- Optimization criterion should be decided based on TCO
- Applications need to be aware of their environment, such as power states

# Performance

What does it mean to improve performance?

- The software is going to run on a specific, real machine
- There is some theoretical limit on how quickly it can work

Improving energy-efficiency by improving performance is called computational efficiency

*"Every circuit not used on a processor is wasting power"*
– Chandler Carruth

# Outline

1 Introduction

2 Computational Efficiency
- Algorithms
- Data Efficiency: Hardware Characteristics
- Loops
- Multithreading
- Performance Libraries/Extensions
- Compiler Optimizations
- Programming Language

3 Energy-to-Solution

4 Energy-Awareness in Practice

# Algorithms

- Complexity theory allows comparing algorithm speed
- Use algorithms that allow the CPU to idle
- Note: recursive algorithms are often energy-inefficient

Improving algorithmic efficiency means solving the underlying problem in another way

# Example: Sub-String Searching

- Initially, consider a trivial $\mathcal{O}(n * m)$ algorithm
- Boyer-Moore algorithm is $\mathcal{O}(n + m)$ and can do the same thing (using the end of the needle)

Algorithmic changes can make a huge difference, but are not something that can necessarily be found by everyone

# Data Efficiency: Hardware Characteristics

| | | |
|---|---|---|
| One cycle on a 1 GHz CPU | 1 | ns |
| L1 Cache reference | 0.5 | ns |
| Branch mispredict | 5 | ns |
| L2 Cache reference | 7 | ns |
| Mutex lock/unlock | 25 | ns |
| Main memory reference | 100 | ns |
| Send 1kB over 1Gbps network | 10,000 | ns |
| Read 4kB randomly from SSD | 150,000 | ns |
| Read 1MB sequentially from memory | 250,000 | ns |
| Read 1MB sequentially from SSD | 1,000,000 | ns |

# Example: Linked Lists

- Nodes are separately allocated
- Traversal operations chase pointers to totally new memory
- In most cases, every step is a cache miss
- Only use Linked Lists when you rarely traverse the list, but frequently update it

Linked Lists are rarely what you want to use

# Loops

- Minimize the use of tight loops
- Convert polling loops to be event-driven
- Have the lowest polling frequency usable, if polling must be used
- Eliminate busy wait (spin-locks) when possible

# Multithreading

- Modern CPUs are able to run things in parallel, allowing faster computation with parallelelized algorithms
- Often requires a (partial) rewrite of legacy applications
- Balancing load across threads allows the CPU to be throttled while maintaining the performance
- Threading done right provides a massive performance boost while having almost no energy impact
- OpenMP, pthreads, TBB and PPL are examples of often used implementations

# Performance Libraries/Extensions

- Using (architecture specific) instruction sets such as SSE2 and Intel AVX can often result in increased performance
- Reducing the amount of CPU instructions per calculation directly relates to the applications energy-efficiency
- Certain applications can be optimized using hardware acceleration
  - Focuses mostly on graphics

# Compiler Optimizations

- By default, compilers optimize for the average processor
- When possible, enable the use of architecture-specific instruction sets using -**mtune=X** and/or -**march=X** (in gcc)
- Enable general compiler optimization using -**Ox**
- Read your compilers man-page for more details

Proebsting's Law: Compiler advances double computing power every 18 years.

# Programming Language

Consider choosing a programming language, which

- is idle-friendly
- lets you program without any further abstraction layers
- has a minimal runtime
- supports multithreading
- is fast

Languages like Fortran, C and C++ are highly recommended

# Outline

1 Introduction

2 Computational Efficiency

3 Energy-to-Solution
- Total Cost of Ownership
- Energy-to-Solution
- Adaptive Run-Time Systems

4 Energy-Awareness in Practice

# Total Cost of Ownership

- Defines the total operation costs of a computing environment like an HPC cluster
- For most applications, increasing the computational efficiency decreases the TCO
- However, some applications require solution-specific changes

# Energy-to-Solution

- Applications need to be aware of their environment
  - For HPC: Adapting CPU clock speed based on application
  - For Mobile: Respecting power states and energy saving modes to allow switching into low-power modes
- Computational efficiency is not always the optimal solution
- Multiple approaches exist to this
  - Throttling of CPU frequency and threads
  - Adaptive run-time based

# Adaptive Run-Time Systems

- Measure performance slowdown against CPU energy savings
- Evaluate $\beta$-effectiveness on (current) savings
- Adapt CPUs on an HPC cluster based on current $\beta$-effectiveness

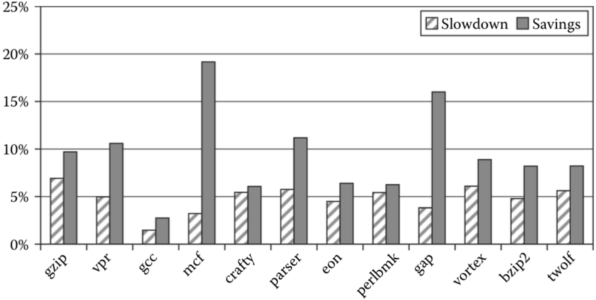



Figure: The actual performance slowdown and CPU energy savings of CPU2000 benchmarks using the presented run-time system

# Outline

1 Introduction

2 Computational Efficiency

3 Energy-to-Solution

4 Energy-Awareness in Practice
- Testing for Energy-Efficiency
- Recommendations
- Conclusion

# Testing for Energy-Efficiency

- Profile system power during application runtime
    - Understand the impact of Idle and Running states
    - Examine timer interrupts
    - Examine disk and file access
- Measure using tools like **Extrae**
- Check for cache misses and hits using e.g. **perf**
- Focus on optimizing code that is executed a lot
    - This can be checked using e.g. **gprof**

# Example: Extrae and pmlib

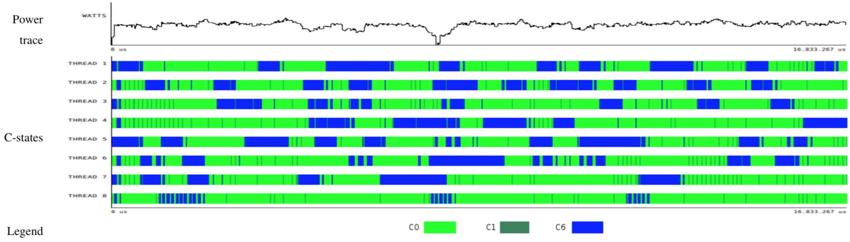



Figure: Power consumption and C-states

# Example: perf

```
perf stat -B -e cache-references,cache-misses
             -e cycles,instructions,branches sleep 5

Performance counter stats for 'sleep 5':

  10573 cache-references
   1949 cache-misses           # 18.34 % of all cache refs
1077328 cycles                 # 0.000 GHz
 715248 instructions           # 0.66 isns per cycle
 151188 branches

5.002714139 seconds time elapsed
```

# Recommendations

Practical recommendations regarding some things said in the previous slides:

- Algorithms: Do not reinvent the wheel
  - You are less likely to get it right by yourself
  - Many programming languages already come with an abstract algorithms library
- Testing: Never trust your instincts, measure instead

# Conclusion

By adopting an energy-aware approach to programming, huge energy-savings can be achieved while often also optimizing the performance at the same time.

Programmers need to be aware that even simple to implement things such as the cache-efficient use of data structures or compiler optimizations can often have a huge impact on their applications energy consumption.

# Literature I


Arndt Bode.
Energy to solution: a new mission for parallel computing.

Chandler Carruth.
Efficiency with Algorithms, Performance with Data Structures.

Wu-Chun Feng.
The Green Computing Book: Tackling Energy Efficiency at Large Scale.

Petter Larsson.
Energy-Efficient Software Guidelines.

T. A. Proebsting.
T.A. Proebsting's Law: Compiler Advances Double Computing Power Every 18 Years.

# Literature II

Rüdiger Kapitza Timo Hönig, Christopher Eibel and Wolfgang Schröder-Preikschat.
Energy-Aware Programming Utilizing the SEEP Framework and Symbolic Execution.