An Experimental Survey of Energy Management Across the Stack

Melanie Kambadur and Martha A. Kim
Columbia University, New York, NY
Importance of power and energy efficiency

Environmental sustainability

Datacenter & supercomputing expense

Battery life (wearables/IoT)

Limiting Hardware Performance
The whole stack is now “energy aware”

```c
int main() {
    cout << "Hello, world!" <<
    return 0;
}
```

- Syntax analyzer
- Semantic analyzer
- Code generator

Circuit

- Stop or slow clock
- Reduce supply voltage
- Both, i.e. DVFS
The whole stack is now “energy aware”

```c
int main() {
    cout << "Hello, world!" <<
    return 0;
}
```

**Microarchitecture**
- Multicore Accelerators

**Circuit**
- Stop or slow clock
- Reduce supply voltage
- Both, i.e. DVFS
The whole stack is now “energy aware”

Platform

- Ambient cooling
- Screen brightness

Microarchitecture

- Multicore
- Accelerators

Circuit

- Stop or slow clock
- Reduce supply voltage
- Both, i.e. DVFS

```cpp
int main() {
    cout << "Hello, world!" <<
    return 0;
}
```
The whole stack is now “energy aware”

Operating System
Energy as a resource
Scheduling & core mapping

Platform
Ambient cooling
Screen brightness

Microarchitecture
Multicore
Accelerators

Circuit
Stop or slow clock
Reduce supply voltage
Both, i.e. DVFS
The whole stack is now “energy aware”

- **Compiler/runtime**
  - Optimize for DVFS
  - Efficient instr. & data layout

- **Operating System**
  - Energy as a resource
  - Scheduling & core mapping

- **Platform**
  - Ambient cooling
  - Screen brightness

- **Microarchitecture**
  - Multicore
  - Accelerators

- **Circuit**
  - Stop or slow clock
  - Reduce supply voltage
  - Both, i.e. DVFS

```cpp
int main() {
    cout << "Hello, world!" << endl;
    return 0;
}
```
The whole stack is now “energy aware”

Applications

Power-accuracy tradeoffs
Reduce bloat

Compiler/runtime

Optimize for DVFS
Efficient instr. & data layout

Operating System

Energy as a resource
Scheduling & core mapping

Platform

Ambient cooling
Screen brightness

Microarchitecture

Multicore
Accelerators

Circuit

Stop or slow clock
Reduce supply voltage
Both, i.e. DVFS

```cpp
4 int main() {
5     cout << "Hello, world!" <<
6     return 0;
7 }
```
Most existing work is incomparable

<table>
<thead>
<tr>
<th>Project</th>
<th>Where published</th>
<th>Max energy savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnerJ: approximate data types via source annotations</td>
<td>PLDI 2011</td>
<td>50%</td>
</tr>
<tr>
<td>Pack &amp; Cap: DVFS and application mapping</td>
<td>MICRO 2011</td>
<td>51.6%</td>
</tr>
</tbody>
</table>
Most existing work is incomparable

<table>
<thead>
<tr>
<th>Project</th>
<th>Where published</th>
<th>Max energy savings</th>
<th>Benchmarks</th>
<th>Parallel?</th>
<th>Energy Accounting</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnerJ: <em>approximate data types via source annotations</em></td>
<td>PLDI 2011</td>
<td>50%</td>
<td>Java</td>
<td>No</td>
<td>Simulated, estimate instr. exec savings in processor &amp; DRAM</td>
</tr>
<tr>
<td>Pack &amp; Cap: <em>DVFS and application mapping</em></td>
<td>MICRO 2011</td>
<td>51.6%</td>
<td>C/C++</td>
<td>Up to 4 threads</td>
<td>Run on Linux/Intel, measure full system power with multimeter</td>
</tr>
</tbody>
</table>
Most existing work is incomparable

<table>
<thead>
<tr>
<th>Project</th>
<th>Where published</th>
<th>Max energy savings</th>
<th>Benchmarks</th>
<th>Parallel?</th>
<th>Energy Accounting</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EnerJ:</strong> approximate data types via source annotations</td>
<td>PLDI 2011</td>
<td>50%</td>
<td>Java</td>
<td>No</td>
<td>Simulated, estimate instr. exec. savings in processor &amp; DRAM</td>
</tr>
<tr>
<td><strong>Pack &amp; Cap:</strong> DVFS and application mapping</td>
<td>MICRO 2011</td>
<td>51.6%</td>
<td>C/C++</td>
<td>Up to 4 threads</td>
<td>Run on Linux/Intel, measure full system power with multimeter</td>
</tr>
</tbody>
</table>

No idea which saves more.
Most existing work is incomparable

<table>
<thead>
<tr>
<th>Project</th>
<th>Where published</th>
<th>Max energy savings</th>
<th>Benchmarks</th>
<th>Parallel?</th>
<th>Energy Accounting</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnerJ:</td>
<td>PLDI 2011</td>
<td>50%</td>
<td>Java</td>
<td>No</td>
<td>Simulated, estimate instr. exec savings in processor &amp; DRAM</td>
</tr>
<tr>
<td>approximate data types via source annotations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pack &amp; DVFS app mapping</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>full system power with multimeter</td>
</tr>
</tbody>
</table>

No idea which saves more.

What happens when they are combined?
A controlled comparison

This survey provides a consistent environment to enable reasonable comparisons

- 41 benchmarks from 5 suites
- gcc, Java HotSpot
- Linux Ubuntu
- Dell PowerEdge Server (no monitor)
- 2x6 core Intel Sandybridge, w/ SMT (24 HW contexts)
- 24 GB RAM
A controlled comparison

This survey provides a consistent environment to enable reasonable comparisons

- 41 benchmarks from 5 suites
  - gcc, Java HotSpot
  - Linux Ubuntu
  - Dell PowerEdge Server (no monitor)
  - 2x6 core Intel Sandybridge, w/ SMT (24 HW contexts)
  - 24 GB RAM

Measure with Intel RAPL counters, sample processor & DRAM power, energy every 50ms.
Efficiency techniques across the stack

Under the controlled system, test 9 widely-available efficiency techniques

<table>
<thead>
<tr>
<th>Application</th>
<th>Parallelism</th>
<th>Source code tuning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>Optimization sets</td>
<td>Compiled v. interpreted</td>
</tr>
<tr>
<td>O/S</td>
<td>Frequency governors</td>
<td>Over-clocking</td>
</tr>
<tr>
<td>Platform</td>
<td>Idle states</td>
<td>RAPL caps</td>
</tr>
<tr>
<td>Microarch.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Efficiency techniques across the stack

Under the controlled system, test 9 widely-available efficiency techniques, with multiple settings apiece. Look at 220 total configs, 200,000 program runs.

<table>
<thead>
<tr>
<th>Application</th>
<th>Parallelism</th>
<th>Source code tuning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>Optimization sets</td>
<td>Compiled v. interpreted</td>
</tr>
<tr>
<td>O/S</td>
<td>Frequency governors</td>
<td>Over-clocking</td>
</tr>
<tr>
<td>Platform</td>
<td>Constant frequencies</td>
<td></td>
</tr>
<tr>
<td>Microarch.</td>
<td>Idle states</td>
<td>RAPL caps</td>
</tr>
</tbody>
</table>

e.g., 1 thread, 4 threads, 16 threads.
Baseline power and performance

System default: max processor frequency (2200 MHz), Turbo Boost enabled, max compiler opts, single threaded.
Baseline power and performance

System default: max processor frequency (2200 MHz), Turbo Boost enabled, max compiler opts, single threaded.
Baseline power and performance

System default: max processor frequency (2200 MHz), Turbo Boost enabled, max compiler opts, single threaded.

24 threads of busy work

Small range of avg. power for single thread
Baseline power and performance

System default: max processor frequency (2200 MHz), Turbo Boost enabled, max compiler opts, single threaded.

24 threads of busy work
Small range of avg. power for single thread
System idle, >50% of the average app
Three values to collect per experiment

- Runtime (seconds)
- 5 seconds
Three values to collect per experiment

- Runtime (seconds): 5 seconds
- Power (Watts): 40 Watts
Three values to collect per experiment

\[
\text{Runtime (s)} \times \text{Power (W)} = \text{Energy (J)}
\]

- Runtime (s): 5 seconds
- Power (W): 40 Watts
- Energy (Joules): 200 Joules

Diagram:
- x-axis: Runtime (seconds)
- y-axis: Power (Watts)
- 40 Watts at (0, 40)
- 200 Joules at (5, 0)
One consequence of linear relationship

If power decreases less than time increases, energy increases.

Power (Watts)

Energy (Joules)

300 Joules

Runtime (seconds)

30 Watts

10 secs
Results: Linux frequency governors & overclocking

- Application:
  - Parallelism
  - Source code tuning

- Compiler:
  - Optimization sets
  - Compiled v. interpreted

- O/S:
  - Frequency governors
  - Over-clocking
  - Constant frequencies
  - Idle states
  - RAPL caps

- Platform

- Microarch.
Linux frequency governors & overclocking

Normalize to summarize 492 data points. Red X is baseline, y-axis shows relative change.
Linux frequency governors & overclocking

- Perf w/ turbo
- Max frequency + overclock
- Perf no turbo
- Max frequency no overclock
- Ondemand
- Dynamic energy efficient frequency
- Powersave
- Min frequency

x-axis shows changing feature
Linux frequency governors & overclocking

![Graph showing relative change in performance and energy efficiency across different governors and overclocking settings.]

- Perf w/turbo +overclock
- Perf no turbo + no overclock
- Ondemand
- Dynamic energy efficient frequency
- Powersave
- Min frequency

Relative Change

Runtime

Baseline

200%

150%

100%

50%

0%

250%

+114%

+ 20%

+ 7%
Linux frequency governors & overclocking

![Graph showing relative change in power and runtime for different governors and overclocking configurations.]

- Perf with turbo
  - Max frequency + overclock: +20%
- Perf without turbo
  - Max frequency no overclock: +7%
- Ondemand
  - Dynamic energy efficient frequency: -17%
- Powersave
  - Min frequency: +114%

Baseline: -31%
Linux frequency governors & overclocking

![Diagram showing relative change in runtime, power, and energy consumption for different governors and overclocking combinations.]

- **Runtime**
- **Power**
- **Energy**
- **Baseline**

**Energy**
- Perf w/ turbo: +20%
- Perf no turbo: +0%
- Ondemand: +7%
- Powersave: +47%
- Max frequency + overclock: -17%
- Max frequency no overclock: -13%
- Dynamic energy efficient frequency: -31%
# Results: Constant frequencies

<table>
<thead>
<tr>
<th>Application</th>
<th>Source code tuning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>Optimization sets</td>
</tr>
<tr>
<td>O/S</td>
<td>Compiled vs. interpreted</td>
</tr>
<tr>
<td>Platform</td>
<td>Frequency governors</td>
</tr>
<tr>
<td>Microarch.</td>
<td>Over-clocking</td>
</tr>
</tbody>
</table>

- Constant frequencies
- Idle states
- RAPL caps
Energy management approach: choose best frequency per application.
Runtime increases more than power decreases, so highest freq. is most energy efficient for every app.
Power-performance tradeoffs vary across apps.
Results: CPU Idle States

- **Application**
  - Parallelism
  - Source code tuning

- **Compiler**
  - Optimization sets
  - Compiled v. interpreted

- **O/S**
  - Frequency governors
  - Over-clocking
  - Constant frequencies

- **Platform**
  - Idle states
  - RAPL caps

- **Microarch.**
Idle states put unused cores into varying levels of ‘sleep’
Because 1/12 cores used, save 19% energy when idle is enabled.

Reduce power too.
## Results: Compiler optimization sets

<table>
<thead>
<tr>
<th>Component</th>
<th>Optimization Sets</th>
<th>Source Code Tuning</th>
<th>Compiled V. Interpreted</th>
<th>Over-clocking</th>
<th>Constant Frequencies</th>
<th>Idle States</th>
<th>RAPL Caps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Parallelism</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>Optimization sets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O/S</td>
<td>Frequency governors</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Platform</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Microarch.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Good compilation saves more energy than any strategy we’ve seen, due exclusively to runtime savings.

Compiler optimization sets (native apps)

- Runtime
- Power
- Energy
- Baseline

i.e., O3 uses 43% of O0 energy
Results: Compilation v. Interpretation

- Application
  - Parallelism
  - Source code tuning

- Compiler
  - Optimization sets
  - Compiled v. interpreted

- O/S
  - Frequency governors
  - Over-clocking
  - Constant frequencies

- Platform
  - Idle states
  - RAPL caps

- Microarch.
Energy differential larger than native: points to energy-programmer effort tradeoff (that JVM is able to correct)?

i.e., compiled uses 11% of interpreted energy
Results: Parallelism

- Application
  - Parallelism
  - Source code tuning
- Compiler
  - Optimization sets
  - Compiled v. interpreted
- O/S
  - Frequency governors
  - Over-clocking
  - Constant frequencies
- Platform
  - Idle states
  - RAPL caps
- Microarch.
1 thread per core, profile on separate thread. Note this data includes 3/5 benchmark suites.
Parallelism

Power increases due to:
1) Increased throughput
2) Fewer idle cores.

So, energy savings is all from runtime savings.

1 thread per core, profile on separate thread. Note this data includes 3/5 benchmark suites.
Results: Combining multiple techniques

- Application
  - Parallelism
  - Source code tuning
- Compiler
  - Optimization sets
  - Compiled v. interpreted
- O/S
  - Frequency governors
  - Over-clocking
  - Constant frequencies
- Platform
  - Idle states
  - RAPL caps
- Microarch.
Combining multiple techniques

Lots of data in the paper + more experiments

<table>
<thead>
<tr>
<th>Thread count</th>
<th>Idle on</th>
<th>Idle off</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-O3</td>
<td>-O2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Perf w/ turbo</td>
<td>100</td>
<td>56</td>
</tr>
<tr>
<td>Perf no turbo</td>
<td>99</td>
<td>53</td>
</tr>
<tr>
<td>Ondemand</td>
<td>96</td>
<td>52</td>
</tr>
<tr>
<td>Powersave</td>
<td>144</td>
<td>64</td>
</tr>
</tbody>
</table>

Baseline
Combining multiple techniques

Just one interesting observation:

<table>
<thead>
<tr>
<th></th>
<th>Idle off</th>
<th>Idle on</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread count</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-O3</td>
<td>-O2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Baseline</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perf w/ turbo</td>
<td>100</td>
<td>56</td>
</tr>
<tr>
<td>Perf no turbo</td>
<td>99</td>
<td>53</td>
</tr>
<tr>
<td>Ondemand</td>
<td>96</td>
<td>52</td>
</tr>
<tr>
<td>Powersave</td>
<td>144</td>
<td>64</td>
</tr>
</tbody>
</table>
Combining multiple techniques

Just one interesting observation:

<table>
<thead>
<tr>
<th>Thread count</th>
<th>-O3</th>
<th>-O2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perf w/ turbo</td>
<td>100</td>
<td>107</td>
</tr>
<tr>
<td>Perf no turbo</td>
<td>99</td>
<td>106</td>
</tr>
<tr>
<td>Ondemand</td>
<td>96</td>
<td>102</td>
</tr>
<tr>
<td>Powersave</td>
<td>144</td>
<td>153</td>
</tr>
</tbody>
</table>

Table:

<table>
<thead>
<tr>
<th>Thread count</th>
<th>-O3</th>
<th>-O2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perf w/ turbo</td>
<td>100</td>
<td>107</td>
</tr>
<tr>
<td>Perf no turbo</td>
<td>99</td>
<td>106</td>
</tr>
<tr>
<td>Ondemand</td>
<td>96</td>
<td>102</td>
</tr>
<tr>
<td>Powersave</td>
<td>144</td>
<td>153</td>
</tr>
</tbody>
</table>
Combining multiple techniques

Just one interesting observation:

<table>
<thead>
<tr>
<th>Thread count</th>
<th>-O3</th>
<th>-O2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Baseline</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perf w/ turbo</td>
<td>100</td>
<td>56</td>
</tr>
<tr>
<td>Perf no turbo</td>
<td>53</td>
<td>4</td>
</tr>
<tr>
<td>Ondemand</td>
<td>96</td>
<td>52</td>
</tr>
<tr>
<td>Powersave</td>
<td>144</td>
<td>64</td>
</tr>
</tbody>
</table>

Ondemand saves energy @ 1 thread, increases energy @ 16 threads
Summary

• Some techniques always save energy
  – Parallelism (20-60%), idle states (15-20%), compiler optimizations (60%)
Summary

• Some techniques always save energy
• Others were neutral, not consistent savers, or saved only power and not energy
  – Frequency tuning surprisingly poor performer despite literature attention
Summary

- Some techniques always save energy
- Others were neutral, not consistent savers, or saved only power and not energy
- Combining techniques can undercut or cancel energy savings
  - *Ondemand* frequency governor saves energy at 1 thread, increases energy at 16 threads.
  - Idle states save 19% at max frequency, only 10% at min. freq.
Summary

• Some techniques always save energy
• Others were neutral, not consistent savers, or saved only power and not energy
• Combining techniques can undercut or cancel energy savings
• This is a lot of data but still a small slice of the world
  – New server gens, mobile, embedded
  – Other research ideas that are hard to replicate
Summary

- Some techniques always save energy
- Others were neutral, not consistent savers, or saved only power and not energy
- Combining techniques can undercut or cancel energy savings
- This is a lot of data but still a small slice of the world
- What will we find if we keep comparing and combining?
  - Can software make a big impact on energy efficiency?
  - Do we need (some) application developers to worry about energy separately from performance?
Questions?

Our data (per application, un-normalized):
http://www.arcade.cs.columbia.edu/energy-study

Melanie Kambadur (melanie@cs.columbia.edu)

Martha Kim (martha@cs.columbia.edu)
RAPL caps constrain application to stay beneath a certain Wattage during a given interval. Save limited power, not effective energy savers.
Potential power-performance tradeoff if ‘wakeups’ are poorly timed, but in our experiments runtime improved because of increased opportunity for overclock.
Different kinds of frequency and voltage tuning

56 configs per application:
- RAPL, idle on
- RAPL, idle off
- Const. freq, idle on
- Const. freq, idle off
- System freq, idle on
- System freq, idle off

Note no normalization! Compiler at O3/VM, 16 threads, turbo on.
- Idle on generally saves energy
- Runtime varies greatly between apps, power not so much
Source code tuning

- Reduce temporary variables
- CSE
- Postpone variable declrs
- Use operator= instead of operator alone
- Use prefix instead of postfix operators for complex types
- Use direct assignments rather than inits then assignment
- Replace * & / opts with shifts or +
- Optimize loops with unrolling and unswitching
- Reduce # of function args

Ineffective, because our applications are large, run on servers, and already well-optimized.