Energy Characterization of Embedded Processors for Software Energy Optimization

Tohru Ishihara
Kyoto University

Lovic Gauthier
Kyushu University

The 11th International Forum on MPSoC and Multicore 2011/07/05
Agenda

- Introduction
- Processor Energy Characterization
- Software Energy Optimization
Motivation

- Portable systems execute power hungry applications
- Most functions are implemented by software
- Energy depends on the software running on processor systems

Software energy analysis is necessary for reducing the energy consumption of embedded systems
Energy Characterization

- Calculate the energy dissipated for a hardware event using post-layout simulation of a target processor system.
- The hardware events include data read, data write, cache miss, program branch execution and so on.

<table>
<thead>
<tr>
<th>Hardware Events</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data read</td>
<td>15 nJ</td>
</tr>
<tr>
<td>Cache miss</td>
<td>32 nJ</td>
</tr>
<tr>
<td>Branch exec.</td>
<td>58 nJ</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Our Approach

- Run a number of training benches on ISS of a target processor
- Run the same benches on a post-layout model of the processor
- Fit a linear model through regression analysis

\[ E_{GL}(1) = e_1 \cdot N_1(1) + e_2 \cdot N_2(1) + \ldots + e_n \cdot N_n(1) \]
\[ E_{GL}(2) = e_1 \cdot N_1(2) + e_2 \cdot N_2(2) + \ldots + e_n \cdot N_n(2) \]
\[ \ldots \]
\[ E_{GL}(m) = e_1 \cdot N_1(m) + e_2 \cdot N_2(m) + \ldots + e_n \cdot N_n(m) \]
Software Energy Analysis

- A linear expression for the software energy consumption
  \[ E_{\text{estimated}} = \sum e_i \cdot N_i \quad e_i : \text{energy of HW event}, \quad N_i : \# \text{HW events} \]
- The number of the hardware events should be counted by instruction set simulator (ISS)

**Energy Lookup Table**

<table>
<thead>
<tr>
<th>HW events</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data read</td>
<td>15 nJ</td>
</tr>
<tr>
<td>Cache miss</td>
<td>32 nJ</td>
</tr>
<tr>
<td>Branch exec.</td>
<td>58 nJ</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Accuracy Evaluation

- **Target system**

  ![Diagram of processor and main memory]

  **Processor**
  - CPU core
  - I-Cache
  - D-Cache

  **Main Memory**
  - SDRAM (Micron’s DDR2)

- **Processors**
  - M32R-II, SH3-DSP (Renesus)
  - MeP (Toshiba)

- 0.18μm CMOS library
Results for M32R-II & SH3-DSP

Energy estimation for JPEG encoder executed on a M32R-II processor

Energy estimation for JPEG encoder executed on a SH3-DSP processor
Multi-Performance Processor

- Based on MeP (Toshiba)
  - PEs have the same ISA but differ in their clock speeds and energy consumptions
  - Intra MPU core: a single PE core runs alternatively
  - Inter MPU cores: multiple MPU cores run concurrently

Cache associativity value can be dynamically changed

Results for MPP

Energy estimation for JPEG encoder run on a MPP with 1.8V 4-w cache:

- Estimation error = 2.3%

Energy estimation for JPEG encoder run on a MPP with 1.0V 1-w cache:

- Estimation error = 3.2%

< 10 sec

9.5 hours
Applications

- Software Energy Analysis
  - Help finding energy bottleneck
- Software Energy Optimization
  - Compiler optimization
  - OS-based power management
The number of instructions executed

Energy dissipated for 1000 instructions [µJ]

- D-main_mem
- D-SPM
- I-cache
- I-SPM
- I-main_mem
- cache miss
- CPU core
- Post Layout

MPEG2 encoder

Energy for cache access is much larger than that for cache misses

Off-chip data access

Cache miss

CPU

Inst. cache access
Energy dissipated for 1000 instructions [$\mu$J]

More than 50% is stack accesses

Cache miss

Off-chip read

word access

JPEG encoder

Inst. cache access

CPU

The number of instructions executed
Software Energy Optimization

Motivation:

- Memory consumes a large amount of energy
- Memory energy depends on program behavior
- Code optimization contributes to the total energy reduction

Power Analysis of ARM920T

- On-chip memory: 43%
- Off-chip memory: 10x of on-chip memory energy
Code and Data Allocation

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>Latency</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPM</td>
<td>Small</td>
<td>Small</td>
<td>Small</td>
</tr>
<tr>
<td>Cache</td>
<td>Small</td>
<td>Small</td>
<td>Large</td>
</tr>
<tr>
<td>Off-chip mem.</td>
<td>Huge</td>
<td>Huge</td>
<td>Huge</td>
</tr>
</tbody>
</table>

Our Compiler Optimization Techniques

Find the optimal locations of functions and data objects in a memory address space.

Application program

- Function A
- Function B
- Global variable C
- Global variable D
- Constant data E
- ...
Our Approach

Previous work…
Find code & data placements which
• maximize # SPM accesses
• minimize # cache misses
• minimize # off-chip accesses
Do not always minimize the energy

Our method…
Minimize the total energy consumption estimated by our model through an ILP


Memory area
- Scratchpad region
- Cacheable region
- Non-cacheable region

Application program
- Function A
- Function B
- Global variable C
- Global variable D
- Constant data E

Total processor energy reduced by 10%
Stack Allocation

- Optimization is done at compile time through an ILP
  - Find the best locations of stack frames based on profiling
  - Store/Load inserted before and after call instructions
- Frames are dynamically moved between SPM and MM
  - Stack frame is generated in SPM when the function is called
  - Store frames into MM if there is no space left in the SPM
Stack Allocation Results

- Circular: Evict oldest frame first if there is no space left in the SPM
- Static: Place frames in the SPM only if SPM does not overflow
- Ours: Placement is optimized by the Integer Linear Programming

Multi-Task System

- The SPM is shared among tasks
- Several previous techniques
  - (a) Spatial sharing
    - ✔ No management required
    - ✔ Very small part of the SPM for each task
  - (b) Temporal sharing
    - ✔ Totally of SPM space to each task
    - ✔ SPM update at context switches
  - (c) Hybrid
    - ✔ Best of both approaches
    - ✔ Compile time profile-based assignment to SPM

SPM Sharing for Multi-Task

- At compile time:
  - Assign an SPM space (i.e., block) to each task
  - Find memory objects to place in each block
  - Find an address for each block in SPM for minimizing overlaps

- At run time:
  - Copy only a part of overlapping with coming task
Multi-Task Results

Summary

- A fast accurate model for SW energy consumption
- The error of our approach is 5% on an average and 20% at the maximum case
- Code and data placement techniques drastically reduce the SW energy

Future work

- Refine stack placement technique and target heap object (WIP)