CS 516 Compilers and Programming Languages II

Compiler Optimizations for Power / Energy / Thermal Introduction
Announcements

- Sakai site is up, together with piazza. Please check whether you have access
- First homework has been posted. Please check your email for your machine assignment
- Office hours: Friday 2:00-3:00pm or by appointment
Power vs. Energy

**power** ($P$): activity level at a given point in time (watts)

**energy** ($E$): total amount of activity (joules)

\[ E = \int P(t) dt \quad \text{general} \]

\[ E = P \times t \quad \text{constant } P \text{ over time} \]

same energy, different (peak) power

**optimizing for (peak) power == optimizing for energy?**

**ANSWER:** Not necessarily! Example: re-schedule activities
Performance (execution time)

Performance is determined by the length of the critical path of a computation.

Sample task graph

Tasks that are independent may be executed in parallel

(task needs \(k\) cycles)

dependence
Performance is determined by the length of the critical path of a computation.

Sample task graph

Tasks that are independent may be executed in parallel.

Critical path has 17 cycles.
Some important lessons to learn:

You can run, but you cannot hide
- pushing instructions onto the non-critical execution path; (“hiding”) does not necessarily reduce energy/power
- higher threshold for profitability of speculation

You cannot beat hardware (need for heterogeneous architectures)
- if an operation is implemented in hardware, and an applications needs it, that’s the best you can do (e.g.: floating-point unit, )
- need to be able to disable hardware if not in use (dark silicon)

Keep the overall picture in mind
- performance is measured for the entire program
- power/energy should also be measured for the entire system, in addition to optimized system component(s)

There is no free lunch
- you typically cannot avoid tradeoffs
- tradeoffs are application specific and/or user specific
Dynamic Frequency and Voltage Scaling (DFVS)
- power (P) and energy (E) are proportional to $C \cdot V_{dd}^2 \cdot \text{freq}$
- reducing freq alone reduces P, but not E
- many architectures support sets of $(V_{dd}, \text{freq})$ pairs
- CMOS technologies have fixed threshold voltages $V_{\text{threshold}}$
  $$V_{\text{threshold}} < V_{dd},$$ i.e., limit on how low $V_{dd}$ can go
- leakage current ($I_{\text{leakage}}$) problem (“static power”):
  - increases with heat and when $V_{dd}$ gets closer to $V_{\text{threshold}}$

Dynamic Resource Configuration/Hibernation
- system components are “switched off” or “put to sleep”
- many architectures support several sleep/hibernation states/levels
- the deeper the “sleep”, the more energy you save, but the cost of “waking up” becomes higher (time, power, and energy)

Remote Task Execution
- use of remote services (let someone else do the work for you)
- cost (communication) < cost (doing it yourself)

QoR Optimizations (Approximation Redundancy)
- may save resources by reducing quality outcomes
Who may perform these optimizations?

- **User / application programmer**
  - introduce new programming language constructs / abstractions
  - crucial if semantic of program is affected
  - provide hints/insights that do not affect semantics

- **Compiler**
  - may perform (static) full-program analyses (execution context)
  - may reshape program behavior
  - low runtime overhead

- **Operating System (OS)**
  - adaptable to runtime behavior (execution context)
  - optimization in multi-programming environments
  - limited execution contexts of single application

- **Hardware**
  - fine-grain optimizations
  - low overhead, but also least execution context

- **Combination of techniques from levels above**
Examples of hybrid static/dynamic approaches

- **Trace-based compilation**
  - derive knowledge about program behaviors through profiling of executions with representative inputs
  - use this information (e.g. model construction) to make better optimization decisions
  - advantage: no “run-time” overhead; disadvantage: need for representative inputs and increased “off-line” overhead

- **Dynamic compilation / JIT compilers (just-in-time)**
  - compile parts of the program (or entire program) at program invocation/execution time
  - example: convert interpreted code to native code for “hot paths” in program execution
  - advantage: knowledge about dynamic behavior that can be exploited through optimization; disadvantage: runtime overhead and potentially additional needed hardware features (e.g. translation caches)
**Goal:** Reduce the energy needed for executing an application with a *soft* execution time **deadline** constraint.

Power and Energy are proportional to $C V^2 f$.

**Safety:** always safe – assume settings supported by hardware

**Opportunity:** CPU idle time (CPU DVFS)

**Profitability:** up to 55% energy savings with up to 6% performance penalties on SPECfp95 on 600MHz - 1.2 GHz AMD Athlon4
Scale voltage and frequency to save energy and still meet deadline

(a) original schedule.

(b) voltage scaled schedule.

(c) power-performance tradeoffs.
Opportunity: Unbalanced Program Regions

Assumes architecture that allows overlapping of CPU and memory activities (e.g.: non-blocking loads)

- cpuBusy
- bothBusy
- memBusy

Memory bound

Memory bound -> balanced
**Goal:** Assign minimal voltages and frequencies to different program regions such that overall performance is only slightly decreased (~ 1%).

**Opportunity:**

Program regions with unbalanced computation and memory requirements.

Architectures that allow overlap of computations and data accesses.
Compiler Algorithm Outline

1) Identify single entry/exit program regions as scheduling candidates (sequences of loop nests, procedure calls, if-statements); enumerate and evaluate all such (forward) sequences

2) Performance modeling
   - determine cpuBusy, memBusy, bothBusy of scheduling candidates;
   - determine relative execution times of scheduling candidates
   - use results to compute slowdown factor $\delta$
     (CPU slow-down) under a soft deadline constraint (e.g.: $\leq 1\%$ performance penalty), and select single best candidate

3) Generate voltage/frequency scheduling instructions and adjust performance optimizations.
Basic Performance Model

\[ T = \text{cpuBusy} + \text{memBusy} + \text{bothBusy} \]

\[ T_{\text{new}}(\delta) = \delta \times \text{cpuBusy} + \max \left( \begin{array} {c} \text{bothBusy} + \text{memBusy} \\ \delta \times \text{bothBusy} \end{array} \right) \]

Constraints on choosing \( \delta \):

1. \((\delta - 1) \times \text{cpuBusy} \leq 1\%
2. \(1 \leq \delta \leq 1 + \frac{\text{memBusy}}{\text{bothBusy}}\)
3. memory latency is divisible by \( \delta \)
- SimpleScalar with memory hierarchy extensions
  - cycle accurate simulation
  - out-of-order superscalar processors
  - branch prediction and speculative execution

- Simulated out-of-order target architecture:
  - 1 cycle L1 cache, non-blocking
  - 10 cycles L2 cache, non-blocking
  - 100 cycles memory, blocking
  - instruction window size = 64
  - instruction issue width = 4 per cycle

- Switching overhead of 10,000 cycles
C Program

- SUIF2 passes
- Instrumented C program

DFVS'ed C program

- SUIF2 passes
- Profile

SUIF2 passes

- gcc
- Simplescalar Simulator
  - reduced ref.in
  - train.in
  - -O3 -funroll-loops

Results

- gcc
  - Simplescalar Simulator
  - -O3 -funroll-loops
## Benefit Analysis (ref.in)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>by hand + ref.in</th>
<th>compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Slow-down</td>
<td>Exec. Time</td>
</tr>
<tr>
<td>swim95</td>
<td>2.02</td>
<td>101.68%</td>
</tr>
<tr>
<td>tomatv95</td>
<td>2.44</td>
<td>101.99%</td>
</tr>
<tr>
<td>applu</td>
<td>1.58</td>
<td>101.82%</td>
</tr>
<tr>
<td>hydro2d</td>
<td>1.33</td>
<td>101.47%</td>
</tr>
</tbody>
</table>

- **compiler** - considers DFVS overheads
  - enumerates all possible regions
  - automates the process
  - uses different input for training

- Soft deadline: 1%
- Single region
- Up to 1 GHz
- Scaling cost=10µs
R=20.31%, f=500MHz
R=20.73%, f=490MHz

by hand

- compiler
R=26.05%, f=500MHz
R=16.44%, f=600MHz

- by hand
- compiler
So far so good, but what if a highly optimizing compiler will eliminate all opportunities for DVFS?

Remember: We need to evaluate new optimizations in the context of all previous optimizations and their impact on the code shape.
Highly optimizing compiler: DEC’s f90 compiler -O5
Alpha 21264-like processor, ref.in input data set
modified Wattch (SimpleScalar based) simulator, $10^6$ cycles summaries
10% soft deadline performance penalty, no switching costs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CPU Energy</th>
<th>Performance Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>swim95</td>
<td>38.86%</td>
<td>9.50%</td>
</tr>
<tr>
<td>tomcatv95</td>
<td>28.30%</td>
<td>8.42%</td>
</tr>
<tr>
<td>applu</td>
<td>51.35%</td>
<td>7.27%</td>
</tr>
<tr>
<td>hydro2d</td>
<td>48.73%</td>
<td>7.32%</td>
</tr>
</tbody>
</table>
hydro2d

slowdown factor

million cycles
Runtime based strategies may have a hard time here due to overheads.

compiler $\delta = 1.8$ million cycles
## Two Commercial DFVS Processors

<table>
<thead>
<tr>
<th>Performance level</th>
<th>Compaq Presario 715US Mobile Athlon 4</th>
<th>Fujitsu LifeBook P2040 Crusoe TM5800</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F(MHz)</td>
<td>V(volts)</td>
</tr>
<tr>
<td>1</td>
<td>600</td>
<td>1.15</td>
</tr>
<tr>
<td>2</td>
<td>700</td>
<td>1.20</td>
</tr>
<tr>
<td>3</td>
<td>800</td>
<td>1.25</td>
</tr>
<tr>
<td>4</td>
<td>900</td>
<td>1.30</td>
</tr>
<tr>
<td>5</td>
<td>1000</td>
<td>1.35</td>
</tr>
<tr>
<td>6</td>
<td>1100</td>
<td>1.40</td>
</tr>
<tr>
<td>7</td>
<td>1200</td>
<td>1.45</td>
</tr>
</tbody>
</table>
**EEL\textsubscript{dfvs}**: 7 discrete performance levels, actual measurements

Linux 2.4.18, compiler: *g77 -O2*,

Mobile Athlon 4 processor

*Physical measurement using power meter*

5% soft performance deadline

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CPU Energy</th>
<th>Performance Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>swim95</td>
<td>44.35%</td>
<td>1.33%</td>
</tr>
<tr>
<td>tomcatv95</td>
<td>46.30%</td>
<td>2.70%</td>
</tr>
<tr>
<td>applu</td>
<td>84.34%</td>
<td>2.23%</td>
</tr>
<tr>
<td>hydro2d</td>
<td>64.32%</td>
<td>3.48%</td>
</tr>
</tbody>
</table>
R=20.31%, f=500MHz, p=1%
R=20.73%, f=490MHz, p=1%
R=100.0%, f=700MHz, p=5%

- by hand
- simulated
- mobile Athlon 4
R=26.05%, f=500MHz, p=1%
R=16.44%, f=600MHz, p=1%
R=95.30%, f=700MHz, p=5%

- by hand
- simulated
- mobile Athlon 4
R = 19.90%, f = 750MHz, p = 1%
R = 18.29%, f = 810MHz, p = 1%
R = 62.97%, f = 1000MHz, p = 5%

- by hand
- simulated
- mobile Athon 4

- expansion edge
R=26.67%, f=750MHz, p=1%
R=84.04%, f=910MHz, p=1%
R=96.93%, f=900MHz, p=5%

- expansion edge
- by hand
- simulated
- mobile Athlon 4
Comparison of execution time with different performance levels of Crusoe TM5800 and mobile Athlon 4.

- Crusoe TM5800, CPU bound (artificial benchmark)
- Mobile Athlon 4
Energy Delay Product Comparison

Overall system physical measurements
Comparison relative to Compaq Presario at peak (1.2GHz)
LifeBook uses Transmeta’s LongRun DVFS technology
Linux 2.4.18, g77 -O2

Energy Delay Product

\[ E \times T = P \times T^2 \]

Presario 715US:
33.6W - 57.3W

LifeBook P2040:
13.0W - 15.9W
Energy Delay Product Comparison

Overall system physical measurements
Comparison relative to Compaq Presario at peak (1.2GHz)
LifeBook uses Transmeta’s LongRun DVFS technology
Linux 2.4.18, g77 -O2

Energy Delay Product

\[ E \times T = P \times T^2 \]

Presario 715US:
33.6W - 57.3W

LifeBook P2040:
13.0W - 15.9W

- Our DVFS compiler algorithm makes the “high performance” laptop even more energy efficient than a low-power laptop with built-in power optimization
- For compute intensive applications, the low-power laptop is much less energy efficient
- For some applications, cannot beat the energy efficiency of the low-power laptop