CS 516 Compilers and Programming Languages II

Compiler Optimizations for Power / Energy / Thermal DVFS 1
• 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)

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

**Answer:** Not necessarily! Example: re-schedule activities

\[ E = \int P(t)dt \]  

**general**

\[ E = P \times t \]  

**constant P over 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.
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 application 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 $C V_{dd}^2 \cdot \text{freq}$
  - switching capacitance $C$, supply voltage $V_{dd}$, frequency (freq)
  - reducing freq alone reduces P, but not E
  - many architectures support sets of ($V_{dd}$, 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**
Who may perform these optimizations?

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

- cpuBusy
- bothBusy
- memBusy
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.
## 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><code>F(MHz)</code></td>
<td><code>V(volts)</code></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>
Basic Performance Model

\[ T = cpu\text{Busy} + mem\text{Busy} + both\text{Busy} \]

\[ T_{new}(\delta) = \delta \cdot cpu\text{Busy} + \max \left( \frac{both\text{Busy} + mem\text{Busy}}{\delta \cdot both\text{busy}} \right) \]

Constraints on choosing \( \delta \):

(1) \( (\delta - 1) \cdot cpu\text{Busy} \leq 1\% \)

(2) \( 1 \leq \delta \leq 1 + \frac{mem\text{Busy}}{both\text{Busy}} \)

(3) memory latency is divisible by \( \delta \)
Continue discussion of compile-time DVFS.

Please look at the two listed papers.