CS 516 Compilers and Programming Languages II

Compiler Optimizations for Power / Energy / Thermal DVFS-2
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.
**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
Opportunity: Unbalanced Program Regions

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

CPU speed

MEM speed

memory bound

CPU speed

MEM speed

memory bound -> balanced

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>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>
Basic Performance Model

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

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

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 \( \delta \)

See: [PACS00]
(3) memory latency is divisible by $\delta$

float $A[n][n]$, accu
for (j=0; j<n; j++)
  for (i=0; i<n; i++)
    accu += $A[i][j]$;

--- 1% performance degradation

- no loop optimizations, no data locality
- loop interchange, loop unrolling, software prefetching, ...

See: [PACS00]
Why Models?

Needed to support optimization decisions, i.e., needed to choose among a set of optimization alternatives.

This means that models do not have to be exact in absolute terms, but in relative terms to rank the optimization alternatives according to their effectiveness / outcome quality.

Models are typically used to predict program behaviors or determine optimization results based on a finite set of observations (parameter values), where observations may be static or dynamic:

\[ \text{model: } (p_1 \times p_2 \times \ldots p_n) \rightarrow \text{result} \]

This mapping can be constructed through different techniques, including linear interpolation, training (machine learning) or through exhaustive enumeration if the observation space is “small”. Latter involves storing results in a (finite) table, where model applications (mappings) are implemented as table look-ups.
- 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
Simulators vs. Hardware Testbeds

- **Simulators** are programs, i.e., software: Easier/cheaper to “make” and change, which is not easy to do for actual hardware

- **Simulators** implement a particular execution model, with full control over the execution environment.
  - **Advantage**
    - Total control over what happens, so no “semantic gap”
    - Can model hardware architectures/features that do not exist yet
  - **Disadvantage**
    - If you implement a bad model, your results will not reflect “reality”
      - e.g.: failure to model interactions among components
    - Need to validate simulators (may sometimes not be possible)

- **Actual hardware testbeds**
  - **Advantage**
    - the real thing: measure behaviors in absolute terms
    - no “bad” models because there are no models
  - **Disadvantage**
    - you may not understand why you get particular measurements (semantic gap)
    - hardware may not exist yet, so nothing to measure
C Program

SUIF2 passes

Instrumented C program

SUIF2 passes

DFVS'ed C program

gcc

Simplescalar Simulator

-O3 -funroll-loops

Profile

gcc

Simplescalar Simulator

Results

reduced ref.in

train.in

-O3 -funroll-loops

See: [LCPC’01]
## Benefit Analysis (ref.in)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>by hand + ref.in</th>
<th></th>
<th>compiler</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Slow-down</td>
<td>Exec. Time</td>
<td>CPU Energy</td>
<td>Slow-down</td>
</tr>
<tr>
<td>swim95</td>
<td>2.02</td>
<td>101.68%</td>
<td>76.79%</td>
<td>2.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>102.67%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>75.70%</td>
</tr>
<tr>
<td>tomcatv95</td>
<td>2.44</td>
<td>101.99%</td>
<td>76.25%</td>
<td>1.69</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100.47%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>83.49%</td>
</tr>
<tr>
<td>applu</td>
<td>1.58</td>
<td>101.82%</td>
<td>90.43%</td>
<td>1.24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>101.22%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>93.94%</td>
</tr>
<tr>
<td>hydro2d</td>
<td>1.33</td>
<td>101.47%</td>
<td>84.61%</td>
<td>1.11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>101.69%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>83.42%</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
By hand

- compiler

R = 20.31%, f = 500MHz
R = 20.73%, f = 490MHz
Lecture 6

- by hand
- compiler

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

call
loop
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>
A graph showing the relationship between slowdown factor and million cycles.
Runtime based strategies may have a hard time here due to overheads.

compiler \( \delta = 1.8 \)  

million cycles
A C program is compiled with SUIF2 passes, resulting in an instrumented C program. This instrumented program is then compiled with the g77 compiler using the -O2 optimization flag, generating machine code. The resulting disassembled code is profiled to identify bottlenecks. The instrumented code is then compiled with g77 using the -O3 optimization flag and the -funroll-loops flag. The machine-train.in is used for training, and the results are analyzed.

See: [PLDI'03]
EEL_{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>
START

- \( R = 20.31\% \), \( f = 500\text{MHz} \), \( p = 1\% \)
- \( R = 20.73\% \), \( f = 490\text{MHz} \), \( p = 1\% \)
- \( R = 100.0\% \), \( f = 700\text{MHz} \), \( p = 5\% \)

END

- by hand
- simulated
- mobile Athlon 4

Lecture 6
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 for different performance levels:

- **Crusoe TM5800**, CPU bound (artificial benchmark)
- **mobile Athlon 4**

Graph shows a decrease in execution time as performance level increases.
Energy Delay Product

E * T = P * T^2

Presario 715US: 33.6W - 57.3W
LifeBook P2040: 13.0W - 15.9W

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
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

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

- 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
• There is significant opportunity for slowing down the CPU and thereby reducing power dissipation and energy without incurring significant runtime overheads

• Is the compiler the right “level” to exploit this, or is an OS / runtime system approach better?
  - Infrequent changes among “stable” regions: past behavior can be used to predict future behavior; compiler advantage: can insert calls to change (freq, voltage) just in time, potentially reducing overhead of voltage/frequency change (note: change up/down may not be symmetric in terms of cost)
  - Frequent changes among short regions (e.g.: hydro2d): compiler can “look into the future” and smooth things out; OS/runtime systems may also be able to do this if some control-theoretical approach is used

• Multi-programming environments with high (freq, voltage) switching overheads need OS management across multiple programs
Resource hibernation.

Please see listed readings.