Software Low-Power Techniques

Naehyuck Chang
Dept. of EECS/CSE
Seoul National University
naehyuck@snu.ac.kr
Contents

- Role of software and power sources
  - Role of software in embedded systems
  - Power sources
- Software power estimation
  - Gate/architecture-level power analysis
  - Instruction-level power analysis
  - Bus switching analysis
- Software power optimizations
  - Algorithm transformation
  - Minimizing memory access cost
  - Instruction selection and scheduling
  - Source-level coding style
  - Source-level code transformation
- Low-power software considerations
- Summary
Why software for power?

- Recent electronic devices: processor-based system

<table>
<thead>
<tr>
<th>Application program</th>
<th>Algorithm implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating system</td>
<td>Actual energy consumer</td>
</tr>
<tr>
<td>Hardware</td>
<td></td>
</tr>
</tbody>
</table>

Software (Application program and OS) control
the behavior of actual energy consumer
Design effort

- Importance of the software design
Contents

• Role of software and power sources
  • Role of software in embedded systems
  • Power sources
• Software power estimation
  • Gate/Architecture-level power analysis
  • Instruction-level power analysis
  • Bus switching analysis
• Software power optimizations
  • Algorithm transformation
  • Minimizing memory access cost
  • Instruction selection and scheduling
  • Source-level coding style
  • Source-level code transformation
• Low-power software considerations
• Summary
Power sources (software perspective)

- **CPU**
  - For the execution of a given program code
  - Need to minimize the program code execution cycles
  - Need to minimize unnecessary activity in the CPU and memory

- **System bus**
  - To communicate with other peripherals
  - Need to minimize the bus switching activities
  - On-chip and off-chip buses

- **Memory**
  - Usually important but more important for data intensive applications
  - Need to reduce the memory access by data reuse or increase the cache hit
  - Need to reduce the memory access cost by proper memory organization
  - Aggregate the memory transaction workload for better power down
How to attack the power sources? (I)

- Active mode
  - CPU
    - Computation cost reduction (mainly for arithmetic operations)
    - In general, the faster the program code is, the lower is the power (the primary reason that most software low-power design is not surprising)
  - System bus
    - Reduce the instruction bus switching activities (op-code sequences)
      - Questionable with I cache
    - Reduce the address bus switching activities (instruction sequences as well as data sequences)
    - Also can be incorporated with hardware improvement (bus encoding scheme)
  - Memory
    - Power-conscious custom memory hierarchy
      - Including the appropriate use of memory type/size and so on
    - Program should exploit the benefit of memory hierarchy by reuse
    - In general, D cache makes the problem difficult
How to attack the power sources? (II)

- Stand-by mode
  - CPU
    - Shut down and reduce the idle power
    - Need a shut down policy
      - Levels of shut down
    - OS-level policy implementation is widely accepted
      - Example: ACPI (Advanced Configuration of Power Interface)
  - System bus
    - Not a major concern
      - Strongly dependent on the bus architecture
  - Memory
    - Shut down and reduce the idle power
Bus switching activity analysis

- Requires knowledge of bus architecture, op-codes, data statistics, and mapping information for address space
- Simulation-based approach is widely used
  - Sequence of op-codes, addresses, data values on busses
  - Compute the switching statistics
    - Can be utilized for power reduction
- Power reduction techniques
  - Cold scheduling for instruction scheduling
  - Considers for data values in DSP applications
- Limitations
  - Cache memory filters out and randomize the bus access sequences
Contents

- Role of software and power sources
  - Role of software in embedded systems
  - Power sources
- Software power estimation
  - Gate/Architecture-level power analysis
  - Instruction-level power analysis
  - Bus switching analysis
- **Software power optimizations**
  - Algorithm transformation
  - Minimizing memory access cost
  - Instruction selection and scheduling
  - Source-level coding style
  - Source-level code transformation
- Low-power software considerations
- Summary
Software power optimizations (I)

- Reduction methods
  - Selection of least expensive instructions or instruction sequences
  - Minimizing the frequency or cost of memory accesses
    - Need to investigate the memory energy model: memory-aware DVS by ELPL
  - Exploiting power minimization features of hardware
    - Such as MTCMOS, Multiple supply voltage technique, and so on

- Optimization procedure
  - Design an algorithm that maps well to available hardware
  - Write a code that maximizes the performance
  - Minimize the energy consumption
Software power optimizations (II)

- Optimization strategy depends on the target applications
  - Battery-powered systems
    - Lengthen the battery lifetime
  - Heat or reliability conscious systems
    - Reduce the instantaneous or average power dissipation
  - Instantaneous power dissipation constraint
    - Hardware approach is preferred
      - Easy (Lower voltage, lower clock rate)
    - Software approach
      - May increase the energy consumption by increasing the # of execution cycles of a program (Inserting NOPs)
      - Fetch throttling
Algorithm transformation

- First, let’s look at the computational resources
- Match the degree of parallelism in an algorithm to the # of parallel resources available
- Major application area
  - Parallel processor applications
  - Low-power DSP synthesis (also exploits parallelism)

- Increasing parallelism will decrease the execution time
  - Reduce the supply voltage and clock rate ⇒ DVS
- Decreasing parallelism will use less operational units
  - Remove operational units or
  - Reduce the leakage power by shutdown ⇒ DPM
Trade-off resource usage and execution

- One well-known technique
  - Reduction operations
- Example: summation

**Single adder**

\[ \text{sum} = 0 \]
\[ \text{for } i = 0 \text{ to } n \]
\[ \text{sum} = \text{sum} + A(i) \]

**Two adders**

\[ \text{sum} = 0 \]
\[ \text{for } i = 0 \text{ to } n/2 - 1 \]
\[ \text{tmp} = A(2i) + A(2i+1) \]
\[ \text{sum} = \text{sum} + \text{tmp} \]

**Four adders**

\[ \text{sum} = 0 \]
\[ \text{for } i = 0 \text{ to } n/4 - 1 \]
\[ \text{tmp1} = A(4i) + A(4i+1) \]
\[ \text{tmp2} = A(4i+2) + A(4i+3) \]
\[ \text{tmp} = \text{tmp1} + \text{tmp2} \]
\[ \text{sum} = \text{sum} + \text{tmp} \]
Contents

- Role of software and power sources
  - Role of software in embedded systems
  - Power sources
- Software power estimation
  - Gate/Architecture-level power analysis
  - Instruction-level power analysis
  - Bus switching analysis
- Software power optimizations
  - Algorithm transformation
  - Minimizing memory access cost
  - Instruction selection and scheduling
  - Source-level coding style
  - Source-level code transformation
- Low-power software considerations
- Summary
Minimizing memory access costs (I)

- Improving memory system efficiency is usually helpful in reducing the power or energy dissipation
- Largely due to memory being both a power and a performance bottleneck
- Classification of typical techniques
  - Minimize the # of memory accesses required by an algorithm
  - Minimize the total memory required by an algorithm
  - Put memory accesses as close as possible to the processor (register $\rightarrow$ cache $\rightarrow$ external memory)
  - Make the most efficient use of the available memory bandwidth (e.g. Load multiple word parallel loads)
Minimizing memory access costs (II)

- Target code section
  - Deeply nested loops for data processing
  - Deeply nested $\Rightarrow$ high frequency
  - Data processing $\Rightarrow$ load and store

- Traditional high-level compilers works well
  - Loop fusion
  - Loop fission
  - Loop interchange
  - Loop tiling (blocking)
Loop transformation techniques (I)

- Loop fusion
  - Merge two loops to reduce the memory accesses

```plaintext
For i = 1 TO N DO
  B[i] = f(A[i])
For i = 1 TO N DO
  C[i] = g(B[i])
ENDFOR
```

- B[i] is reused
- Instruction cache should be large enough to have both function f and g
Loop transformation techniques (II)

- **Loop fission**
  - Break a loop into several pieces

  ```
  For i = 1 TO N DO
    B[i] = C[i-1] * X + C
    C[i] = 1/B[i]
    D[i] = sqrt(C[i])
  ENDFOR
  ```

  - When the instruction cache is small

  ```
  For ib = 0 TO N-1 DO
    B[ib+1] = C[ib] * X + C
    C[ib+1] = 1/B[ib+1]
  ENDFOR
  ```

  ```
  For ib = 0 TO N-1 DO
  ENDFOR
  ```

  ```
  For ib = 0 TO N-1 DO
    D[ib+1] = sqrt(C[ib+1])
  ENDFOR
  ```

  ```
  i = N+1
  ```
Loop transformation techniques (III)

- **Loop interchange**
  - Change the order of loops
  - Fortran: column-wise data ordering

  ```fortran
  do i = 1,n
    do j = 1,n
      total[i] = total[i] + a[i,j]
    end do
  end do
  ```

  ```fortran
  do j = 1,n
    do i = i,n
      total[i] = total[i] + a[i,j]
    end do
  end do
  ```

  - **Two consecutive array accesses**
    - Original code: a[i][j] followed by a[i][j+1] ⇒ separated by n+1
    - Transformed code: a[i][j] followed by a[i+1][j] ⇒ separated by 1
    - Improves the spatial locality
Loop transformation techniques (IV)

- Loop tiling (blocking)
  - Localize the array accesses for cache reuse

```plaintext
do i=1, n
  do j=1, n
    a[i,j] = b[j,i]
  end do
end do
```

- \((nxn)\) array access \(\Rightarrow (n/64)^2 \times (64 \times 64)\) array access

```plaintext
do TI=1, n, 64
  do TJ=1, n, 64
    do i=TI, min(TI+63, n)
      do j=TJ, min(TJ+63, n)
        a[i,j] = b[j,i]
      end do
    end do
  end do
end do
```

Seoul National University
Loop transformation techniques (V)

- Loop swapping
  - Swap the loop blocks to reduce the lifetime of large arrays
  - Some physical memory locations can be easily shared

```
For i = 1 TO N DO
  B[i] = f(A[i])
For i = 1 TO N DO
  D = g(C[i], D)
```

```
For i = 1 TO N DO
  D = g(C[i], D)
For i = 1 TO N DO
  B[i] = f(A[i])
```

- If C is not used afterward in the original code...
- Lifetime of C
  - Original code: until the end of the second loop
  - Transformed code: until the end of the first loop
    - Array B and C can share the same physical location
Fitting an algorithm to an architecture

- Vector quantization of video data by Lidsky and Raeby
  - 4 x 4 pixel region of an image $\Rightarrow$ quantized by 256-element codebook of precomputed vectors
  - Linear search $\Rightarrow$ binary search
    - $O(N) \Rightarrow O(\log N)$

- Good for a fast pipelined implementation
- Clock rate is reduced by a factor of 8
- Memory accesses: reduced by a factor of 30
- Power: scaled down by a factor of 17
Parallel loads (I)

- May not be feasible for general purpose or DSP processor
- One distinct feature of such processors
  - Parallel loads (or dual loads)
- Dual loads
  - Effective for energy reduction, but not necessarily for power
  - 47% energy saving shown by Tiwari
Parallel loads (II)

- To utilize the parallel loads
  - Optimize the memory allocation of a program to maximize opportunities for using dual loads
  - Combine memory accesses as much as possible

```
<table>
<thead>
<tr>
<th>Case (a)</th>
<th>Case (b)</th>
<th>Case (c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM A</td>
<td>x, y, z</td>
<td>x, y</td>
</tr>
<tr>
<td>RAM B</td>
<td>x, y, z</td>
<td>x, z</td>
</tr>
<tr>
<td>LOAD B ← x</td>
<td>DLOAD B ← x, A ← z</td>
<td></td>
</tr>
<tr>
<td>LOAD C ← y</td>
<td>LOAD C ← y</td>
<td></td>
</tr>
<tr>
<td>LOAD A ← z;</td>
<td>MULT B ← B, C</td>
<td></td>
</tr>
<tr>
<td>MULT B ← B, C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD A ← A, B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Energy (pJ)</td>
<td>10.57</td>
<td>9.32</td>
</tr>
<tr>
<td>Power (μW)</td>
<td>105.7</td>
<td>93.2</td>
</tr>
</tbody>
</table>
```
Other memory related techniques

- According to Davison’s paper
  - Register allocation
    - Minimize the external memory allocation
  - Cache blocking
    - Loop tiling mentioned earlier
  - Register blocking
    - Similar to loop tiling, redundant loading is eliminated
  - Recurrence detection and optimization
    - Use of registers for values that are carried over from one level of recursion to the next
  - Compact multiple memory references into a single reference
Memory access coalescing

- By Davidson
- Combine loop unrolling and memory reference compaction
  - Unrolling: transforms several iterations of a loop into a block of straight line code
    - Minimize the loop overhead
    - May increase instruction cache misses
  - Memory reference compaction
    - Coalescing narrow loads and stores into a wide load and store
  - By combining two techniques, higher chances for coalescing
Unified register/memory bank allocation (I)

- By Sudarsanam
- After instruction selection phase
- Procedure
  - For each basic block, combine instructions
    - Either two memory operations or one memory and one ALU operation
    - W/o changing the order of operations
  - Constraint graph generation from the compacted code
    - Vertex: symbolic register or variable
    - Edge: an allocation constraint on the pair of vertices with color code
      - Weight: penalty when the constraint is violated
  - Label each vertex by simulated annealing
Memory bank assignment (I)

- By Lee and Tiwari
- Formulated as a graph-partitioning problem
- Solved by simulated annealing
- Cost function
  - # of execution cycles required to accomplish all of the memory transfers indicated by the graph
- Graph
  - Vertex: variable
  - Edge: there exists an ALU operation requiring two variables as arguments
- After partition
  - If an edge crosses partition, possible to compact two memory transfers into a dual-load operation
  - If not, two memory operations are performed sequentially
Memory bank assignment (II)

- Example code
  
  ```
  ADD R_1 ← a, e  
  ADD R_2 ← e, b  
  ADD R_3 ← a, b  
  ADD R_4 ← b, d  
  ADD R_5 ← a, c  
  ADD R_6 ← c, d  
  ```

- Total cost
  - e(a,e) = 2
  - All others = 1
  - Total = 7

- Corresponding access graph

- Partitioned access graph
Contents

• Role of software and power sources
  • Role of software in embedded systems
  • Power sources
• Software power estimation
  • Gate/Architecture-level power analysis
  • Instruction-level power analysis
  • Bus switching analysis
• Software power optimizations
  • Algorithm transformation
  • Minimizing memory access cost
  • Instruction selection and scheduling
  • Source-level coding style
  • Source-level code transformation
• Low-power software considerations
• Summary
Instruction selection and scheduling

• What we have appreciated up to this point
  • Many code size optimization techniques (for speed and code size) should be equally applicable to power or energy optimization
  • Difference
    • Objective function used in code generation is different
• Energy and performance optimization are going to the same direction
• But their improvement ratios may be different
• Code size minimization: not directly linked
  • But good to improve cache performance ⇒ energy saving
Instruction packing (I)

- A feature of Fujitsu DSP called Elixir (and others)
- Pack a memory instruction and ALU instruction into one
  - Much of the cost of each instruction is overhead that is not duplicated when operations are executed in parallel
    - By sharing some resources
    - About 50% energy reduction
- Similar benefits for parallel loads
- All these are special cases of VLIW and superscalar architecture
  - Supporting parallel executions
Instruction packing (II)

• Tested instructions

<table>
<thead>
<tr>
<th>class</th>
<th>addressing</th>
<th>functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDI</td>
<td>immed → reg</td>
<td>load immediate data to a register</td>
</tr>
<tr>
<td>LAB</td>
<td>mem1 → reg A, mem2 → reg B</td>
<td>transfer memory data to registers A, B</td>
</tr>
<tr>
<td>MOV1</td>
<td>reg1 → reg2</td>
<td>move data from one register to another</td>
</tr>
<tr>
<td>MOV2</td>
<td>mem → reg, or</td>
<td></td>
</tr>
<tr>
<td></td>
<td>reg → mem</td>
<td>move data from memory to a register, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>from a register to memory</td>
</tr>
<tr>
<td>ASL</td>
<td>reg specified implicitly</td>
<td>add/sub, shift, logic operations in ALU</td>
</tr>
<tr>
<td>MAC</td>
<td>reg specified implicitly</td>
<td>multiply and accumulate in ALU</td>
</tr>
</tbody>
</table>

• Datapath
Instruction packing (III)

- Experimental results

<table>
<thead>
<tr>
<th>LDI</th>
<th>LAB</th>
<th>MOV1</th>
<th>MOV2</th>
<th>ASL</th>
<th>MAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>range</td>
<td>15.8 - 22.9</td>
<td>34.6 - 38.5</td>
<td>18.8 - 20.7</td>
<td>17.6 - 19.2</td>
<td>15.8 - 17.2</td>
</tr>
<tr>
<td>average base</td>
<td>19.4</td>
<td>36.5</td>
<td>19.8</td>
<td>18.4</td>
<td>16.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instruction</th>
<th>ASL:LAB</th>
<th>ASL:MOV1</th>
<th>ASL:MOV2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range</td>
<td>34.5 - 38.7</td>
<td>15.7 - 17.4</td>
<td>18.7 - 20.4</td>
</tr>
<tr>
<td>average base</td>
<td>36.6</td>
<td>16.6</td>
<td>19.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instruction</th>
<th>MAC:LAB</th>
<th>MAC:MOV1</th>
<th>MAC:MOV2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range</td>
<td>33.9 - 39.9</td>
<td>15.9 - 18.9</td>
<td>19.0 - 21.2</td>
</tr>
<tr>
<td>average base</td>
<td>36.9</td>
<td>17.4</td>
<td>20.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>adjacent instructions</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDI, ASL:LAB</td>
<td>23.8</td>
</tr>
<tr>
<td>LDI, ASL:MOV1</td>
<td>19.6</td>
</tr>
<tr>
<td>LDI, ASL:MOV2</td>
<td>20.0</td>
</tr>
<tr>
<td>MOV1, ASL:LAB</td>
<td>14.7</td>
</tr>
<tr>
<td>MOV1, ASL:MOV2</td>
<td>27.1</td>
</tr>
<tr>
<td>MOV2, ASL:LAB</td>
<td>27.1</td>
</tr>
<tr>
<td>MOV2, ASL:MOV2</td>
<td>19.3</td>
</tr>
<tr>
<td>MAC, ASL:LAB</td>
<td>27.9</td>
</tr>
<tr>
<td>ASL:LAB, ASL:MOV1</td>
<td>29.3</td>
</tr>
<tr>
<td>ASL:MOV2, ASL:MOV2</td>
<td>18.7</td>
</tr>
</tbody>
</table>

Base cost reduction

Overhead cost reduction
Instruction ordering

- Mostly concerns on the inter-instruction effects
- Cost function
  - Hamming distance
    - By Su, Tsui, Despain
    - By Tomiyama, Ishihara, Inoue, Yasuura
  - Measured data
    - By Tiwari, Malik, Wolfe
- Less effect than the minimization of program execution cycles
- More effective in DSP systems rather than GPP
Cold scheduling (I)

- By Su, Tsui, Despain
- Binary coded addressing $\Rightarrow$ Gray coded addressing
  - Assume that instructions are serially executed in most cases
  - The same to data access when large arrays are used
- Reorder instructions to reduce inter-instruction effects on instruction bus
  - Consider instruction op-codes
  - Inter-instruction cost is op-code Hamming distance
  - Use list scheduler where priority criterion is tied to Hamming distance
Cold scheduling (II)

- Binary code $\Leftrightarrow$ Gray code conversion

\[
B = \langle b_{n-1}, b_{n-2}, \ldots, b_1, b_0 \rangle \\
G = \langle g_{n-1}, g_{n-2}, \ldots, g_1, g_0 \rangle.
\]

**Binary code $\rightarrow$ Gray code**

\[
g_n = b_n \\
g_i = b_{i+1} \oplus b_i \quad (i = n-1, 0)
\]

**Gray code $\rightarrow$ Binary code**

\[
b_n = g_n \\
b_i = b_{i+1} \oplus g_i \quad (i = n-1, 0)
\]
Cold scheduling (III)

- Example using Hamming distance for op-codes

- \( ( ) 11111111 \)
- \( (A) 10010011 \)
- \( (B) 11101000 \)
- \( (C) 10111011 \)
- \( (D) 01110100 \)
- \( ( ) 11111111 \)

Total 24 switchings

- \( ( ) 11111111 \)
- \( (A) 10010011 \)
- \( (C) 10111011 \)
- \( (B) 11101000 \)
- \( (D) 01110100 \)
- \( ( ) 11111111 \)

Total 18 switchings
Cold scheduling (IV)

- Impact of Gray coding

Bit switches of instruction address (37% reduction)

Bit switches of data address (8% reduction)

* BPI: Bit switches Per executed instruction
Cold scheduling (V)

- Overall saving

- 20 ~30% reduction in switching activities
Contents

- Role of software and power sources
  - Role of software in embedded systems
  - Power sources
- Software power estimation
  - Gate/Architecture-level power analysis
  - Instruction-level power analysis
  - Bus switching analysis
- Software power optimizations
  - Algorithm transformation
  - Minimizing memory access cost
  - Instruction selection and scheduling
  - Source-level coding style
  - Source-level code transformation
- Low-power software considerations
- Summary
Coding styles

- Use processor-specific instruction style:
  - Variable types
  - Function calls style
  - Conditionalized instructions (for ARM)
- Follow general guidelines for software coding
  - Use table look-up instead of conditionals
  - Make local copies of global variables so that they can be assigned to registers
- Avoid multiple memory look-ups with pointer chains
Example: ARM variable types

- Default “int” variable type is 18.2% more energy efficient than “char” or “short”
- Sign or zero extending is needed for shorter variable types

```c
char charinc (char a)
{
    return a + 1;
}

int wordinc (int a)
{
    return a + 1;
}
```
Example: ARM conditional execution

- All ARM instructions are conditional
- Conditional execution reduces the number of branches
- Grouped conditions should be used instead of separate if statements since they help the compiler conditionalize instructions
- 1.25% of energy reduction

```c
int g(int a, int b, int c, int d) {
    if (a > 0 && b > 0 && c < 0 && d < 0)
        return a + b + c + d;
    return -1;
}
```
Example: Switch versus Table Lookup

- Table lookup is 52.3% more energy efficient than a dense switch statement since fewer jumps are required

```c
switch(condition) {
    case 0: return "EQ";
    case 1: return "NE";
    case 2: return "CS";
    case 3: return "CC";
    default: return 0;
}
```

```c
if ((unsigned) condition >= 4 )
    return 0;
return "EQ\ONE\0CS\0CC\0" + 3*condition;
```
Example: MPEG

- Compiler optimizations account for only 0.6% energy savings
- Source code optimizations give as much as 32.3% energy savings

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Balance</td>
<td>none</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>none</td>
<td>Code Size</td>
<td>none</td>
<td>0</td>
<td>-0.6</td>
<td>-0.6</td>
</tr>
<tr>
<td>none</td>
<td>Time</td>
<td>none</td>
<td>0</td>
<td>-0.2</td>
<td>-0.2</td>
</tr>
<tr>
<td>none</td>
<td>Balance</td>
<td>all</td>
<td>0</td>
<td>-0.6</td>
<td>-0.6</td>
</tr>
<tr>
<td>all</td>
<td>Balance</td>
<td>none</td>
<td>-5.8</td>
<td>-35</td>
<td>-32.3</td>
</tr>
<tr>
<td>all</td>
<td>Balance</td>
<td>all</td>
<td>-5.8</td>
<td>-35</td>
<td>-32.3</td>
</tr>
</tbody>
</table>

Compilation options
- Code size / Time / Balance

Cross-jump
- Load multiple
  - Common sub-expression elimination
Contents

• Role of software and power sources
  • Role of software in embedded systems
  • Power sources
• Software power estimation
  • Gate/Architecture-level power analysis
  • Instruction-level power analysis
  • Bus switching analysis
• Software power optimizations
  • Algorithm transformation
  • Minimizing memory access cost
  • Instruction selection and scheduling
  • Source-level coding style
  • Source-level code transformation
• Low-power software considerations
• Summary
Source code transformation

- Minimize power-consuming activity

- Computation
  
  \[
  A \times B + A \times C \quad \rightarrow \quad A \times (B + C)
  \]

- Communication
  
  \[
  \begin{align*}
  &\text{receive}(A) \\
  &\text{for } (c = 1..N) \\
  &\quad B = c \times A
  \end{align*}
  \]
  \[
  \begin{align*}
  &\text{receive}(A) \\
  &\text{for } (c = 1..N) \\
  &\quad B = c \times A
  \end{align*}
  \]

- Storage
  
  \[
  \begin{align*}
  &\text{for } (c = 1..N) \\
  &\quad B[c] = A[c] \times D[c]
  \end{align*}
  \]
  \[
  \begin{align*}
  &\text{for } (c = 1..N) \\
  &\quad F[c] = A[c] \times D[c] - 1
  \end{align*}
  \]
Automating source-code transformations

- **Objective**
  - Achieve lower-energy and faster code in average

- **Methodology**
  - Profiling for values and frequencies
  - Code specialization

- Transformations are mainly independent of target architecture
Challenges

- Cost of communication dominates
  - Explicit communication
    - Message passing
    - Easier to handle
  - Implicit communication
    - Shared memory
    - Easier to program, harder to optimize

- Application software optimization for energy and performance requires awareness of energy cost of storage and communication
Contents

- Role of software and power sources
  - Role of software in embedded systems
  - Power sources
- Software power estimation
  - Gate/Architecture-level power analysis
  - Instruction-level power analysis
  - Bus switching analysis
- Software power optimizations
  - Algorithm transformation
  - Minimizing memory access cost
  - Instruction selection and scheduling
  - Source-level coding style
  - Source-level code transformation
- Low-power software considerations
- Summary
# Memory system considerations

<table>
<thead>
<tr>
<th>Memory System Feature</th>
<th>Low-Power Software Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total memory size</td>
<td>Minimizing total memory requirements through code compaction and algorithm transformations allows for a smaller memory system with lower capacitances and improved performance. However, this benefit might have to be traded off against a faster algorithm that requires more memory.</td>
</tr>
<tr>
<td>Partitioning into banks; how many, how big</td>
<td>Needed if parallel loads are to be used. The size of application, data structures, and access patterns should determine this.</td>
</tr>
<tr>
<td>Wide data bus</td>
<td>Needed for parallel loads.</td>
</tr>
<tr>
<td>Proximity to CPU</td>
<td>Memory close to CPU reduces capacitances; makes memory use less expensive.</td>
</tr>
<tr>
<td>Cache size</td>
<td>Should be sized to minimize cache misses for the intended application. Software should apply techniques such as cache blocking to maintain high degree of spatial and temporal locality.</td>
</tr>
<tr>
<td>Cache protocol</td>
<td>If an application’s memory access patterns are well defined, the protocol can be optimized.</td>
</tr>
<tr>
<td>Cache locking</td>
<td>If significant portions of an application can run entirely from cache, this can be used to prevent any external memory accesses while those portions of code are executing.</td>
</tr>
</tbody>
</table>
## Architectural considerations

<table>
<thead>
<tr>
<th>Architectural Feature</th>
<th>Low-Power Software Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class of processor</td>
<td>Application dependent. How well does the instruction set fit the application? Is there hardware support for operations common to the application? Does the hardware implement functions not needed by the application?</td>
</tr>
<tr>
<td>DSP/RISC/CISC</td>
<td></td>
</tr>
<tr>
<td>Parallel processing</td>
<td>Does the level and type of parallelism in an algorithm fit one of these architectures? If so, the parallel architecture can greatly improve performance and allow reduced voltages and clock rates to be used.</td>
</tr>
<tr>
<td>VLIW, Superscalar</td>
<td></td>
</tr>
<tr>
<td>SIMD, MIMD. How much parallelism?</td>
<td></td>
</tr>
<tr>
<td>Bus architecture</td>
<td>Separate buses (e.g., address, data, instruction, I/O) can make it easier to optimize instruction sequences, addressing patterns, and data correlations to minimize bus switching.</td>
</tr>
<tr>
<td>Register file size</td>
<td>Increased register count eases register allocation and reduces memory accesses, but too large of a register file can make register accesses as expensive as a cache access.</td>
</tr>
</tbody>
</table>
# Power management considerations

<table>
<thead>
<tr>
<th>Feature</th>
<th>Low-Power Software Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software vs. hardware control</td>
<td>Software control allows power management to be tailored to the application at the cost of added execution cycles.</td>
</tr>
<tr>
<td>Clock/voltage removal:</td>
<td>Some kind of shutdown modes are needed in order to realize a benefit from minimized execution times.</td>
</tr>
<tr>
<td>At what level of granularity?</td>
<td></td>
</tr>
<tr>
<td>Clock/voltage reduction</td>
<td>If there is a schedule slack associated with a software task, reduce power by slowing down the clock and reducing the supply voltage.</td>
</tr>
<tr>
<td>Guarded evaluation</td>
<td>Latch the inputs of functional units to prevent meaningless calculations when outputs of units are not being used. This increases the power reduction from reduction in strength and operand swapping.</td>
</tr>
</tbody>
</table>
Summary

- High possibility to optimize energy in software design
- CPU, bus and memory are major power consumers
- Performance and energy improvements are going to the same direction
- Traditional optimization techniques can be used by replacing the cost function
- Source-level optimization may pay off more than compiler back-end optimization
References