ESE680-002 (ESE534): Computer Organization

Day 24: April 11, 2007
Specialization

Previously

- How to support bit processing operations
- How to compose any task
- Instantaneous << potential computation

Today

- What bit operations do I need to perform?
- Specialization
  - Binding Time
  - Specialization Time Models
  - Specialization Benefits
  - Expression

Quote

- The fastest instructions you can execute, are the ones you don’t.

Idea

- **Goal**: Minimize computation must perform
- Instantaneous computing requirements less than general case
- Some data known or predictable
  - compute minimum computational residue
- As know more data \(\rightarrow\) reduce computation
- Dual of **generalization** we saw for local control

Know More \(\rightarrow\) Less Compute
Typical Optimization

- Once know another piece of information about a computation
  (data value, parameter, usage limit)

- Fold into computation
  producing smaller computational residue

Opportunity Exists

- Spatial unfolding of computation
  – can afford more specificity of operation
  – *E.g.* last assignment (FIR,IIR)

- Fold (early) bound data into problem

- Common/exceptional cases

Opportunity

- Arises for programmables
  – can change their *instantaneous* implementation

  – don’t have to cover all cases with a single configuration

  – can be heavily specialized
  - while still capable of solving entire problem
    - (all problems, all cases)
Opportunity

- With bit level control
  - larger space of optimization than word level
- While true for both spatial and temporal programmables
  - bigger effect/benefits for spatial

Multiply Example

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Feature</th>
<th>Area and Time</th>
<th>16-16</th>
<th>8-8</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom 16-16</td>
<td>0.85/μm</td>
<td>1.9M, 46 ns</td>
<td>9.6</td>
<td>9.6</td>
<td>1.0</td>
</tr>
<tr>
<td>Custom 8-8</td>
<td>0.85/μm</td>
<td>3.3M, 4.3 ns</td>
<td>9.6</td>
<td>9.6</td>
<td>1.0</td>
</tr>
<tr>
<td>SRAM 5-5</td>
<td>3.76/μm</td>
<td>1.25M, 40 ns</td>
<td>1.3</td>
<td>1.3</td>
<td>1.3</td>
</tr>
<tr>
<td>SRAM 5-8</td>
<td>5.05/μm</td>
<td>13.6M, 13.1 ns</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Multiply Show

- Specialization in datapath width
- Specialization in data

Benefits

Empirical Examples

- UART
- Pattern match
- Less than
- Multiply revisited
  - more than just constant propagation
- ATR

UART

- I8251 Intel (PC) standard UART
- Many operating modes
  - bits
  - parity
  - sync/async
- Run in same mode for length of connection
UART FSMs

<table>
<thead>
<tr>
<th>FSM</th>
<th>Fully Generic</th>
<th>Area Mapped</th>
<th>Specialized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Speed Mapped</td>
<td>CLBs</td>
<td>path</td>
</tr>
<tr>
<td>8251 processor (10)</td>
<td>11</td>
<td>3.5</td>
<td>5</td>
</tr>
<tr>
<td>8251 transmitter</td>
<td>10.5</td>
<td>4.5</td>
<td>57.5</td>
</tr>
<tr>
<td></td>
<td>Asynchronous, parity</td>
<td>28</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>1 sync char, parity</td>
<td>35</td>
<td>3.5</td>
</tr>
<tr>
<td>8251 receiver</td>
<td>32</td>
<td>5.5</td>
<td>52.5</td>
</tr>
<tr>
<td></td>
<td>Asynchronous, parity</td>
<td>29</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>Asynchronous, no parity</td>
<td>28</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>External sync, parity</td>
<td>31</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>Internal, 2 sync chars, parity</td>
<td>31</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>Internal, 1 sync char, parity</td>
<td>31</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>Internal, 1 sync char, no parity</td>
<td>31</td>
<td>4.5</td>
</tr>
</tbody>
</table>

UART Composite

<table>
<thead>
<tr>
<th>design</th>
<th>Fully Generic</th>
<th>Area Mapped</th>
<th>Specialized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Speed Mapped</td>
<td>CLBs</td>
<td>path</td>
</tr>
<tr>
<td>8251 core</td>
<td>358.5</td>
<td>8.5</td>
<td>348.5</td>
</tr>
<tr>
<td>Async, 64 clks/b</td>
<td>201</td>
<td>6</td>
<td>141.5</td>
</tr>
<tr>
<td>1 clks/b, 8n1</td>
<td>201</td>
<td>6</td>
<td>141.5</td>
</tr>
<tr>
<td>Sync, internal, 2 sync, 8o</td>
<td>201</td>
<td>6</td>
<td>141.5</td>
</tr>
<tr>
<td>Sync, external, 5n</td>
<td>201</td>
<td>6</td>
<td>141.5</td>
</tr>
</tbody>
</table>

Pattern Match

- Savings:
  - 2N bit input computation \( \rightarrow N \)
  - if \( N \) variable, maybe trim unneeded portion
  - state elements store target
  - control load target

Pattern Match

<table>
<thead>
<tr>
<th>(size)</th>
<th>CLBs</th>
<th>path</th>
<th>CLBs</th>
<th>path</th>
<th>AT Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = b )</td>
<td>( b ) variable</td>
<td>( b ) constant</td>
<td>w/state</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(8)</td>
<td>2.5 (+4)</td>
<td>2</td>
<td>1.5</td>
<td>2</td>
<td>0.60</td>
</tr>
<tr>
<td>(16)</td>
<td>5.5 (+8)</td>
<td>3</td>
<td>2.5</td>
<td>2</td>
<td>0.30</td>
</tr>
<tr>
<td>(32)</td>
<td>10.5 (+16)</td>
<td>3</td>
<td>5.5</td>
<td>3</td>
<td>0.52</td>
</tr>
<tr>
<td>(64)</td>
<td>21.5 (+32)</td>
<td>4</td>
<td>10.5</td>
<td>3</td>
<td>0.37</td>
</tr>
</tbody>
</table>

Less Than (Bounds check?)

- Area depend on target value
- But all targets less than generic comparison

Multiply (revisited)

- Specialization can be more than constant propagation
- Naive,
  - save product term generation
  - complexity number of 1's in constant input
- Can do better exploiting algebraic properties
Multiply

- Never really need more than \( \lfloor N/2 \rfloor \) one bits in constant
- If more than \( N/2 \) ones:
  - invert \( c \) \( (2^{N+1}-1-c) \)
  - multiply by \( x \) \( (2^{N+1}-1-c)x \)
  - add \( x \) \( (2^{N+1}-1-c)x \)
  - subtract from \( (2^{N+1})x \) \( = cx \)

Multiply

- At most \( \lfloor N/2 \rfloor + 2 \) adds for any constant
- Exploiting common subexpressions can do better:
  - e.g.
    - \( c=10101010 \)
    - \( t_1=x+x<<2 \)
    - \( t_2=t_1<<5+t_1<<1 \)

Example: ATR

- Automatic Target Recognition
  - need to score image for a number of different patterns
    - different views of tanks, missiles, etc.
  - reduce target image to a binary template with don’t cares
  - need to track many (e.g. 70-100) templates for each image region
  - templates themselves are sparse
    - small fraction of care pixels

Example: UCLA ATR

- UCLA
  - specialize to template
  - ignore don’t care pixels
  - only build adder tree to care pixels
  - exploit common subexpressions
  - get 10 templates in a XC4010

\[ \text{[Villasenor et. al./FCCM'96]} \]
Example: FIR Filtering

\[ Y_n = w_1 x_1 + w_2 x_2 + \ldots + \ldots \]

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Feature Size (\mu m)</th>
<th>( T )</th>
<th>( P )</th>
</tr>
</thead>
<tbody>
<tr>
<td>32b RISC</td>
<td>0.75</td>
<td>0.020</td>
<td></td>
</tr>
<tr>
<td>16b DSP</td>
<td>0.65</td>
<td>0.057</td>
<td></td>
</tr>
<tr>
<td>32b RISC/USIP</td>
<td>0.25</td>
<td>0.021</td>
<td></td>
</tr>
<tr>
<td>64b RISC</td>
<td>0.18</td>
<td>0.064</td>
<td></td>
</tr>
<tr>
<td>FPGA (XC4K)</td>
<td>0.50</td>
<td>1.9</td>
<td></td>
</tr>
<tr>
<td>Altera 8K</td>
<td>0.30</td>
<td>3.6</td>
<td></td>
</tr>
<tr>
<td>Full Custom</td>
<td>0.75</td>
<td>3.6</td>
<td></td>
</tr>
<tr>
<td>(fixed coefficient)</td>
<td>0.60</td>
<td>2.4</td>
<td></td>
</tr>
<tr>
<td>(n.b. 16b samples)</td>
<td>.60</td>
<td>56</td>
<td></td>
</tr>
</tbody>
</table>

Usage Classes

**Specialization Usage Classes**

- Known binding time
- Dynamic binding, persistent use
  - apparent
  - empirical
- Common case

**Known Binding Time**

- \( \text{Sum} = 0 \)
- For \( i = 0 \rightarrow N \)
  - For \( i = 0 \rightarrow N \)
    - \( \text{Sum} + = V[I] \)
  - For \( i = 0 \rightarrow N \)
    - \( V[I] = V[I] / \text{Sum} \)

**Dynamic Binding Time**

- \( cexp = 0; \)
- For \( i = 0 \rightarrow V.\text{length} \)
  - if \( (V[I].\text{exp} = cexp) \)
    - \( cexp = V[I].\text{exp} \)
    - \( V[I].\text{mant} << cexp \)
  - \( \text{Thread 1:} \)
    - \( a = \text{src.read}() \)
    - if (a.newavg())
      - \( \text{avg} = a.\text{avg}() \)
  - \( \text{Thread 2:} \)
    - \( v = \text{data.read}() \)
    - \( \text{out.write}(v.\text{avg}) \)

**Empirical Binding**

- Have to check if value changed
  - Checking value \( O(N) \) area [pattern match]
  - Interesting because computations
    - can be \( O(2^N) \) [Day 9]
  - Often greater area than pattern match
- Also Rent’s Rule:
  - Computation > linear in IO
  - \( IO = C \cdot n^p \rightarrow n \propto IO^{1/p} \)
Common/Uncommon Case

• For i = 0 → N
  – If (V[i] == 10)
    • SumSq += V[i] \* V[i];
  – elseif (V[i] < 10)
    • SumSq += V[i] \* V[i];
  – else
    • SumSq += V[i] \* V[i];

Exploitation Patterns

• Full Specialization (Partial Evaluation)
  – May have to run (synth?) p&r at runtime
• Worst-case footprint
  – e.g. multiplier worst-case, avg., this case
• Constructive Instance Generator
• Range specialization (wide-word datapath)
  – data width
• Template
  – e.g. pattern match – only fillin LUT prog.

Opportunity Example

Bit Constancy Lattice

• binding time for bits of variables
  (storage-based)

<table>
<thead>
<tr>
<th>Constancy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBD</td>
<td>\ldots Constant between definitions</td>
</tr>
<tr>
<td>CSSI</td>
<td>\ldots Constant in some scope invocations</td>
</tr>
<tr>
<td>CESI</td>
<td>\ldots Constant in each scope invocation</td>
</tr>
<tr>
<td>CASI</td>
<td>\ldots Constant across scope invocations</td>
</tr>
<tr>
<td>CAPI</td>
<td>\ldots Constant across program invocations</td>
</tr>
<tr>
<td>const</td>
<td>\ldots declared const</td>
</tr>
</tbody>
</table>

[Experiment: Eylon Caspi/UCB]

Binding Times

• Pre-fabrication
• Application/algorithm selection
• Compilation
• Installation
• Program startup (load time)
• Instantiation (new ...)
• Epochs
• Procedure
• Loop

Experiments

• Applications:
  – UCLA MediaBench:
    adpcm, epic, g721, gsm, jpeg, mesa, mpeg2
    (not shown today - ghostscript, pegwit, pgp, rasta)
  – gzip, versatility, SPECint95 (parts)
• Compiler optimize \rightarrow instrument for profiling \rightarrow run
• analyze variable usage, ignore heap
  – heap-reads typically 0-10% of all bit-reads
  – 90-10 rule (variables) - ~90% of bit reads in
1-20% or bits

[Experiment: Eylon Caspi/UCB]
Empirical Bit-Reads Classification

Bit-Reads Classification
- regular across programs
  - SCASI, CASI, CBD stddev ~11%
- nearly no activity in variables declared const
- ~65% in constant + signed bits
  - trivially exploited

Constant Bit-Ranges
- 32b data paths are too wide
- 55% of all bit-reads are to sign-bits
- most CASI reads clustered in bit-ranges (10% of 11%)
- CASI+SCASI reads (50%) are positioned:
  - 2% low-order
  - 8% whole-word
  - constant
  - 39% high-order
  - 1% elsewhere

Issue Roundup

Expression Patterns
- Generators
- Instantiation/Immutable computations
  - (disallow mutation once created)
- Special methods (only allow mutation with)
- Data Flow (binding time apparent)
- Control Flow
  - (explicitly separate common/uncommon case)
- Empirical discovery

Benefits
- Much of the benefits come from reduced area
  - reduced area
    - room for more spatial operation
    - maybe less interconnect delay
- Fully exploiting, full specialization
  - don’t know how big a block is until see values
  - dynamic resource scheduling
Optimization Prospects

• Area-Time Tradeoff
  \[ T_{\text{spcl}} = T_{\text{sc}} + T_{\text{load}} \]
  \[ AT_{\text{gen}} = A_{\text{gen}} \times T_{\text{gen}} \]
  \[ AT_{\text{spcl}} = A_{\text{spcl}} \times (T_{\text{sc}} + T_{\text{load}}) \]

• If compute long enough
  \[ T_{\text{sc}} >> T_{\text{load}} \rightarrow \text{amortize out load} \]

Storage

• Will have to store configurations somewhere
  • LUT \sim 1M\lambda^2
  • Configuration 64+ bits
    – SRAM: 80K\lambda^2 (12-13 for parity)
    – Dense DRAM: 6.4K\lambda^2 (160 for parity)

Saving Instruction Storage

• Cache common, rest on alternate media
  – e.g. disk
• Compressed Descriptions
• Algorithmically composed descriptions
  – good for regular datapaths
  – think Kolmogorov complexity
• Compute values, fill in template
• Run-time configuration generation

Open

• How much opportunity exists in a given program?
• Can we measure entropy of programs?
  – How constant/predictable is the data compute on?
  – Maximum potential benefit if exploit?
  – Measure efficiency of architecture/implementation like measure efficiency of compressor?

Big Ideas

• Programmable advantage
  – Minimize work by specializing to instantaneous computing requirements
• Savings depends on functional complexity
  – but can be substantial for large blocks
  – close gap with custom?

Big Ideas

• Several models of structure
  – slow changing/early bound data, common case
• Several models of exploitation
  – template, range, bounds, full special