CS184a: Computer Architecture (Structure and Organization)

Day 23: March 7, 2005
Specialization

Previously

• How to support bit processing operations
• How to compose any task

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 Size (n)</th>
<th>Area and Time</th>
<th>16-bit</th>
<th>16-byte</th>
<th>8-bit</th>
<th>8-byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom 5×16</td>
<td>0.60 um</td>
<td>3.3 MHz, 40 ns</td>
<td>9.0</td>
<td>9.0</td>
<td>9.0</td>
<td>9.0</td>
</tr>
<tr>
<td>Custom 8×16</td>
<td>0.60 um</td>
<td>3.3 MHz, 4.3 ns</td>
<td>7.0</td>
<td>7.0</td>
<td>7.0</td>
<td>7.0</td>
</tr>
<tr>
<td>Golb-Array 5×16</td>
<td>0.75 um</td>
<td>1.5 MHz, 80 ns</td>
<td>1.3</td>
<td>1.3</td>
<td>1.3</td>
<td>1.3</td>
</tr>
<tr>
<td>FPGA</td>
<td>0.70 um</td>
<td>3.4 MHz, 120 ns</td>
<td>9.07</td>
<td>9.07</td>
<td>9.07</td>
<td>9.07</td>
</tr>
<tr>
<td>16-bit ROM</td>
<td>0.80 um</td>
<td>1.5 MHz, 107 ns</td>
<td>5.4</td>
<td>5.4</td>
<td>5.4</td>
<td>5.4</td>
</tr>
<tr>
<td>32-bit ROM</td>
<td>0.10 um</td>
<td>1.5 MHz, 157 ns</td>
<td>5.4</td>
<td>5.4</td>
<td>5.4</td>
<td>5.4</td>
</tr>
</tbody>
</table>

Benefits

Empirical Examples

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

Multiply Show

- Specialization in datapath width
- Specialization in data

Benefit Examples

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

<table>
<thead>
<tr>
<th>FSM</th>
<th>Fully Generic</th>
<th>Specialized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Speed Mapped</td>
<td>Area Mapped</td>
</tr>
<tr>
<td></td>
<td>CLBs path</td>
<td>CLBs path</td>
</tr>
<tr>
<td>I8251 proc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I8251 trm1</td>
<td>6.5</td>
<td>3.5</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5.5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>2.5</td>
</tr>
<tr>
<td>I8251 rcvr</td>
<td>24</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>37</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>5.5</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>4</td>
</tr>
</tbody>
</table>

### UART Composite

<table>
<thead>
<tr>
<th>design</th>
<th>Fully Generic</th>
<th>Specialized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Speed Mapped</td>
<td>Area Mapped</td>
</tr>
<tr>
<td></td>
<td>CLBs</td>
<td>path</td>
</tr>
<tr>
<td>I8251 core</td>
<td>358.5</td>
<td>8.5</td>
</tr>
<tr>
<td>Async, 16 cklks/bit, 8n1</td>
<td>216.5</td>
<td>7</td>
</tr>
<tr>
<td>Async, 1 cklks/bit, 5n1</td>
<td>201</td>
<td>6</td>
</tr>
<tr>
<td>Sync, internal, 2 sync, 8n</td>
<td>141.5</td>
<td>4.5</td>
</tr>
<tr>
<td>Sync, external, 5n</td>
<td>136</td>
<td>5.5</td>
</tr>
</tbody>
</table>

### Pattern Match

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

### 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 \( c \) \( (2^{N+1}-1-c) \)
  - (less than \( N/2 \) ones)
  - 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.
    - \( c=10101010 \)
    - \( t1=x+x<<2 \)
    - \( t2=t1<<5+t1<<1 \)

Example: ATR

- Automatic Target Recognition
  - need to score image for a number of different patterns
    - different views of tanks, missles, 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

[Villasenor et. al./FCCM'96]
Example: FIR Filtering

\[ Y_i = w_1 x_i + w_2 x_{i+1} + \ldots \]

Application metric:
TAPs = filter taps
multiply accumulate

\[ Y_i = w_1 x_i + w_2 x_{i+1} + \ldots \]

Usage Classes

**Specialization Usage Classes**
- Known binding time
- Dynamic binding, persistent use
  - apparent
  - empirical
- Common case

**Known Binding Time**
- \( \text{Sum} = 0 \)
- For \( i = 0 \rightarrow V.\text{length} \)
  - if \( V[i].\text{exp} \neq cexp \)
    - \( cexp = V[i].\text{exp} \)
    - \( \text{Vres}[i] = V[i].\text{mant} \ll \text{cexp} \)
  - For \( i = 0 \rightarrow V.\text{length} \)
    - \( \text{Vres}[i] = V[i]/\text{Sum} \)

**Dynamic Binding Time**
- \( \text{cexp} = 0 \)
- For \( i = 0 \rightarrow V.\text{length} \)
  - if \( (V[i].\text{exp} = \text{cexp}) \)
    - \( \text{cexp} = V[i].\text{exp} \)
    - \( \text{avg} = a.\text{avg}() \)
  - Thread 1:
    - \( a = \text{src}.\text{read}() \)
    - \( \text{avg} = a.\text{avg}() \)
  - Thread 2:
    - \( v = \text{data}.\text{read}() \)
    - \( \text{out}.\text{write}(v/\text{avg}) \)

**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
    - \( \text{IO} = C n^p \rightarrow n \propto \text{IO}^{1/p} \)
Common/Exceptional Case

- For I=0→N
  - Sum+=V[I]
  - delta=V[I]-V[I-1]
  - SumSq+=V[I]*V[I]
  - ... if (overflow)
    - ...

- For IB=0→N/B
  - For II= 0→B
    - I=II+IB
    - Sum+=V[I]
    - delta=V[I]-V[I-1]
    - SumSq+=V[I]*V[I]
    - ... if (overflow)
      - ...

Binding Times

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

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>CBD</th>
<th>SCBD</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSSI</td>
<td>SCSSI</td>
</tr>
<tr>
<td>CESI</td>
<td>SCESI</td>
</tr>
<tr>
<td>CASI</td>
<td>SCASI</td>
</tr>
<tr>
<td>const</td>
<td></td>
</tr>
</tbody>
</table>

  ... Constant between definitions
  ... + signed
  ... Constant in some scope invocations
  ... + signed
  ... Constant in each scope invocation
  ... + signed
  ... Constant across scope invocations
  ... + signed
  ... Constant across program invocations
  ... + signed
  ... declared const

Experiments

- Applications:
  - UCLA MediaBench:
    - adpcm, epic, g721, gsm, jpeg, mesa, mpeg2
      (not shown today - ghostscript, pegwit, pgp, rasta)
  - gzip, versatility, SPECint95 (parts)
- Compiler optimize instrument for profiling 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
  - $T_{spcl} = T_{sc} + T_{load}$
  - $AT_{gen} = A_{gen} \times T_{gen}$
  - $AT_{spcl} = A_{spcl} \times (T_{sc} + T_{load})$

- If compute long enough
  - $T_{sc} \gg T_{load} \rightarrow$ amortize out load

Storage

- Will have to store configurations somewhere
- LUT ~ $1M \times 2$
- Configuration 64+ bits
  - SRAM: $80K \times 2$ (12-13 for parity)
  - Dense DRAM: $6.4K \times 2$ (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?