ESE534: Computer Organization

Day 25: April 26, 2010
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.
  - …and the least energy, too!

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

Precalss 1: Know More \( \rightarrow \) Less Compute

How does circuit simplify if know \( A=1? \)
Preclass 2: How does circuit simplify if know B=0?

Possible Optimization
• Once know another piece of information about a computation (data value, parameter, usage limit)
• Fold into computation producing smaller computational residue

Preclass 3
• How many 4-LUTs for 8b-equality compare?
• How many 4-LUTs for 8b compare to constant?
Pattern Match

• Savings:
  – 2N bit input computation → N
  – if N variable, maybe trim unneeded portion
  – state elements store target
  – control load target

<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>

Opportunity Exists

• Spatial unfolding of computation
  – can afford more specificity of operation
• 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)

Preclass 4

```c
int main(int argc, char *argv[])
{
  FILE *fd;
  int cnt;
  char *target;
  if (argc == 2)
    strcpy(target, argv[1], MATCH_LENGTH);
  else
    add_open(argv[2], "r");
  cnt = count_matches(fd, target);
  printf(stdout, "Matches: \%d", cnt);
}
```
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</th>
<th>Area and Time</th>
<th>16-16</th>
<th>8-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom 16-16</td>
<td>0.66 µm</td>
<td>2.9M, 4.0 s</td>
<td>9.2</td>
<td>5.5</td>
</tr>
<tr>
<td>Custom 8-8</td>
<td>0.36 µm</td>
<td>3.9M, 4.3 s</td>
<td>1.3</td>
<td>1.3</td>
</tr>
<tr>
<td>Gate-Array 16-16</td>
<td>2.1 µm</td>
<td>2.9M, 3.8 s</td>
<td>2.9</td>
<td>1.3</td>
</tr>
<tr>
<td>FPGA (XILINX)</td>
<td>0.66 µm</td>
<td>1.3M, 17.0 s</td>
<td>0.0067</td>
<td>0.0037</td>
</tr>
<tr>
<td>16b DSP</td>
<td>0.85 µm</td>
<td>570M, 75.0 ns</td>
<td>0.0008</td>
<td>0.0005</td>
</tr>
<tr>
<td>RISC (no multiply)</td>
<td>0.75 µm</td>
<td>128M, 66 cycles</td>
<td>0.017</td>
<td>0.003</td>
</tr>
</tbody>
</table>

Multiply Show

- Specialization in datapath width
- Specialization in data

Benefit Examples

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

Benefits

Empirical Examples
UART

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

UART Composite

<table>
<thead>
<tr>
<th>design</th>
<th>Fully Generic</th>
<th>Speed Mapped</th>
<th>Area Mapped</th>
<th>Specialized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CLBs</td>
<td>path</td>
<td>CLBs</td>
<td>path</td>
</tr>
<tr>
<td>I8251 core</td>
<td>358.5</td>
<td>8.5</td>
<td>348.5</td>
<td>10.5</td>
</tr>
<tr>
<td>Async, 64 cks/bit, 8s2</td>
<td>201</td>
<td>6</td>
<td>141.5</td>
<td>4.5</td>
</tr>
<tr>
<td>Async, 16 cks/bit, 8n1</td>
<td>141.5</td>
<td>4.5</td>
<td>165</td>
<td>4.5</td>
</tr>
<tr>
<td>Sync, external, 2 sync, 8o</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UART FSMs

<table>
<thead>
<tr>
<th>FSM</th>
<th>Fully Generic</th>
<th>Speed Mapped</th>
<th>Area Mapped</th>
<th>Specialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>I8251 processor (n)</td>
<td>11</td>
<td>3.5</td>
<td>11</td>
<td>3.5</td>
</tr>
<tr>
<td>I8251 transmitter</td>
<td>57.5</td>
<td>4.5</td>
<td>57.5</td>
<td>4.6</td>
</tr>
<tr>
<td>I8251 receiver</td>
<td>52.5</td>
<td>5.5</td>
<td>52.5</td>
<td>5.5</td>
</tr>
</tbody>
</table>

Less Than (Bounds check?)

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

Multiply

- How savings in a multiply by constant?
- Multiply by 80?
  - 0101000
- Multiply by 255?

Multiply (revisited)

- Specialization can be more than constant propagation
- Naïve,
  - 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
- Example: multiply by 255:
  - \( 256x-x = 255x \)
  - \( t1=x<<8 \)
  - \( res=t1-x \)

Multiply

- At most \( \lfloor N/2 \rfloor + 2 \) adds for any constant
- Exploiting common subexpressions can do better:
  - e.g.
    - \( c=10101010 \)
    - \( t1=x+x<<2 \) (101x)
    - \( t2=t1<<5+t1<<1 \)

Multiply Example

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Feature Size (( \mu ))</th>
<th>Area and Time</th>
<th>16-16</th>
<th>8-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom 16x16</td>
<td>0.66 \mu m</td>
<td>2.6M, 45 ns</td>
<td>9.5</td>
<td>9.5</td>
</tr>
<tr>
<td>Custom 8x8</td>
<td>0.85 \mu m</td>
<td>3.2M, 43 ns</td>
<td>1.3</td>
<td>1.3</td>
</tr>
<tr>
<td>Gate-Array 16x16</td>
<td>0.75 \mu m</td>
<td>2.8M, 40 ns</td>
<td>1.3</td>
<td>1.3</td>
</tr>
<tr>
<td>RAGA (9X96)</td>
<td>0.65 \mu m</td>
<td>1.9M, 26 ns</td>
<td>0.097</td>
<td>0.04</td>
</tr>
<tr>
<td>16b DSP</td>
<td>0.66 \mu m</td>
<td>3.9M, 48 ns</td>
<td>0.057</td>
<td>0.057</td>
</tr>
<tr>
<td>RISC (no multiplier)</td>
<td>0.75 \mu m</td>
<td>1.5M, 16cycles</td>
<td>0.04</td>
<td>0.035</td>
</tr>
</tbody>
</table>
| 16b adders | 16b operations -- 44 cycles
  10b constant -- 7 cycles
  8b constant -- 4 cycles
| 16b adders | 16b operations -- 44 cycles
  10b constant -- 7 cycles
  8b constant -- 4 cycles

Example: FIR Filtering

Application metric:
TAPs = filter taps
multiply accumulate

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Feature Size (( \mu ))</th>
<th>( TAPs/2^N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>32b RISC</td>
<td>0.75 \mu m</td>
<td>0.020</td>
</tr>
<tr>
<td>16b DSP</td>
<td>0.66 \mu m</td>
<td>0.057</td>
</tr>
<tr>
<td>32b RISC/DSP</td>
<td>0.65 \mu m</td>
<td>0.021</td>
</tr>
<tr>
<td>64b RISC</td>
<td>0.18 \mu m</td>
<td>0.064</td>
</tr>
<tr>
<td>FPGA (Xilinx)</td>
<td>0.60 \mu m</td>
<td>1.9</td>
</tr>
<tr>
<td>FPGA (Altera)</td>
<td>0.30 \mu m</td>
<td>3.6</td>
</tr>
<tr>
<td>Full Custom</td>
<td>0.75 \mu m</td>
<td>3.6</td>
</tr>
<tr>
<td>(fixed coefficient) (n.b. 16b samples)</td>
<td>0.60 \mu m</td>
<td>56</td>
</tr>
</tbody>
</table>
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: ATR

- 16x16x2=512 flops to hold single target pattern
- 16x16=256 LUTs to compute match
- 256 score bits→8b score ~ 500 adder bits in tree
- more for retiming

~800 LUTs here
Maybe fit 1 generic template in XC4010 (400 CLBs)?

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]

Usage Classes

Known Binding Time

- Sum=0
- For I=0→N
  - SUM+=V[I]
- For I=0→N
  - VN[I]=V[I]/SUM
- Scope/Procedure Invocation
  - Scale(max,min,V)
  - for I=0→V.length
    - tmp=(V[I]-min)
    - Vres[I]=tmp/(max-min)
Dynamic Binding Time

- $c_{exp}=0$;
- For $i=0 \rightarrow V.length$
  - if ($V[i].exp!=c_{exp}$)
    - $c_{exp}=V[i].exp$;
  - $V_{res}[i]=V[i].mant<<c_{exp}$

Thread 1:
- $a=src.read()$
- if ($a.newavg())$
  - $avg=a.avg()$

Thread 2:
- $v=data.read()$
- out.write($vavg$)

Empirical Binding

- Have to check if value changed
  - Checking value $O(N)$ area [pattern match]
  - Interesting because computations
    - can be $O(2^N)$ [Day 14]
    - often greater area than pattern match
  - Also Rent’s Rule:
    - Computation $> linear$ in IO
    - $IO=C n^p \rightarrow n \propto IO^{(1/p)}$

Common/Uncommon Case

- For $i=0 \rightarrow N$
  - if ($V[i]==10$)
    - $SumSq+=V[i]*V[i]$;
  - elseif ($V[i]<10$)
    - $SumSq+=V[i]*V[i]$;
  - else
    - $SumSq+=V[i]*V[i]$;

- For $i=0 \rightarrow N$
  - if ($V[i]==10$)
    - $SumSq=100$
  - 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.
Bit Constancy Lattice

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

<table>
<thead>
<tr>
<th>Constancy</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBD</td>
<td>Constant between definitions</td>
<td>... + signed</td>
</tr>
<tr>
<td>SCBD</td>
<td>Constant in some scope invocations</td>
<td>... + signed</td>
</tr>
<tr>
<td>CSSI</td>
<td>Constant in each scope invocation</td>
<td>... + signed</td>
</tr>
<tr>
<td>CEBSI</td>
<td>Constant across scope invocations</td>
<td>... + signed</td>
</tr>
<tr>
<td>CASI</td>
<td>Constant across program invocations</td>
<td>... + signed</td>
</tr>
<tr>
<td>CAPI</td>
<td>... declared const</td>
<td></td>
</tr>
</tbody>
</table>

Experiments

- Applications:
  - UCLA MediaBench:
    - adpcm, epic, g721, gsm, jpeg, mesa, mpeg2
    - (not shown today - ghostscript, pegwit, ppg, 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

Empirical 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

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

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

- Benefits come from reduced area & energy
  - reduced area → performance
  - room for more spatial operation
  - maybe less interconnect delay
- Challenge: 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
  - \( T_{sc} \gg T_{load} \) → amortize out load

Storage

- Will have to store configurations somewhere
  - LUT ~ 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, flash
- 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?
Admin

• No new reading 😊
• Discussion period ends today
• Final due May 10 (2 weeks)
  – Traveling early next week
  – Not up late weekend before due
• Homework 7,8 still ungraded 😞
  – Hope to start end of week

Big Ideas [MSB]

• 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 [MSB-1]

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