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
- **Opportunity**: Some data known or predictable
  – compute minimum computational residue
- As know more data → reduce computation
- Dual of generalization we saw for local control

Preclass 1:

Know More → Less Compute

How does circuit simplify if know A=1?
Know More → Less Compute

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

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 compare(char *target, char *potential)
{
    int i;
    char *t = target;
    char *p = potential;
    for (i=0; i<MATCH_LENGTH; ++i)
    {
        if (t[i] == p[i])
            return 0;
        ++t;
        ++p;
    }
    return 1;
}

int count_matches(FILE *fd, char *target)
{
    char *line = (char *)malloc(sizeof(char) * MAX_LINE_LENGTH);
    int matches = 0;
    while (fgets(line, fd, MAX_LINE_LENGTH))
    {
        if (compare(target, line) == 0)
            ++matches;
        line = NULL;
    }
    return matches;
}
```

Preclass 4: Circuit
Preclass 4: Circuit

Reconfiguring Logic

• Simple model:
  – Address like memory

• Today’s commercial devices:
  – Shift configuration in serially
    • Slower but cheaper
  – Segmented
    • So can reconfigure only part of the chip at a time

Reconfiguring Logic

• Simple model:
  – Address like memory

Optimization Prospects

• Area-Time Tradeoff
  – $T_{spcl} = T_{sc} + T_{load}$
  – $T_{sc} = N \times T_{scycle}$
  – $T_{gen} = N \times T_{gcycle}$
  – $AT_{gen} = A_{gen} \times T_{gen}$
  – $AT_{spcl} = A_{spcl} \times (T_{sc} + T_{load})$

  • If compute long enough (N large enough)
    – $T_{sc} >\gg T_{load}$ → amortize out load

Preclass 5

• $T_{slod}=100\mu s$, $T_{sycyle}=1\text{ns}$
• $T_{gload}=0$, $T_{gcycle}=2\text{ns}$

• Ratio $T_{gslask}/T_{cslask}$ for $N=10^4$ ?
• Breakeven $N$?

Optimization Prospects

• Area-Time Tradeoff
  – $T_{spcl} = T_{sc} + T_{load}$
  – $T_{sc} = N \times T_{scycle}$
  – $T_{gen} = N \times T_{gcycle}$
  – $AT_{gen} = A_{gen} \times T_{gen}$
  – $AT_{spcl} = A_{spcl} \times (T_{sc} + T_{load})$

  • If compute long enough (N large enough)
    – $T_{sc} >\gg T_{load}$ → amortize out load
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 ((1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom 16–16</td>
<td>0.62 µm</td>
</tr>
<tr>
<td>Custom 8–8</td>
<td>0.96 µm</td>
</tr>
<tr>
<td>Space-Andys 15–16</td>
<td>0.75 µm</td>
</tr>
<tr>
<td>TRISA (RCB)</td>
<td>0.66 µm</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Area and Time</th>
<th>16:16</th>
<th>8:8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>in ms</td>
<td>scale</td>
</tr>
<tr>
<td>Custom 16–16</td>
<td>3.88</td>
<td>9.7</td>
</tr>
<tr>
<td>Custom 8–8</td>
<td>3.25</td>
<td>7.8</td>
</tr>
<tr>
<td>Space-Andys</td>
<td>3.16</td>
<td>9.5</td>
</tr>
<tr>
<td>TRISA</td>
<td>1.25</td>
<td>1.8</td>
</tr>
</tbody>
</table>

Multiply Show

• Specialization in datapath width
• Specialization in data

Benefits

Empirical Examples

Benefit Examples

• Less than
• Multiply revisited
  – more than just constant propagation
• ATR

Less Than (Bounds check?)

<table>
<thead>
<tr>
<th>Function</th>
<th>Speed Mapped path</th>
<th>Area Mapped path</th>
<th>Speed Mapped path</th>
<th>Area Mapped path</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a \leq b)</td>
<td>Speed Mapped path</td>
<td>Area Mapped path</td>
<td>Speed Mapped path</td>
<td>Area Mapped path</td>
</tr>
<tr>
<td>(a \leq b)</td>
<td>Speed Mapped path</td>
<td>Area Mapped path</td>
<td>Speed Mapped path</td>
<td>Area Mapped path</td>
</tr>
</tbody>
</table>

| \(a \leq b\) | Speed Mapped path | Area Mapped path | Speed Mapped path | Area Mapped path |
| \(a \leq b\) | Speed Mapped path | Area Mapped path | Speed Mapped path | Area Mapped path |
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 \[ \lfloor \frac{N}{2} \rfloor \] one bits in constant
- Example: multiply by 255:
  - \(256x - x = 255x\)
  - \(t1 = x << 8\)
  - \(res = t1 - x\)

Multiply

- Never really need more than \[ \lfloor \frac{N}{2} \rfloor \] 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 \(x\) \((2^{N+1} - c)x\)
  - 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.
    - \(c = 10101010\)
    - \(t1 = x + x << 2\) \(\left(101x\right)\)
    - \(t2 = t1 << 5 + t1 << 1\)
    - \(\Rightarrow 2\) adds instead of 3
Multiply Example

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Feature Size (L)</th>
<th>Area and Time</th>
<th>16-16</th>
<th>8-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom 16x16</td>
<td>0.85μm</td>
<td>2.9M 17.4 ms</td>
<td>9.8</td>
<td>9.8</td>
</tr>
<tr>
<td>Custom 8x8</td>
<td>0.85μm</td>
<td>3.2M 4.5 ms</td>
<td>7.5</td>
<td>7.5</td>
</tr>
<tr>
<td>Globe-Andy 16x16</td>
<td>0.79μm</td>
<td>3M 17.2 ms</td>
<td>12.3</td>
<td>12.3</td>
</tr>
<tr>
<td>FPGA</td>
<td>0.66μm</td>
<td>1.25M 17.66 ms</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>(no multiplier)</td>
<td>0.66μm</td>
<td>3M 17.66 ms</td>
<td>0.66</td>
<td>0.66</td>
</tr>
</tbody>
</table>

Example: FIR Filtering

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

Application metric:
TAPs = filter taps
multiply accumulate

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Feature Size (L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32b RISC</td>
<td>0.75μm</td>
</tr>
<tr>
<td>16b DSP</td>
<td>0.66μm</td>
</tr>
<tr>
<td>32b RISC/DSP</td>
<td>0.25μm</td>
</tr>
<tr>
<td>64b RISC</td>
<td>0.15μm</td>
</tr>
<tr>
<td>FPGA (XC40K)</td>
<td>0.60μm</td>
</tr>
<tr>
<td>Full Custom</td>
<td>0.30μm</td>
</tr>
<tr>
<td>(Altara 8X)</td>
<td>3.6</td>
</tr>
<tr>
<td>(n.b. 16b samples)</td>
<td>56</td>
</tr>
</tbody>
</table>

Opportunity Exists

- Spatial unfolding of computation
  - can afford more specificity of operation

- What opportunity do we lose if sequentializing on single multiplier?

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 without cares
  - need to track many (e.g. 70-100) templates for each image region
  - templates themselves are sparse
    - small fraction of care pixels

- 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

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

Specialization Usage Classes

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

Known Binding Time

- $\text{Sum}=0$
- For $i=0 \rightarrow N$
  - $\text{Sum}+=V[i]$
- For $i=0 \rightarrow N$
  - $\text{VN}[i]=V[i]/\text{Sum}$

Scope/Procedure Invocation

$\text{Scale}(\text{max}, \text{min}, V)$
- for $i=0 \rightarrow V$.length
  - $\text{tmp}=(V[i]-\text{min})$
  - $\text{Vres}[i]=\text{tmp}/(\text{max}-\text{min})$

Dynamic Binding Time

- $\text{cexp}=0$
- For $i=0 \rightarrow V$.length
  - if ($V[i].\text{exp}\neq\text{cexp}$)
    - $\text{cexp}=V[i].\text{exp}$
    - $\text{Vres}[i]=V[i].\text{mant}<<\text{cexp}$

Thread 1:
- $a=\text{src.read()}$
- if ($a.\text{newavg}()$)
  - $\text{avg}=a.\text{avg}()$

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 13]
    - often greater area than pattern match
  - Also Rent’s Rule:
    - Computation $>\text{linear in IO}$
    - $\text{IO}=C \cdot n^p \rightarrow n \propto \text{IO}^{1/p}$

Common/Uncommon Case

- For $i=0 \rightarrow N$
  - $\text{SumSq}+=V[i]^2$;

- What if we know that $V[j]=10$ half the time?

- What if we know that $V[j]\leq10$ another $40\%$ of the time?
**Common/Uncommon Case**

- For \( i = 0 \rightarrow N \)
  - If \( V[i] == 10 \)
    - \( \text{SumSq} += V[i] \cdot V[i] \);
  - elseif \( V[i] < 10 \)
    - \( \text{SumSq} += V[i] \cdot V[i] \);
  - else
    - \( \text{SumSq} += V[i] \cdot V[i] \);

- For \( i = 0 \rightarrow N \)
  - If \( V[i] == 10 \)
    - \( \text{SumSq} += 100 \);
  - elseif \( V[i] < 10 \)
    - \( \text{SumSq} += V[i] \cdot V[i] \);
  - else
    - \( \text{SumSq} += V[i] \cdot V[i] \);

**Potential Binding Times**

- What are the potential binding times for values?
  - i.e. at what points might values be defined then held constant?

**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**

(Time Permitting)

**Bit Constancy Lattice**

- binding time for bits of variables (storage-based)
  - \( \text{CSD} \)
  - \( \text{CSS} \)
  - \( \text{CSI} \)
  - \( \text{CES} \)
  - \( \text{CASI} \)
  - \( \text{CAP} \)
  - \( \text{const} \)

  ....... 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
  ....... declared const

[Experiment: Eylon Caspi/UCB]
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

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

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 $\rightarrow$ 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

Storage

- Will have to store configurations somewhere
- LUT $\sim 250K F^2$
- Configuration 64+ bits
  - SRAM: $20KF^2$ (12-13 for parity)
  - Dense DRAM: $1.6KF^2$ (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?

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
Admin

- FM1 due today
- Reading for Monday online