Accelerators For Everything

Bolaji Bankole, Jens Ertman
QsCores (Quasi-Specific Cores)
What is a QsCore

- A hardware accelerator core connected to a CPU
- Composed to accelerate several specific segments of code
- Synthesized hardware determined before the chip is manufactured
- Can be combined with other QsCores to accelerate more at the expense of area
  - And energy but not in the same way (we’ll get to it)
- Can be called with arguments in lieu of running on the general purpose CPU
Motivation

- With advances in transistor technology counts are going up but usable area is going down
- Why not take extra area and make accelerators for common tasks?
  - What if those accelerators focused on energy efficiency?
  - What if those accelerators combined multiple similar “hotspots” of the code to cover more of the runtime?
- More energy efficiency means that more compute can occur on the chip
Mining for Similar Code Patterns

- Generate a program dependence graph for each hotspot in the code
- Compare these graphs based on the similarity of their nodes and dependencies
- Take the two hotspots and generate a new graph that performs both

```
FlipTrackChangeFCL
++
(
if (flipf) {
  if (trueflip) {
    // True
  } else {
    // False
  }
} else {
  if (trueflip) {
    // True
  } else {
    // False
  }
}
```

```
BestLookAheadScore
++
(
if (trueflip) {
  if (trueflip) {
    // True
  } else {
    // False
  }
} else {
  if (trueflip) {
    // True
  } else {
    // False
  }
}
```
Determining the Set of QsCores

- Generate all pairs in the merge set
- Take the highest quality QsCore merge and replace the previous two in the set with it
- Keep going until either an area constraint is met or there is nothing left to merge

$$Q_b = \frac{C_b S_b}{A_b P_b}$$

1: while $|B| > 1$ do
2:   $(b_1, b_2) = \text{vmax}_{(b_1 \in B, b_2 \in B)} \frac{C_{b_1 \triangleleft b_2}}{A_{b_1 \triangleleft b_2}^2} - \frac{C_{b_1}}{A_{b_1}^2} - \frac{C_{b_2}}{A_{b_2}^2}$
3:   $B = B \setminus \{b_1, b_2\}$
4:   $B = B \cup \{b_1 \triangleleft b_2\}$
5:   Record the merging of $b_1$ and $b_2$ and the resulting values of $Q'_B$ and $\sum_{b \in B} A_b$.
6: end while
**Physical QsCores**

- Generated from the C code to verilog then synthesized
- Cores are then integrated with a CPU with shared D and I cache using scan chains
Results Core Count

- Energy use increases slower than decreasing area
- Much fewer cores required to cover a larger number of features
Quality of QsCores

- In testing the set of QsCores determined by their algorithm it created the best set of QsCores in all cases.
- QsCores are backwards compatible if old versions of the code are included in the set of hotspots to be merged.
## Final Results of Energy Efficiency

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Q SCORE power</td>
<td>6.7 mW</td>
</tr>
<tr>
<td>Average Q SCORE energy efficiency (compared to baseline CPU)</td>
<td>23.23×</td>
</tr>
<tr>
<td>Average execution time per Q SCORE invocation</td>
<td>8951 cycles</td>
</tr>
<tr>
<td>Average invocation overhead</td>
<td>316 cycles</td>
</tr>
<tr>
<td>Average system execution coverage per Q SCORE</td>
<td>4.41%</td>
</tr>
<tr>
<td>Average Q SCORE area</td>
<td>0.041 mm²</td>
</tr>
</tbody>
</table>
Conservation Cores
What

- Accelerators with the goal of energy reduction
  - Less sensitive in this than performance oriented accelerators
- Patchable(?) to add flexibility and longevity
- Communicate with the system through shared caches and scan-chain interface
- Very similar idea to QsCores

![Diagram](image-url)
Why

- Breakdown of CMOS scaling means that only so much of a processor can be practically run at full speed
- Trade area for energy efficiency to get better use of the die area
- Same overall rationale as QsCores
How

- Most frequently used code snippets are augmented for reconfigurability and synthesized
- Compiler knows the c-cores in the processor and includes stubs to invoke them, with patches when necessary
C-Core Function

- State machine closely resembles code structure
  - Helps memory ordering
- Multi cycle loops for complex operations and memory
- Small scan chains for arguments, large ones for patches, other ones for internal state
  - Added instructions to move data to and from scan chains
- At runtime, check for relevant c-core and use it if available
Patching

- Configurable constants
  - Registers to change constants in the program
- Generalized operators
- Control flow changes
  - Raise exceptions for CPU to handle, modify conditionals, etc
Results

- Benefits (and costs) of patchability

<table>
<thead>
<tr>
<th>C-core</th>
<th>Ver.</th>
<th>Key</th>
<th>LOC</th>
<th>% Exe.</th>
<th>Area (mm²)</th>
<th>Freq. (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Non-P. Patch</td>
<td>Non-P. Patch</td>
</tr>
<tr>
<td>brj2</td>
<td>1.0</td>
<td>A</td>
<td>231</td>
<td>71.1</td>
<td>0.128</td>
<td>0.128</td>
</tr>
<tr>
<td>fallbackSort</td>
<td>1.05</td>
<td>F</td>
<td>231</td>
<td>71.1</td>
<td>0.128</td>
<td>0.128</td>
</tr>
<tr>
<td>jpgc</td>
<td>extract.MCU</td>
<td>v1</td>
<td>266</td>
<td>49.3</td>
<td>0.108</td>
<td>0.205</td>
</tr>
<tr>
<td></td>
<td>getrgb.yccRows</td>
<td>v1</td>
<td>39</td>
<td>5.1</td>
<td>0.020</td>
<td>0.044</td>
</tr>
<tr>
<td></td>
<td>subsample</td>
<td>v1</td>
<td>40</td>
<td>17.7</td>
<td>0.023</td>
<td>0.039</td>
</tr>
<tr>
<td>jpgc</td>
<td>extract.MCU</td>
<td>v2</td>
<td>277</td>
<td>49.5</td>
<td>0.108</td>
<td>0.205</td>
</tr>
<tr>
<td></td>
<td>getrgb.yccRows</td>
<td>v2</td>
<td>37</td>
<td>5.1</td>
<td>0.020</td>
<td>0.044</td>
</tr>
<tr>
<td></td>
<td>subsample</td>
<td>v2</td>
<td>36</td>
<td>17.8</td>
<td>0.023</td>
<td>0.039</td>
</tr>
<tr>
<td>jpgc</td>
<td>jpeg.dct_low</td>
<td>v5</td>
<td>233</td>
<td>21.5</td>
<td>0.133</td>
<td>0.222</td>
</tr>
<tr>
<td>ycc.rgbconvert</td>
<td>v5</td>
<td>35</td>
<td>33.0</td>
<td>0.023</td>
<td>0.043</td>
<td>1663</td>
</tr>
<tr>
<td>jpgc</td>
<td>jpeg.dct_low</td>
<td>v6</td>
<td>236</td>
<td>21.7</td>
<td>0.135</td>
<td>0.222</td>
</tr>
<tr>
<td>ycc.rgbconvert</td>
<td>v6</td>
<td>35</td>
<td>33.7</td>
<td>0.024</td>
<td>0.043</td>
<td>1676</td>
</tr>
<tr>
<td>mcf</td>
<td>primal.a.ba</td>
<td>2000</td>
<td>A</td>
<td>64</td>
<td>35.2</td>
<td>0.033</td>
</tr>
<tr>
<td></td>
<td>refresh.potential</td>
<td>2000</td>
<td>A</td>
<td>44</td>
<td>8.8</td>
<td>0.017</td>
</tr>
<tr>
<td>mcf</td>
<td>primal.a.ba</td>
<td>2006</td>
<td>B</td>
<td>64</td>
<td>53.3</td>
<td>0.032</td>
</tr>
<tr>
<td></td>
<td>refresh.potential</td>
<td>2006</td>
<td>B</td>
<td>41</td>
<td>1.3</td>
<td>0.015</td>
</tr>
<tr>
<td>vpr</td>
<td>try_avap</td>
<td>4.22</td>
<td>A</td>
<td>858</td>
<td>61.1</td>
<td>0.181</td>
</tr>
<tr>
<td></td>
<td>try_avap</td>
<td>4.3</td>
<td>B</td>
<td>861</td>
<td>27.0</td>
<td>0.181</td>
</tr>
</tbody>
</table>

Table 3. Conservation core statistics The c-cores we generated vary greatly in size and complexity. In the “Key” column, the letters correspond to application versions and the Roman numerals denote specific functions from the application that a c-core targets. “LOC” is lines of C source code, and “% Exe” is the percentage of execution that each function comprises in the application.

Figure 7. Conservation core effectiveness over time Since c-cores target stable applications, they can deliver efficiency gains over a very long period of time.
Results