Register Pointer Architecture for Efficient Embedded Processors

JongSoo Park, Sung-Boem Park, James Balfour, David Black-Schaffer, Christos Kozyrakis, William Dally
Stanford University
Register Pointer Architecture (RPA)

- Indirection
- Capture More Locality
- Performance ↑, without Power and Code Size ↑
Embedded Computing

Real time constraint

Energy Efficiency

Cost Efficiency

30 frame/sec voice

Jongsoo Park, Stanford University
Inefficient Microprocessor

<table>
<thead>
<tr>
<th>Type</th>
<th>MOPS/mW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microprocessor</td>
<td>0.13</td>
</tr>
<tr>
<td>DSP</td>
<td>7</td>
</tr>
<tr>
<td>ASIC</td>
<td>200</td>
</tr>
</tbody>
</table>

[Broderson, ISSCC 2002]
How to close the gap?

- Efficient Embedded Computing (EEC)

- Large portion of energy spent on data supply
  - 45% energy go to cache [Segars, ISSCC 2001]

- This work’s focus: Energy efficient data supply
Memory Hierarchy

Register Cache Main Memory

Fast, Close Less Energy

Inflexible
for (i = 0; i < NUM_IN - 3; i++) {
    acc = 0;
    for (j = 0; j < 3; j++) {
        acc += coeff[j]*in[i + j];
    }
    out[j] = acc;
}
Unrolling

Inner-loop unrolling

\[
\begin{align*}
\text{coeff0} &= \text{coeff}[0]; \quad \text{coeff1} = \text{coeff}[1]; \\
\text{coeff2} &= \text{coeff}[2]; \\
\text{for} \ (i = 0; \ i < \text{NUM}_{-}\text{IN} - 3; \ i++) \ {}
\end{align*}
\]

\[
\begin{align*}
\text{acc} &= \text{coeff0} \times \text{in}[i]; \\
\text{acc} &= \text{coeff1} \times \text{in}[i+1]; \\
\text{acc} &= \text{coeff2} \times \text{in}[i+2]; \\
\text{out}[i] &= \text{acc};
\end{align*}
\]

- coeff0~2: allocated in registers
- 3 loads per input
- code size: \(\mathcal{O}(\# \text{ of taps})\)

Without unrolling

\[
\begin{align*}
\text{for} \ (i = 0; \ i < \text{NUM}_{-}\text{IN} - 3; \ i++) \ {}
\end{align*}
\]

\[
\begin{align*}
\text{acc} &= 0; \\
\text{for} \ (j = 0; \ j < 3; \ j++) \ {}
\end{align*}
\]

\[
\begin{align*}
\text{acc} &= \text{coeff}[j] \times \text{in}[i + j]; \\
\text{out}[j] &= \text{acc};
\end{align*}
\]

- 6 loads per input
Full Unrolling

\[
in0 = \text{in}[0]; \quad in1 = \text{in}[1];
\]

\[
\text{for } (i = 0; i < \text{NUM}_\text{IN} - 3; i += 3 ) \{ \\
\quad in2 = \text{in}[i + 2]; \\
\quad acc = \text{coeff0}\times in0; \\
\quad acc += \text{coeff1}\times in1; \\
\quad acc += \text{coeff2}\times in2; \\
\quad out[i] = acc;
\}
\]

\[
in0 = \text{in}[i + 3]; \\
\quad acc = \text{coeff0}\times in1; \\
\quad acc += \text{coeff1}\times in2; \\
\quad acc += \text{coeff2}\times in0; \\
\quad out[i+1] = acc;
\]

\[
in1 = \text{in}[i + 4]; \\
\quad acc = \text{coeff0}\times in2; \\
\quad acc += \text{coeff1}\times in0; \\
\quad acc += \text{coeff2}\times in1; \\
\quad out[i + 2] = acc;
\}
\]

- 1 load per input
- code size: \(O((\# \text{ of taps})^2)\)
Problems of Unrolling

- Code size
  - 35 taps FIR with ARM ISA
    - Inner loop unroll: 14 instruction $\rightarrow$ 75 instructions (5.4x)
    - Fully unroll: 14 instructions $\rightarrow$ 1229 instructions (88x)
Our Approach

Indirection ↔ Unrolling

Capture More Locality
Register Pointer Architecture (RPA)

Instruction → Register Pointers → Register File

IF → ID → EX → MEM → WB

Jongsoo Park, Stanford University
FIR with RPA (2)

\[ acc = in0*coeff0 + in2*coeff1 + in0*coeff2 \]

Jongsoo Park, Stanford University
Experiment Setup

• Configuration

Baseline 16 Regs VS RPA 64 Regs VS Unrolling 64 Regs

• Applications

Unrollable?

No → Sort, Merge, StringSearch, PGP

Yes → 1DFIR, MD5, MatMult, 2DFIR, GSM, TiffDither

• ARM ISA, SimpleScalar, Panalyzer
Execution Time

replaces memory accesses with register accesses
fewer cache accesses compensate larger register file’s energy consumption
Execution Time: RPA vs. Unrolling

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>RPA</th>
<th>Unroll</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort</td>
<td>0.61</td>
<td>0.41</td>
</tr>
<tr>
<td>Merge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>StringSearch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PGPVerify</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1DFIR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MD5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MatMult</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2DFIR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GSM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TiffDither</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Avg. 0.89
Comparison with Unrolling

Normalized execution time

execute
branch
l-miss
D-miss
ld/st

Jongsoo Park, Stanford University
Total Code Size

Normalized Code Size

- 1DFIR
- MD5
- MatMult
- 2DFIR
- GSM
- Tiffdither
- Avg.

Comparison between RPA and unroll:
- 3.7 (1DFIR RPA)
- 1.52 (2DFIR unroll)
- 1.09 (Avg.)
Summary of Comparison

- Better
- Energy
- Code Size
- Unroll
- RPA
- Base
Conclusion

Indirection (RPA) ↔ Unrolling

Capture More Locality

30% Performance ↑, without Power and Code Size ↑