How Computers Work
Jakob Stoklund Olesen
Apple
How Computers Work

- Out of order CPU pipeline
- Optimizing for out of order CPUs
- Machine trace metrics analysis
- Future work
Out of Order CPU Pipeline

Branch Predictor -> Fetch -> Decode -> Rename -> Scheduler -> Load -> ALU -> ALU -> Br -> Retire -> Reorder Buffer
Dot Product

int dot(int a[], int b[], int n)
{
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += a[i]*b[i];
    return sum;
}

Dot Product

loop:
  ldr r3 ← [r0, r6, lsl #2]
  ldr r4 ← [r1, r6, lsl #2]
  mul r3 ← r3, r4
  add r5 ← r3, r5
  add r6 ← r6, #1
  cmp r6, r2
  bne loop
loop:
    ldr r3 ← [r0, r6, lsl #2]
    ldr r4 ← [r1, r6, lsl #2]
    mul r3 ← r3, r4
    add r5 ← r3, r5
    add r6 ← r6, #1
    cmp r6, r2
    bne loop

Retire →

p100 ← ldr [p10, p94, lsl #2]
p101 ← ldr [p11, p94, lsl #2]
p102 ← mul p100, p101
p103 ← add p102, p95
p104 ← add p94, #1
p105 ← cmp p104, p12
    bne p105, taken

Renamed

Speculate →
p106 ← ldr [p10, p104, lsl #2]
p107 ← ldr [p11, p104, lsl #2]
p108 ← mul p107, p106
p109 ← add p108, p103
p110 ← add p104, #1
p111 ← cmp p110, p12
    bne p111, taken

Reordered Buffer
p100 ← ldr [p10, p94, lsl #2]  
p101 ← ldr [p11, p94, lsl #2]  
p102 ← mul p100, p101  
p103 ← add p102, p95  
p104 ← add p94, #1  
p105 ← cmp p104, p12  
bne p105, taken  

p106 ← ldr [p10, p104, lsl #2]  
p107 ← ldr [p11, p104, lsl #2]  
p108 ← mul p107, p106  
p109 ← add p108, p103  
p110 ← add p104, #1  
p111 ← cmp p110, p12  
bne p111, taken  

p112 ← ldr [p10, p110, lsl #2]  
p113 ← ldr [p11, p110, lsl #2]  
p114 ← mul p112, p113
p100 ← ldr [p10, p94, lsl #2]
p101 ← ldr [p11, p94, lsl #2]
p102 ← mul p100, p101
p103 ← add p102, p95
p104 ← add p94, #1
p105 ← cmp p104, p12
bne p105, taken

p106 ← ldr [p10, p104, lsl #2]
p107 ← ldr [p11, p104, lsl #2]
p108 ← mul p107, p106
p109 ← add p108, p103
p110 ← add p104, #1
p111 ← cmp p110, p12
bne p111, taken

p112 ← ldr [p10, p110, lsl #2]
p113 ← ldr [p11, p110, lsl #2]
p114 ← mul p112, p113
\[\text{p100} \leftarrow \text{ldr} \ [\text{p10}, \text{p94}, \text{lsl} \ #2]\]
\[\text{p101} \leftarrow \text{ldr} \ [\text{p11}, \text{p94}, \text{lsl} \ #2]\]
\[\text{p102} \leftarrow \text{mul} \ \text{p100}, \text{p101}\]
\[\text{p103} \leftarrow \text{add} \ \text{p102}, \text{p95}\]
\[\text{p104} \leftarrow \text{add} \ \text{p94}, \#1\]
\[\text{p105} \leftarrow \text{cmp} \ \text{p104}, \text{p12}\]
\[\text{bne} \ \text{p105}, \text{taken}\]
\[\text{p106} \leftarrow \text{ldr} \ [\text{p10}, \text{p104}, \text{lsl} \ #2]\]
\[\text{p107} \leftarrow \text{ldr} \ [\text{p11}, \text{p104}, \text{lsl} \ #2]\]
\[\text{p108} \leftarrow \text{mul} \ \text{p107}, \text{p106}\]
\[\text{p109} \leftarrow \text{add} \ \text{p108}, \text{p103}\]
\[\text{p110} \leftarrow \text{add} \ \text{p104}, \#1\]
\[\text{p111} \leftarrow \text{cmp} \ \text{p110}, \text{p12}\]
\[\text{bne} \ \text{p111}, \text{taken}\]
\[\text{p112} \leftarrow \text{ldr} \ [\text{p10}, \text{p110}, \text{lsl} \ #2]\]
\[\text{p113} \leftarrow \text{ldr} \ [\text{p11}, \text{p110}, \text{lsl} \ #2]\]
\[\text{p114} \leftarrow \text{mul} \ \text{p112}, \text{p113}\]
p100 ← ldr [p10, p94, lsl #2]
p101 ← ldr [p11, p94, lsl #2]
p102 ← mul p100, p101
p103 ← add p102, p95
p104 ← add p94, #1
p105 ← cmp p104, p12
bne p105, taken
p106 ← ldr [p10, p104, lsl #2]
p107 ← ldr [p11, p104, lsl #2]
p108 ← mul p107, p106
p109 ← add p108, p103
p110 ← add p104, #1
p111 ← cmp p110, p12
bne p111, taken
p112 ← ldr [p10, p110, lsl #2]
p113 ← ldr [p11, p110, lsl #2]
p114 ← mul p112, p113
Throughput

- Map μops to functional units
- One μop per cycle per functional unit
- Multiple ALU functional units
- ADD throughput is 1/3 cycle/instruction
Multiply-Accumulate

loop:
  ldr  r3 ← [r0, r6, lsl #2]
  ldr  r4 ← [r1, r6, lsl #2]
  mla  r5 ← r3, r4, r5
  add  r6 ← r6, #1
  cmp  r6, r2
  bne  loop
Load

<table>
<thead>
<tr>
<th>Load</th>
<th>ALU</th>
<th>ALU</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ldr</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ldr</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ldr</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ldr</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ldr</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ldr</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>mla</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>mla</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>mla</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>mla</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

loop:
ldr r3 ← [r0, r6, lsl #2]
ldr r4 ← [r1, r6, lsl #2]
mla r5 ← r3, r4, r5
add r6 ← r6, #1
cmp r6, r2
bne loop

4 cycles loop-carried dependence
2x slower!
int len(node *p)
{
    int n = 0;
    while (p)
        p = p->next, n++
    return n;
}
Pointer Chasing

loop:
  ldr r1 ← [r1]
  add r0 ← r0, #1
  cmp r1, #0
  bxeq lr
  b loop
loop:
  ldr r1 ← [r1]
  add r0 ← r0, #1
  cmp r1, #0
  bxeq lr
  b loop

p100 ← ldr [p97]
p101 ← add p98, #1
p102 ← cmp p100, #0
bxeq p102, not taken
p103 ← ldr [p100]
p104 ← add p101, #1
p105 ← cmp p104, #0
bxeq p105, not taken
p106 ← ldr [p103]
p107 ← add p104, #1
p108 ← cmp p107, #0
bxeq p108, not taken
p100 ← ldr [p97]
p101 ← add p98, #1
p102 ← cmp p100, #0
bxeq p102, not taken
p103 ← ldr [p100]
p104 ← add p101, #1
p105 ← cmp p104, #0
bxeq p105, not taken
p106 ← ldr [p103]
p107 ← add p104, #1
p108 ← cmp p107, #0
bxeq p108, not taken
p100 ← ldr [p97]
p101 ← add p98, #1
p102 ← cmp p100, #0
bxeq p102, not taken
p103 ← ldr [p100]
p104 ← add p101, #1
p105 ← cmp p104, #0
bxeq p105, not taken
p106 ← ldr [p103]
p107 ← add p104, #1
p108 ← cmp p107, #0
bxeq p108, not taken
p100 ← ldr [p97]
p101 ← add p98, #1
p102 ← cmp p100, #0
bxeq p102, not taken
p103 ← ldr [p100]
p104 ← add p101, #1
p105 ← cmp p104, #0
bxeq p105, not taken
p106 ← ldr [p103]
p107 ← add p104, #1
p108 ← cmp p107, #0
bxeq p108, not taken
Latency

• Each µop must wait for operands to be computed
• Pipelined units can use multiple cycles per instruction
• Load latency is 4 cycles from L1 cache
• Long dependency chains cause idle cycles
What Can Compilers Do?

- Reduce number of μops
- Reduce dependency chains to improve instruction-level parallelism
- Balance resources: Functional units, architectural registers
- Go for code size if nothing else helps
Reassociate

- Maximize ILP
- Reduce critical path
- Beware of register pressure
Unroll Loops

• Small loops are unrolled by OoO execution
• Unroll very small loops to reduce overhead
• Unroll large loops to expose ILP by scheduling iterations in parallel
• Only helps if iterations are independent
• Beware of register pressure
Unroll and Reassociate

loop:
   mla r1 ← ..., r1
   mla r2 ← ..., r2
   mla r3 ← ..., r3
   mla r4 ← ..., r4
end:
   add r0 ← r1, r2
   add r1 ← r3, r4
   add r0 ← r0, r1
Unroll and Reassociate

• Difficult after instruction selection
• Handled by the loop vectorizer
• Needs to estimate register pressure on IR
• MI scheduler can mitigate some register pressure problems
Schedule for OoO

• No need for detailed itineraries
• New instruction scheduling models
• Schedule for register pressure and ILP
• Overlap long instruction chains
• Keep track of register pressure
If-conversion

mov (...) → rdx
mov (...) → rsi
lea (rsi, rdx) → rcx
lea 32768(rsi, rdx) → rsi
cmp 65536, rsi
jb end

lea (rsi, rdx) → rcx
lea 32768(rsi, rdx) → rsi
cmp 65536, rsi
cmovnb rdx → rcx
mov cx, (…)

end:
mov cx, (…)

mov (...) → rdx
mov (...) → rsi
lea (rsi, rdx) → rcx
test rcx, rcx
mov -32768 → rcx
cmovg r8 → rdx
cmp 65536, rsi
cmovnb rdx → rcx
mov cx, (…)

If-conversion

- Reduces branch predictor pressure
- Avoids expensive branch mispredictions
- Executes more instructions
- Can extend the critical path
- Includes condition in critical path
If-conversion

mov (...) → rdx
mov (...) → rsi
lea (rsi, rdx) → rcx

lea 32768(rsi, rdx) → rsi
cmp 65536, rsi
jb end

ten rcx, rcx
mov -32768 → rcx
cmovg r8 → rcx

dend:
mov cx, (...)
If-conversion

mov (...) → rdx
mov (...) → rsi
lea (rsi, rdx) → rcx

test rcx, rcx
mov -32768 → rdx
cmovg r8 → rdx

cmovnb rdx → rcx
mov cx, (…)

lea 32768(rsi, rdx) → rsi
cmp 65536, rsi
Machine Trace Metrics

- Picks a trace of multiple basic blocks
- Computes CPU resources used by trace
- Computes instruction latencies
- Computes critical path and “slack”
Slack

Cmov

Add

Mul

2 cycles slack
Throughput

Add  Add  Add  Br

Mul  Ldr  Ldr
Throughput

- Add
- Add
- Add
- Ldr
- Ldr
- Mul
- Br
Rematerialization

\[
\begin{align*}
\text{mov r1} & \leftarrow 123 \\
\text{str r1} & \rightarrow [\text{sp}+8] \\
\text{loop:} & \\
& \ldots \\
\text{ldr r1} & \leftarrow [\text{sp}+8]
\end{align*}
\]

\[
\begin{align*}
\text{loop:} & \\
& \ldots \\
\text{mov r1} & \leftarrow 123
\end{align*}
\]
Rematerialization

\[
\begin{align*}
\text{mov } r1 & \leftarrow 123 \\
\text{str } r1 & \rightarrow [sp+8]
\end{align*}
\]

loop:
\[
\begin{align*}
\lddr r1 & \leftarrow [sp+8]
\end{align*}
\]

loop:
\[
\begin{align*}
\text{add} & \\
\text{add} & \\
\text{mov } r1 & \leftarrow 123
\end{align*}
\]
Code Motion

- Sink code back into loops
- Sometimes instructions are free
- Use registers to improve ILP
Code Generator

- SelectionDAG
- Early SSA Optimizations
- MachineTraceMetrics
- ILP Optimizations
- LICM, CSE, Sinking, Peephole
- Leaving SSA Form
- MI Scheduler
- Register Allocator
IR Optimizers

- Canonicalization
- Inlining
- Loop Vectorizer
- Loop Strength Reduction
- SelectionDAG
- Target Info
Future Work

• Pass ordering, canonicalization vs lowering
• Late reassociation
• Latency-aware mul/mla transformation
• Code motion, rethink spill costs
• Reverse if-conversion
Questions?