Update:
PBQP Register Allocation
PBQP Background

- Born out of DSP compiler research
- Easily support complex constraints
- Implemented in LLVM in version 2.4
Cost Model

For each variable, a 1D table describes costs:

<table>
<thead>
<tr>
<th>SPILL</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>0</td>
</tr>
<tr>
<td>R1</td>
<td>5</td>
</tr>
</tbody>
</table>
Cost Model

For pairs of variables... 2D tables describes costs:

<table>
<thead>
<tr>
<th>SPILL</th>
<th>R0</th>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPILL</th>
<th>R0</th>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>∞</td>
</tr>
</tbody>
</table>

A

B

50
0
5
Built-in Constraints

• Spill costs
• Interference
• Coalescing

You can add your own costs on top...
Use case 1

• An extremely small CPU:
  • 16-bits instructions set
  • 16 x 16-bits integer registers
• Many 4-register-operands domain specific instructions:
  \[
  \text{instr reg1, reg2, reg3, reg4}
  \]
• Can not be encoded in 16 bits, so need some \textit{pairing}:
  \[
  \text{instr reg(I), reg(J)[, reg(I+1), reg(J+1)]}
  \]
Use case 1: coalescing

- Pairing requires *different* registers
- But the coalescer will have happily coalesced registers in a pair if they hold the same value
- So the first step was to *smartly* undo the coalescer's work:
  - insert register copies --- only where really needed
  - do not forget to update liveness info!
Use case 1: constraints

• Obvious pairing constraint:
  \[ \text{reg}(l+1) \text{ is the successor of } \text{reg}(l) \]

• The pairing constraint is transitive:
  \[ \text{reg}(l) == \text{reg}(J+1) \Rightarrow \text{pair2}((\text{reg}(l+1), \text{reg}(J))) \]

• The constraint set must be complete ... or things will break
Recent Work

- Easier to use with composable constraints
- Solver improvements
  - Reductions in memory consumption
  - Reductions in compile time
  - Improvements in allocation quality
PBQP is 17% slower than Greedy
AArch64: execution time PBQP / Greedy

PBQP is 1% slower than Greedy
Try It Out

• Competitive code quality / performance
• Easy to customize
• Do not hesitate to talk to us:
  • Arnaud: arnaud.degrandmaison@arm.com
  • Lang: lhames@gmail.com
Iterative Register Allocation

Machine Function

Abstract Model

Rewrite

Abstract Solution

Build

Solve

Vertex Coloring

PBQP