Punctual Coalescing

Fernando Magno Quintão Pereira
Register Coalescing

- Register coalescing is an optimization on top of register allocation. The objective is to map both variables used in a copy instruction to the same register.

```c
int a = 0;
int b = a;
print(b);
```

```c
R1 = 0;
R2 = R1;
print(R2);
```

```c
int a = 0;
R1 = R1;
print(R1);
```
The Importance of Register Coalescing

Size reduction

Execution speedup

Energy saving
Punctual Coalescing

- *Register assignment* technique based on *puzzle solving*.
- It deals with *aliased registers*, e.g. x86.
  - No worries: I will define these concepts 😊
- It is $O(K*I)$, where $K$ = number of registers, and $I$ is number of instructions.
- **Strength**: very fast!
- **Weakness**: it is a local solution.
Register Aliasing

Two registers alias when an assignment to one may change the value of the other.

Ex. X86’s GP regs:

<table>
<thead>
<tr>
<th>AX</th>
<th>BX</th>
<th>CX</th>
<th>DX</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td>AL</td>
<td>BH</td>
<td>BL</td>
</tr>
<tr>
<td>CH</td>
<td>CL</td>
<td>DH</td>
<td>DL</td>
</tr>
</tbody>
</table>

Ex. PowerPC’s doubles:

<table>
<thead>
<tr>
<th>D0</th>
<th>D15</th>
</tr>
</thead>
<tbody>
<tr>
<td>F10</td>
<td>F15</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>F30</td>
<td>F31</td>
</tr>
</tbody>
</table>

And also ARM, UltraSparc, ST240, etc
Register Aliasing

• Register aliasing allows the register allocator to keep more variables in registers.
  – E.g.: two floats in the same register.

• But finding a register assignment that minimizes the number of registers taken is NP-complete.
  – Lee et al: Aliased register allocation for straight-line programs is NP-complete, TCS, 2008
Register allocation: fit the bars on the boxes.
Aliased register allocation: Bars may have different widths.
**Complexity result:** optimal aliased register allocation is NP-complete.

Program

\[ X_H = \bullet \]

\[ Y = \bullet \]

\[ X_L = \bullet \]

\[ Y_H = Y \]

\[ Z = X_L \]

\[ \bullet = X_H, Y_H, Z \]

Live Ranges

Registers
**Elementary form**: split variables between each pair of consecutive instructions.

Elementary Program

\[
\begin{align*}
a_1 &= \cdot \\
(a_2) &= (a_1) \\
B_2 &= \cdot \\
(a_3, B_3) &= (a_2, B_2) \\
c_3 &= \cdot \\
(a_4, B_4, c_4) &= (a_3, B_3, c_3) \\
d_4 &= B_4 \\
(a_5, c_5, d_5) &= (a_4, c_4, d_4) \\
E_5 &= c_5 \\
(a_6, c_6, E_6) &= (a_5, c_5, E_5) \\
\cdot &= a_6, d_6, E_6
\end{align*}
\]
**Complexity result:** aliased register allocation has polynomial time solution for elementary form programs.

### Elementary Program

\[
X_H = \cdot \\
Y = \cdot \\
X_L = \cdot \\
Y_H = Y \\
Y_L = X_H \\
X = X_L \\
\cdot = Y_L, Y_H, X
\]
Elementary Form = Slow RA?

• If a program has $O(V)$ variables and $O(I)$ instructions, then the elementary form program has $O(V*I)$ variables.

• If we assume that each instruction defines a variable, we have $O(V*I) = O(V^2)$.

• Many graph coloring based register allocators are $O(V^2)$.

• $O(V^4)$ is just too slow....

• We have a $O(K*I)$ approach, where $K$ is the number of registers!
Growth in number of variables

Data extracted from 4054 program traces taken from SPEC CPU 2000.
Our solution: traverse the program from top to bottom using the solution of previous puzzles to guide the solution of the next one to be solved.
Punctual Coalescing

• **Instance**: two consecutive puzzles, $p_1$ and $p_2$, such that $p_1$ is already solved.

• **Problem**: find a solution for $p_2$ that minimizes the number of copies inserted between $p_1$ and $p_1$. Puzzle $p_1$ is called the **guider**, and puzzle $p_2$ is called the **follower**.
Example

Guider

Follower

This solution has only three matches.

This solution is better: it has four matches.
Polynomial time Punctual Coalescing

• Punctual coalescing has a polynomial time solution, as long as the follower starts with an empty register board.
  – These are 89% of all the puzzles found in SPEC CPU 2000, when compiled to x86 using LLVM.
• Our solution is $O(K)$, where $K$ is the number of registers. For the whole program, $O(I*K)$.
• For the solution, see our paper: *Punctual Coalescing*. 
# Punctual ≠ Global

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a, b, c, d := -</td>
<td>a, b, c, d := -</td>
<td>a, b, c, d := -</td>
<td>a, b, c, d := -</td>
</tr>
<tr>
<td>- := b, d</td>
<td>- := b, d</td>
<td>- := b, d</td>
<td>- := b, d</td>
</tr>
<tr>
<td>E := -</td>
<td>E := -</td>
<td>E := -</td>
<td>E := -</td>
</tr>
<tr>
<td>- := E, a, c</td>
<td>- := E, a, c</td>
<td>- := E, a, c</td>
<td>- := E, a, c</td>
</tr>
</tbody>
</table>

**Global Coalescing**: find the register assignment that Minimizes the number of copies in the whole program. This problem is **NP-complete**.
Experiments

• We have compared four different coalescers:
  – A coalescing oblivious register allocator based on the coloring of chordal graphs.
  – The punctual coalescing heuristics used in register allocation by puzzle solving.
  – The optimal punctual coalescer.
  – An optimal solution for global coalescing, based on integer linear programming.

• Data: program traces from SPEC CPU 2000.
<table>
<thead>
<tr>
<th></th>
<th>Gzip</th>
<th>Vpr</th>
<th>Gcc</th>
<th>Mcf</th>
<th>Crafty</th>
<th>Parser</th>
<th>Gap</th>
<th>Vortex</th>
<th>Bzip2</th>
<th>Twolf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chordal (x 1000)</td>
<td>13,4</td>
<td>47,6</td>
<td>329,8</td>
<td>4,75</td>
<td>46,2</td>
<td>27,0</td>
<td>174,6</td>
<td>199,3</td>
<td>10,5</td>
<td>101,8</td>
</tr>
<tr>
<td>Heuristics</td>
<td>13</td>
<td>32</td>
<td>241</td>
<td>1</td>
<td>21</td>
<td>19</td>
<td>135</td>
<td>79</td>
<td>12</td>
<td>44</td>
</tr>
<tr>
<td>Punctual</td>
<td>0</td>
<td>10</td>
<td>17</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>33</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ILP</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Conclusion

• Punctual coalescing is a viable alternative for JIT compilers.
  – Has been implemented in LLVM with very competitive compilation times.

• Results are very good for program traces.

• We need a better algorithm for whole function optimization!
  – Slower compilation time, but better code quality.