Register Allocation by Puzzle Solving

Fernando M Q Pereira
Jens Palsberg

UCLA
University of California, Los Angeles

Presented at PLDI 2008
Call for papers and participation

• Jul 15, 2009: POPL submission deadline
  – Principles of Programming Languages
  – POPL will be in Madrid, Jan 20-22, 2010

• Aug 9-15, 2009 at UCLA:
  – LICS, IEEE Logic in Computer Science
  – SAS, Static Analysis Symposium
  – Six workshops
Three messages

• SSA-based register allocation for x86

• Register allocation = solving puzzles

• Split live ranges everywhere!
A compiler

source language

parser

intermediate representation

code generator

machine code
A better compiler

source language

parser

intermediate representation

code generator with a

register allocator

machine code
What is register allocation?

A = 10
B = 20
C = A + 30
Print C + 40 + B

Assume we have two registers
Reg. allocation = liveness analysis + graph coloring

A = 10

B = 20

C = A + 30

Print C + 40 + B

Interference graph:
A ——— B ——— C

With colors:
A ——— B ——— C
After register allocation

A = 10
B = 20
C = A + 30
Print C + 40 + B

R1 = 10
R2 = 20
R1 = R1 + 30
Print R1 + 40 + R2
Many models of register allocation

- Graph coloring
  - George & Appel TOPLAS ‘96, iterated register coalescing
  - Smith, Ramsey & Holloway, PLDI ‘04, aliased registers
- Linear scan
  - Poletto & Sarkar TOPLAS ‘99, excellent for JIT compilers
- Integer linear programming
  - Appel & George PLDI ‘01, optimal spilling
- Partitioned Boolean quadratic programming
  - Scholz & Eckstein SCOPES ‘02, optimal spilling
- Multi-commodity network flow
  - Koes & Goldstein PLDI ‘06, iterative within a time budget
Complications

- **Spill**: If there are not enough registers, then represent the remaining variables in memory.
- **Coalesce**: For an assignment $x=y$, try to map $x,y$ to the same register.
- **Live-range splitting**: Enable a variable to sometimes be in a register and sometimes in memory.
- **Pre-coloring**: Some instructions require it.
- **Register aliasing and pairing**: Found on x86, ARM, SPARC V8+V9, etc.
- **Rematerialization**: Don’t store, recompute!
SSA to the rescue

- Theorem: a program in strict SSA form has a chordal interference graph

- Enables two-phase register allocation:
  1. Spilling
  2. Coloring and coalescing

Goal of spilling: make
need-for-registers \leq \# registers

- If all the registers have the same size, then
need-for-registers = \text{size(largest clique)}
Comparison

Code Quality

SSA based

Linear Scan

Complexity of Register Allocator

Iterated Register Coalescing

Simple and Good!
Our goal

• **Our goal**: implement two-phase register allocation for x86 and handle pre-coloring, and register aliasing and pairing

• **Problem 1**: with pre-coloring, computing need-for-registers is NP-complete for unit interval graphs [Marx, 2006]

• **Problem 2**: with register aliasing and pairing, computing need-for-registers is NP-complete for interval graphs [Lee, Palsberg, Pereira, ICALP 2007]
Register allocation for elementary programs is equivalent to solving puzzles.

• Puzzles can be solved in polynomial time.
• The elementary program requires at most as many registers as the original program.
Program

\[\begin{align*}
  a &= \bullet \\
  B &= \bullet \\
  c &= \bullet \\
  d &= B \\
  E &= c \\
  \bullet &= a, d, E
\end{align*}\]

Live Ranges

\[\begin{align*}
  a \\
  c \\
  d \\
  B \\
  E
\end{align*}\]

Registers

\[\begin{align*}
  X_H & & X_L \\
  Y_H & & Y_L \\
  Z_H & & Z_L
\end{align*}\]
An optimal solution

Program

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

Live Ranges

\[ a \]
\[ c \]
\[ d \]
\[ B \]
\[ E \]

Registers

\[ X_H \]
\[ X_L \]
\[ Y_H \]
\[ Y_L \]
\[ Z_H \]
\[ Z_L \]
A new representation might help

Elementary Program

\[ a_1 = \bullet \]
\[(a_2) = (a_1) \]
\[ B_2 = \bullet \]
\[(a_3, B_3) = (a_2, B_2) \]
\[ c_3 = \bullet \]
\[(a_4, B_4, c_4) = (a_3, B_3, c_3) \]
\[ d_4 = B_4 \]
\[(a_5, c_5, d_5) = (a_4, c_4, d_4) \]
\[ E_5 = c_5 \]
\[(a_6, c_6, E_6) = (a_5, c_5, E_5) \]
\[ \bullet = a_6, d_6, E_6 \]

Live Ranges

Registers

\[ x_H x_L y_H y_L z_H z_L \]
And it does, indeed!

Assembly Program

\( X_H = \bullet \)

\( Y = \bullet \)

\( X_L = \bullet \)

\( Y_H = Y \)

\( Y_L = X_H \)

\( X = X_L \)

\( \bullet = Y_L, Y_H, X \)

Live Ranges

Registers

\( X_H \)

\( X_L \)

\( Y_H \)

\( Y_L \)

\( Z_H \)

\( Z_L \)
Old problem, new abstraction

Puzzle pieces

\[
\begin{align*}
a_1 &= \bullet \\
(a_2) &= (a_1) \\
B_2 &= \bullet \\
(a_3, B_3) &= (a_2, B_2) \\
c_3 &= \bullet \\
(a_4, B_4, c_4) &= (a_3, B_3, c_3) \\
d_4 &= B_4 \\
(a_5, c_5, d_5) &= (a_4, c_4, d_4) \\
E_5 &= c_5 \\
(a_6, c_6, E_6) &= (a_5, c_5, E_5) \\
\bullet &= a_6, d_6, E_6
\end{align*}
\]
Puzzle solving is the dual of graph coloring

- Graph coloring places the registers on the variables
  - The variables form a graph
  - The registers are colors for the graph

- Puzzle solving places the variables on the registers
  - The registers form a board
  - The variables are pieces for the board
## Two types of puzzles

<table>
<thead>
<tr>
<th>Type</th>
<th>Board</th>
<th>Pieces</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><img src="image1" alt="Board Type-0" /></td>
<td><img src="image2" alt="Pieces Type-0" /></td>
</tr>
<tr>
<td>K-1</td>
<td><img src="image3" alt="Board Type-0" /></td>
<td><img src="image4" alt="Pieces Type-0" /></td>
</tr>
<tr>
<td>1</td>
<td><img src="image5" alt="Board Type-1" /></td>
<td><img src="image6" alt="Pieces Type-1" /></td>
</tr>
<tr>
<td></td>
<td><img src="image7" alt="Board Type-1" /></td>
<td><img src="image8" alt="Pieces Type-1" /></td>
</tr>
</tbody>
</table>
PowerPC: 32 general purpose integer registers:

\[ R_0 \quad R_1 \quad R_2 \quad \cdots \quad R_{31} \]

ARM: 16 double precision floating-point registers:

\[ S_0 \quad S_1 \quad S_2 \quad S_3 \quad S_4 \quad S_5 \quad \cdots \quad S_{30} \quad S_{31} \]

\[ D_0 \quad D_1 \quad D_2 \quad D_{16} \]
Hybrid puzzle: type-0 and type-1.

<table>
<thead>
<tr>
<th>32 bits</th>
<th>EAX</th>
<th>EBX</th>
<th>ECX</th>
<th>EDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bits</td>
<td>AX</td>
<td>BX</td>
<td>CX</td>
<td>DX</td>
</tr>
<tr>
<td>8 bits</td>
<td>AH</td>
<td>AL</td>
<td>BH</td>
<td>BL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>32 bits</th>
<th>EBP</th>
<th>ESI</th>
<th>EDI</th>
<th>ESP</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bits</td>
<td>BP</td>
<td>SI</td>
<td>DI</td>
<td>SP</td>
</tr>
</tbody>
</table>
Split live ranges everywhere!
[Appel & George, PLDI 2001]

Basic block

\( \varphi \)

\( \pi \)

Parallel copy

Statement_1

Statement_2
From a program to an elementary program

\[ A = \cdot \]
\[ p_1: \]
\[ \text{jump } L_2, L_3 \]

\[ c = \cdot \]
\[ p_3: \]
\[ \text{jump } L_4 \]
\[ p_4: \]
\[ \text{jump } L_4 \]

\[ \text{joint } L_2, L_3 \]
\[ p_9: \]
\[ \cdot = c, A \]

\[ A_{01} = \cdot \]
\[ p_1: (A_1)=(A_{01}) \]
\[ [(A_2):L_2, (A_5):L_3]=\pi(A_1) \]

\[ c_{23} = \cdot \]
\[ p_3: (A_3,c_3)=(A_2,c_{23}) \]
\[ [(A_4,c_4):L_4]=\pi(A_3,c_3) \]

\[ A_{56} = \]
\[ p_6: (A_6,AL_6)=(A_5,AL_{56}) \]
\[ c_{67} = AL_6 \]
\[ p_7: (A_7,c_7)=(A_6,c_{67}) \]
\[ [(A_8,c_8):L_4]=\pi(A_7,c_7) \]

\[ p_9: (A_9,c_9)= \phi[(A_4,c_4):L_2, (A_8,c_8):L_3] \]
\[ \cdot = c_9, A_9 \]

\[ p_{10}: \]
\[ [()]:L_{\text{end}} = \pi() \]
From variables to puzzle pieces

variables

\[ p_x: \quad (C, d, E, f) = (C', d', E', f') \]
\[ A, b = C, d, E \]

\[ p_{x+1}: \quad (A'', b'', E'', f'') = (A, b, E, f) \]

live ranges

\[ A \quad b \quad C \quad d \quad E \quad f \]

p_x

p_{x+1}

puzzle pieces

A \quad b \quad C \quad d \quad E \quad f
From an elementary program to puzzles

\[ A_{01} = \circ \]

\[ P_1: (A_1) = (A_{01}) \]

\[ [(A_2): L_2, (A_3): L_3] = \pi(A_1) \]

\[ c_{23} = \circ \]

\[ P_3: (A_3, c_3) = (A_2, c_{23}) \]

\[ [(A_4, c_4): L_4] = \pi(A_3, c_3) \]

\[ AL_{56} = \circ \]

\[ P_6: (A_6, AL_6) = (A_5, AL_{56}) \]

\[ c_{67} = AL_6 \]

\[ P_7: (A_7, c_7) = (A_6, c_{67}) \]

\[ [(A_8, c_8): L_4] = \pi(A_7, c_7) \]

\[ p_9: (A_9, c_9) = \phi[(A_4, c_4): L_2, (A_8, c_8): L_3] \]

\[ \circ = c_9, A_9 \]

\[ P_{10}: \]

\[ [(\cdot): L_{end}] = \pi() \]
A linear algorithm for solving type-1 puzzles

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

( )

( )

( )

( )

( )

( )

( )

( )

( )

( )

( )

( )
Foundations

• **Theorem:** spill-free register allocation with pre-coloring for an elementary program is equivalent to solving a collection of puzzles

• **Theorem:** a puzzle is solvable if and only if our program succeeds on the puzzle

• **Theorem:** Our puzzle solving program runs in $O(#\text{regs})$ time
Two-phase register allocation for x86

1. Spilling:
   • The puzzle solver determines \text{MaxLive}
   • Remove pieces until \text{need-for-registers} = \#\text{registers}
   • We use Belady’s algorithm (also used in linear scan) for spilling

2. Coloring and coalescing:
   • Use a puzzle solver that is guided by the solution to the previous puzzle
   • If puzzle contain no pre-coloring, we can guarantee a minimal number of copies.
From puzzles to assembly code

Register bank

AX = •

p_1:
jump L_2, L_3

BL = •

p_2:

p_3:

xchg BX,AX
jump L_4

BX = AX

AL =

p_5:
P6:
AL = AL

p_7:
jump L_4

p_8:

p_9:

joint L_2, L_3

p_10:

= BL, AX

jump L_{end}
Experimental results comparing four allocators

- Puzzle solving
- Extended version of linear scan (default in LLVM)
- Iterated register coalescing by George & Appel, POPL 1996, with extensions by Smith, Ramsey & Holloway, PLDI 2004
- Partitioned boolean quadratic programs (PBQP): Scholz & Eckstein, LCTES/SCOPES 2002
- All four implemented in LLVM 1.9, running on an 32-bit X86
### Benchmarks: SPEC 2000, etc; total: 1.3 M LOC

**This talk: SPEC CPU 2000**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LoC</th>
<th>Asm/bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>224,099</td>
<td>12,868,208</td>
</tr>
<tr>
<td>perlbmk</td>
<td>85,814</td>
<td>7,010,809</td>
</tr>
<tr>
<td>gap</td>
<td>71,461</td>
<td>4,256,317</td>
</tr>
<tr>
<td>vortex</td>
<td>67,262</td>
<td>2,714,588</td>
</tr>
<tr>
<td>mesa</td>
<td>59,394</td>
<td>3,820,633</td>
</tr>
<tr>
<td>crafty</td>
<td>21,197</td>
<td>1,573,423</td>
</tr>
<tr>
<td>(nine more)</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Total</td>
<td>611,028</td>
<td>37,339,663</td>
</tr>
<tr>
<td>Benchmark</td>
<td>#puzzles</td>
<td>avg</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
<td>-----</td>
</tr>
<tr>
<td>gcc</td>
<td>476,649</td>
<td>1.03</td>
</tr>
<tr>
<td>perlbmk</td>
<td>265,905</td>
<td>1.03</td>
</tr>
<tr>
<td>gap</td>
<td>158,757</td>
<td>1.05</td>
</tr>
<tr>
<td>vortex</td>
<td>116,496</td>
<td>1.02</td>
</tr>
<tr>
<td>mesa</td>
<td>139,537</td>
<td>1.08</td>
</tr>
<tr>
<td>crafty</td>
<td>59,504</td>
<td>1.06</td>
</tr>
<tr>
<td>(nine more)</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Total</td>
<td>1,401,793</td>
<td>1.05</td>
</tr>
</tbody>
</table>
A variety of puzzles
Number of moves inserted per puzzle

Shorter is better

#(move/swaps) implementing (phi/pi)-nodes
#Internal Moves
Compilation time: extended linear scan vs. puzzles

Data normalized to puzzle solver

Shorter is better

<table>
<thead>
<tr>
<th>Program</th>
<th>Register Assignment Pass</th>
<th>Total comp</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>perlblk</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>gap</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>mesa</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>vortex</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>twolf</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>crafty</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>vpr</td>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>ammp</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>parser</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>gzip</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>bzip2</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>art</td>
<td>1.3</td>
<td>1.3</td>
</tr>
<tr>
<td>equake</td>
<td>2.4</td>
<td>2.4</td>
</tr>
<tr>
<td>mcf</td>
<td>1.7</td>
<td>1.7</td>
</tr>
<tr>
<td>Average</td>
<td>1.3</td>
<td>1.3</td>
</tr>
</tbody>
</table>
Execution time, normalized to GCC -O3

Run time normalized to gcc -O3

Shorter is better

Puzzle solver  ELS (LLVM's default)  EIRC  PBQP
Conclusion

• Register allocation by puzzle solving

• SSA-based register allocation for x86

• Handles pre-coloring, and register aliasing and pairing

• Split live ranges everywhere!

• Fast compilation time

• Competitive code quality
### Three types of puzzles

<table>
<thead>
<tr>
<th>Type</th>
<th>Board</th>
<th>Kinds of Pieces</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>...</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>K-1</td>
<td>X, Z</td>
</tr>
<tr>
<td>Type-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-1</td>
<td></td>
<td>Y, X, Y, X, Z</td>
</tr>
<tr>
<td>Type-2</td>
<td></td>
<td>Y, Y, Y, X, X, X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z, Z, Z</td>
</tr>
</tbody>
</table>
Elementary programs $\rightarrow$ elementary graphs

An elementary program has an elementary interference graph

\[
\begin{align*}
A_{01} &= \cdot \\
P_1: (A_1) &= (A_{01}) \\
[(A_2):L_2, (A_2):L_1] &= \text{pi}(A_2) \\
\end{align*}
\]

\[
\begin{align*}
c_{23} &= \cdot \\
P_3: (A_3, c_3) &= (A_2, c_{23}) \\
[(A_4, c_4):L_4] &= \text{pi}(A_3, c_3) \\
\end{align*}
\]

\[
\begin{align*}
A_{L56} &= \cdot \\
P_6: (A_6, A_{L6}) &= (A_5, A_{L56}) \\
c_{67} &= A_{L6} \\
P_7: (A_7, c_7) &= (A_6, c_{67}) \\
[(A_8, c_8):L_4] &= \text{pi}(A_7, c_7) \\
\end{align*}
\]

\[
\begin{align*}
p_9: (A_9, c_9) &= \text{phi}[(A_4, c_4):L_2, (A_9, c_9):L_3] \\
\cdot &= c_9, A_3 \\
P_{10}: \\
[() : \text{L_{end}}] &= \text{pi()} \\
\end{align*}
\]
Six classes of graphs

- Elementary Programs
- Unit interval graphs
- Interval graphs
- RDV-graphs
- Chordal graphs
- Clique substitutions of $P_3$

$\subset$

- SSI-form Programs
- SSA-form Programs

Elementary programs $\subset$ SSI-form programs $\subset$ SSA-form programs
## Complexity results for four graph problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>General</th>
<th>Chordal</th>
<th>Interval</th>
<th>Elementary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aligned 1-2-coloring extension</td>
<td>NP-complete</td>
<td>NP-complete</td>
<td>NP-complete</td>
<td>Linear time [this paper]</td>
</tr>
<tr>
<td>Aligned 1-2-coloring</td>
<td>NP-complete</td>
<td>NP-complete</td>
<td>NP-complete</td>
<td>Linear time [this paper]</td>
</tr>
<tr>
<td>Coloring extension</td>
<td>NP-complete</td>
<td>NP-complete</td>
<td>NP-complete</td>
<td>Linear time [this paper]</td>
</tr>
<tr>
<td>Coloring</td>
<td>NP-complete</td>
<td>Linear time</td>
<td>Linear time</td>
<td>Linear time</td>
</tr>
</tbody>
</table>