Puzzle solving

Simone Campanoni
simone.campanoni@northwestern.edu
Materials

• Research paper:
  • Authors: Fernando Magno Quintao Pereira, Jens Palsberg
  • Title: Register Allocation by Puzzle Solving
  • Conference: PLDI 2008

• Ph.D. thesis
  • Author: Fernando Magno Quintao Pereira
  • Title: Register Allocation by Puzzle Solving
  • UCLA 2008
Register Allocation

A. Spill all variables
B. Puzzle solving
C. Linear scan
D. Graph coloring
E. Integer linear programming

... in significantly less time!
Outline

• Register allocation abstractions

• From a program to a collection of puzzles

• Solve puzzles

• From solved puzzles to assembly code
A graph-coloring register allocator

Register allocator

Assign colors

Interferences

Code generation

Interference graph, \( f \)

Interference graph colored, \( f \)

IN, OUT

Liveness analysis

Interferences

Graph coloring

Code analysis

Spill

\( f \) with var spilled

\( f \) without variables and with registers

In this class:
• All variables have the same type
• A register can store any variable
Graph coloring abstraction: a problem

```plaintext
(@MyVeryImportantFunction
  0
  MyVar1 <- 2
  MyVar2 <- 40
  MyVar3 <- 0
  MyVar3 += MyVar1
  MyVar3 += MyVar2
  print MyVar3
  return)
```

Can this be obtained by the graph-coloring algorithm you learned in this class?

**Register aliasing**
- r8 can store either one 64-bit value or two 32-bit values
- r9 can store 64 bit values

**Variables**
- MyVar1: 64 bits
- MyVar2: 32 bits
- MyVar3: 32 bits
Puzzle Abstraction

• Puzzle = board (1 area = 1 register) + pieces (variables)

• Pieces cannot overlap

• Some pieces are already placed on the board

• Task: fit the remaining pieces on the board (register allocation)
From register file to puzzle boards

• Every area of a puzzle is divided in two rows (soon will be clear why)

• Registers determine the shape of the puzzle board
  Register aliasing determines the #columns

PowerPC
ARM integer registers
From register file to puzzle boards

- Every area of a puzzle is divided into two rows (soon will be clear why).
- Registers determine the shape of the puzzle board.
- Register aliasing determines the number of columns.

### PowerPC
- ARM integer registers

### SPARC v8
- ARM float registers

### SPARC v9
- 8 quad-precision floating point registers

---

- x86, 8 integer registers, AX=EAX, SI=ESI, etc.
Puzzle pieces accepted by boards

<table>
<thead>
<tr>
<th>Type</th>
<th>Board</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>K-1</td>
</tr>
<tr>
<td>Type-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Type-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><img src="image3.png" alt="Diagram" /></td>
<td><img src="image4.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Type-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><img src="image5.png" alt="Diagram" /></td>
<td><img src="image6.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Our class ->
Outline

• Register allocation abstractions

• From a program to a collection of puzzles

• Solve puzzles

• From solved puzzles to assembly code
From a program to puzzle pieces

1. Convert a program into an *elementary program*
   A. Transform code into SSA form

2. Map the elementary program into puzzle pieces
Static Single Assignment (SSA) representation

• A variable is set only by one instruction in the function body
  myVar1 <- 5
  myVar2 <- 7
  myVar3 <- 42

• A static assignment can be executed more than once
SSA and not SSA example

float myF (float par1, float par2, float par3){
    return (par1 * par2) + par3; }

float myF(float par1, float par2, float par3) {
    myVar1 = par1 * par2
    myVar1 = myVar1 + par3
    ret myVar1}

float myF(float par1, float par2, float par3) {
    myVar1 = par1 * par2
    myVar2 = myVar1 + par3
    ret myVar2}
What about joins?

• Add $\Phi$ functions/nodes to model joins
  • One argument for each incoming branch

• Operationally
  • selects one of the arguments based on how control flow reach this node

• At code generation time, need to eliminate $\Phi$ nodes

```
If (b > N)
    b = c + 1
    b = d + 1
```

Not SSA

```
If (?) > N
    b1 = c + 1
    b2 = d + 1
```

Still not SSA

```
b3 = \Phi(b1, b2)
If (b3 > N)
    b1 = c + 1
    b2 = d + 1
```

SSA
Eliminating $\Phi$

- Basic idea: $\Phi$ represents facts that value of join may come from different paths
  - So just set along each possible path

\[
\begin{align*}
b_1 &= c + 1 \\
b_2 &= d + 1 \\
\text{If } (b_3 > N) \\
b_3 &= \Phi(b_1, b_2)
\end{align*}
\]

\[
\begin{align*}
b_1 &= c + 1 \\
b_3 &= b_1 \\
\text{If } (b_3 > N) \\
b_2 &= d + 1 \\
b_3 &= b_2
\end{align*}
\]

Not SSA
Eliminating $\Phi$ in practice

• Copies performed at $\Phi$ may not be useful
• Joined value may not be used later in the program
  (So why leave it in?)

• Use dead code elimination to kill useless $\Phi$s
• Register allocation maps the variables to machine registers
From a program to puzzle pieces

1. Convert a program into an *elementary program*
   A. Transform code into SSA form
   B. Transform A into SSI form

2. Map the elementary program into puzzle pieces
Static Single Information (SSI) form

In a program in SSI form:

• Every basic block ends with a $\pi$-function that renames the variables that are alive going out of the basic block

```
If (b > 1)
    ... = c + 1  ... = c * 2
```

Not SSI

```
If (b > 1)
    (c1, c2) = $\pi$(c)
    ... = c1 + 1  ... = c2 * 2
```

SSI
SSA and SSI code

If \( b > 1 \)

\[ \begin{align*}
&b = d + 1 \\
&b = d + 4 \\
&b_3 = \Phi(b_1, b_2) \\
&\text{If } (b_3 > 1) \\
&c_1 = c + 1 \\
&c_2 = c \times 2 \\
&b_1 = d_1 + 1 \\
&b_2 = d_2 + 4 \\
&\text{SSA and SSI}
\end{align*} \]

Not SSA and not SSI

SSA but not SSI

SSA and SSI
From a program to puzzle pieces

1. Convert a program into an *elementary program*
   A. Transform code into SSA form
   B. Transform A into SSI form
   C. Insert in B parallel copies between every instruction pair

2. Map the elementary program into puzzle pieces
Parallel copies

- Rename variables in parallel

\[
\begin{align*}
V &= X + Y \\
Z &= A + B
\end{align*}
\]

\[
(V_1, X_1, Y_1, Z_1, A_1, B_1) = (V, X, Y, Z, A, B)
\]

\[
V_1 = X_1 + Y_1
\]

\[
(V_2, X_2, Y_2, Z_2, A_2, B_2) = (V_1, X_1, Y_1, Z_1, A_1, B_1)
\]

\[
Z_2 = A_2 + B_2
\]
From a program to puzzle pieces

1. Convert a program into an *elementary program*
   A. Transform code into SSA form
   B. Transform A into SSI form
   C. Insert in B parallel copies between every instruction pair

We have obtained an elementary program!
Elementary form: an example

(a) 

L_1
A = •
p_1:
branch L_2, L_3

L_2

L_3

L_4

join L_2, L_3

p_2:
p_5:

p_3:
jump L_4

p_4:
p_8:

p_9:
• = c, A
p_10:
jump L_{end}

(b) 

L_1
A_{01} = •
p_1: (A_1) = (A_{01})
p_{2,5}: [(A_2):L_2, (A_5):L_3] = \pi (A_1)

L_2

L_3

AL = •
p_6:
c = AL
p_7:
jump L_4

p_6: (A_6, AL_6) = (A_5, AL_{56})
c_{67} = AL_6
p_7: (A_7,c_7) = (A_6,c_{67})
p_8: [(A_8,c_8):L_4] = \pi(A_7,c_7)

L_4
p_9: (A_9, c_9) = \Phi[(A_4, c_4):L_2, (A_8, c_8):L_3]
• = c_9, A_9
p_10: () = 0
p_11: [():L_{end}] = \pi()
From a program to puzzle pieces

1. Convert a program into an elementary program
   A. Transform code into its SSA form
   B. Transform code into its SSI form
   C. Insert parallel copies between every instruction pair

2. Map the elementary program into puzzle pieces
Add puzzle boards

\[
L_1
\]
\[
A_{01} = \bullet
\]
\[
p_1: (A_1) = (A_{01})
\]
\[
p_{2,5}: [(A_2):L_2, (A_5):L_3] = \pi(A_1)
\]
\[
p_0: [\emptyset: L_1] = \pi()
\]

\[
L_2
\]
\[
c_{23} = \bullet
\]
\[
p_3: (A_3,c_3) = (A_2,c_{23})
\]
\[
p_4: [(A_4,c_4):L_4] = \pi(A_3,c_3)
\]

\[
L_3
\]
\[
A_{56} = \bullet
\]
\[
p_6: (A_6, A_{56}) = (A_5, A_{56})
\]
\[
c_{67} = A_{56}
\]
\[
p_7: (A_7,c_7) = (A_6,c_{67})
\]
\[
p_8: [(A_8,c_8):L_4] = \pi(A_7,c_7)
\]

\[
L_4
\]
\[
p_9: (A_9,c_9) = \Phi[(A_4,c_4):L_2, (A_8,c_8):L_3]
\]
\[
= c_9, A_9
\]
\[
p_{10}: () = ()
\]
\[
p_{11}: [\emptyset: L_\text{end}] = \pi() \]

The board:

\[
\begin{array}{cc}
AX & BX \\
\hline
AH & AL \\
BH & BL \\
\end{array}
\]

\[
\begin{array}{cccc}
p_0 & p_2 & p_3 & p_4 \\
| & | & | & | \\
p_5 & p_6 & p_7 & p_8 \\
| & | & | & | \\
p_0 & p_1 & & \\
\end{array}
\]
Generating puzzle pieces

- For each instruction i
  - Create one puzzle piece for each live-in and live-out variable
  - If the live range ends at i, then the puzzle piece is X
  - If the live range begins at i, then Z-piece
  - Otherwise Y-piece

\[
V_1 \text{ (used later)} = V_2 \text{ (last use)} + 3
\]
\[
r_{10} = r_{10} + 3
\]
Example

\[ A_{01} = \cdot \]
\[ p_1: (A_1) = (A_{01}) \]
\[ p_{2,5}: [(A_2):L_2, (A_5):L_3] = \pi(A_1) \]

\[ c_{23} = \]
\[ p_3: (A_3, c_3) = (A_2, c_{23}) \]
\[ p_4: [(A_4, c_4):L_4] = \pi(A_3, c_3) \]

\[ A_{L56} = \cdot \]
\[ p_6: (A_6, A_{L56}) = (A_5, A_{L56}) \]
\[ c_{67} = A_{L56} \]
\[ p_7: (A_7, c_7) = (A_6, c_{67}) \]
\[ p_8: [(A_8, c_8):L_4] = \pi(A_7, c_7) \]

\[ p_9: (A_9, c_9) = \Phi[(A_4, c_4):L_2, (A_8, c_8):L_3] \]
\[ \cdot = c_9, A_9 \]
\[ p_{10}: () = () \]
\[ p_{11}: [():L_{\text{end}}] = \pi() \]

**Variables**
\[ p_x: (C, d, E, f, g) = (C', d', E', f') \]
\[ A, b = C, d, E \]
\[ p_{x+1}: (A'', b'', E'', f'', g'') = (A, b, E, f) \]

**Live Ranges**

**Pieces**

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

\[ A_{01} = \bullet \]
\[ p_1: (A_1) = (A_{01}) \]
\[ p_{2,5}: [(A_2):L_2, (A_5):L_3] = \pi(A_1) \]
\[ p_0: [(\):L_1] = \pi() \]

\[ c_{23} = \]
\[ p_3: (A_3, c_3) = (A_2, c_{23}) \]
\[ p_4: [(A_4, c_4):L_4] = \pi(A_3, c_3) \]

\[ \text{AL}_{56} = \bullet \]
\[ p_6: (A_6, \text{AL}_{6}) = (A_5, \text{AL}_{56}) \]
\[ c_{67} = \text{AL}_{5} \]
\[ p_7: (A_7, c_7) = (A_6, c_{67}) \]
\[ p_8: [(A_8, c_8):L_4] = \pi(A_7, c_7) \]

\[ p_9: (A_9, c_9) = \Phi[(A_4, c_4):L_2, (A_8, c_8):L_3] \]
\[ = c_9, A_9 \]
\[ p_{10}: () = () \]
\[ p_{11}: [(\):L_{end}] = \pi() \]

The board:

\[ \text{AX} \]
\[ \text{BX} \]

\[ p_0 \]
\[ p_1 \]

\[ p_2 \]
\[ p_3 \]

\[ p_5 \]
\[ p_6 \]

\[ p_7 \]
\[ p_8 \]
Outline

• Register allocation abstractions

• From a program to a collection of puzzles

• Solve puzzles

• From solved puzzles to assembly code
Solving type 1 puzzles

• Approach proposed: complete one area at a time

• For each area:
  • Pad a puzzle with size-1 X- and Z-pieces until the area of puzzle pieces == board

  Board with 1 pre-assigned piece

  Padding

• Solve the puzzle
Solving type 1 puzzles: a visual language

Puzzle solver -> Statement+
Statement -> Rule | Condition
Condition -> (Rule : Statement)

Rule ->

- Rule = how to complete an area
- Rule composed by pattern:
  what needs to be already filled (match/not-match an area)
- strategy:
  what type of pieces to add and where
- A rule $r$ succeeds in an area $a$ iff
  i. $r$ matches $a$ and
  ii. pieces of the strategy of $r$ are available
Solving type 1 puzzles: a visual language

Puzzle solver -> Statement+
Statement -> Rule | Condition
Condition -> (Rule : Statement)

Rule ->

Puzzle solver success

• A program succeeds iff all statements succeeds

• A rule \( r \) succeeds in an area \( a \) iff
  1. \( r \) matches \( a \)
  2. pieces of the strategy of \( r \) are available

• A condition \((r : s)\) succeeds iff
  1. \( r \) succeeds or
  2. \( s \) succeeds
  3. All rules of a condition must have the same pattern
Solving type 1 puzzles: a visual language

Puzzle solver -> Statement+
Statement -> Rule | Condition
Condition -> (Rule : Statement)

Puzzle solver execution
○ For each statement $s_1, \ldots, s_n$
  
  ❖ For each area $a$ such that the pattern of $s_i$ matches $a$
    
    □ Apply $s_i$ to $a$
    □ If $s_i$ fails, terminate and report failure
Program execution: an example

• A puzzle solver

s1
X X
Z

s2
X
Z
Y
Z

1. s1 matches a1 only
2. Apply s1 to a1 succeeds and returns this puzzle

• Puzzle

a1
R8
X K
Q
a2
r9
K X
Q Y

Y Z

3. s2 matches a2 only
4. Apply s2 to a2
   A. Apply first rule of s2: fails
   B. Apply second rule of s2: success

Puzzle solved!
Program execution: another example

- A puzzle solver

Puzzle

1. $s_1$ matches $a_1$ only
   
   A. Apply first rule of $s_1$: success

2. Apply $s_1$ to $a_1$

3. $s_2$ matches $a_2$ and $a_3$

4. Apply $s_2$ to $a_2$

5. Apply $s_2$ to $a_3$

Puzzle solved!
Program execution: yet another example

• A puzzle solver

\[
\begin{array}{c}
\text{s1} \\
\begin{array}{cc}
X & X \\
\hline & X \\
\end{array} \\
\text{s2} \\
\begin{array}{cc}
X & \_ \\
\hline & Y \\
\end{array}
\end{array}
\]

• Puzzle

\[
\begin{array}{ccc}
a1 & a2 & a3 \\
\hline & & \\
\begin{array}{c}
x_1 \\
\hline x_2 \\
x_3 \\
\hline y_1 \\
y_2
\end{array}
\end{array}
\]

Finding the right puzzle solver is the key!

1. s1 matches a1 only
2. Apply s1 to a1
   A. Apply first rule of s1: success

   \[
   \begin{array}{ccc}
a1 & a2 & a3 \\
\hline
\begin{array}{c}
x_1 & x_2 \\
\hline x_3 \\
\end{array} & \\
\begin{array}{c}
\_ & \_ \\
\hline \_ & \_ \\
\end{array}
\end{array}
\]

3. s2 matches a2 and a3
4. Apply s2 to a2: fail
   No 1-size x pieces, we used them all in s1
Solution to solve type 1 puzzles

Theorem: a type-1 area is solvable iff this program succeeds

Wait, ... did we just solve an NP problem in polynomial time?

Register allocation: complete all areas

Simplified problem solved: complete one area at a time
Solution to solve type 1 puzzles: complexity

For one instruction in P:
- Application of a rule to an area: $O(1)$
- A puzzle solver $O(1)$ rules on each area of a board
- Execution of a puzzle solver on a board with $K$ areas takes $O(K)$ time

Corollary 3.
Spill-free register allocation with pre-coloring for an elementary program $P$ and $K$ registers is solvable in $O(|P| \times K)$ time
### Solving type 0 puzzles

<table>
<thead>
<tr>
<th>Type-0</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></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z</td>
</tr>
<tr>
<td>K-1</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type-1</th>
<th>Board</th>
<th>Kinds of Pieces</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type-2</th>
<th>Board</th>
<th>Kinds of Pieces</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z</td>
</tr>
</tbody>
</table>
Solving type 0 puzzles: algorithm

- Place all Y-pieces on the board
- Place all X- and Z-pieces on the board
Spilling

• If the algorithm to solve a puzzle fails i.e., the need for registers exceeds the number of available registers => spill

• Observation: translating a program into its elementary form creates families of variables, one per original variable

• To spill:
  • Choose a variable $v$ to spill from the original program
  • Spill all variables in the elementary form that belong to the same family of $v$
Outline

• Register allocation abstractions

• From a program to a collection of puzzles

• Solve puzzles

• From solved puzzles to assembly code
From solved puzzles to assembly code
From solved puzzles to assembly code
Thank you!

Compilation time

Generated code run time

A

C

D

E

This lecture → B

Equivalent quality of graph coloring

... in significantly less time!

Ideal

Compilation time

Thank you!
Always have faith in your ability

Success will come your way eventually

Best of luck!