Simone Campanoni
simonec@eecs.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

Equivalent quality of graph coloring... in significantly less time!

Generated-code run time

Compilation time

Ideal
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

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

Can this be obtained by the graph-coloring algorithm you learned in this class?
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></th>
<th>Board</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Type-0 board 0" /></td>
</tr>
<tr>
<td>Type-1</td>
<td><img src="image" alt="Type-1 board" /></td>
</tr>
<tr>
<td>Type-2</td>
<td><img src="image" alt="Type-2 board" /></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 ($b > N$)

$b = c + 1$

$b = d + 1$

If ($? > N$)

$b1 = c + 1$

$b2 = d + 1$

If ($b3 > N$)

$b3 = \Phi(b1, b2)$

Not SSA

Still not SSA

SSA
Eliminating $\Phi$

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

\begin{align*}
b1 &= c + 1 \\
b2 &= d + 1 \\
\text{If } (b3 > N) &
\end{align*}

\text{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 π-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) = π(c)
  ...
  ... = c1 + 1
  ...
  ... = c2 * 2
```

SSI
SSA and SSI code

If (b > 1)

... = c + 1
... = c * 2

If (b3 > 1)
(c1, c2) = π(c)

... = c + 1
... = c * 2

b3 = Φ(b1, b2)
b1 = d1 + 1
b2 = d2 + 4

... = c1 + 1
... = c2 * 2

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

V = X + Y
Z = A + B

(V1, X1, Y1, Z1, A1, B1) = (V, X, Y, Z, A, B)
V1 = X1 + Y1
(V2, X2, Y2, Z2, A2, B2) = (V1, X1, Y1, Z1, A1, B1)
Z2 = A2 + B2
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
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

L1
\[
\begin{align*}
A_{01} &= \cdot \\
p_1: (A_1) &= (A_{01}) \\
\end{align*}
\]
\[
p_0: [\emptyset]:L_1 = \pi()
\]

L2
\[
\begin{align*}
c_{23} &= \\
p_3: (A_3, c_3) &= (A_2, c_{23}) \\
p_4: [(A_4, c_4):L_4] &= \pi(A_3, c_3)
\end{align*}
\]

L3
\[
\begin{align*}
A_{L56} &= \cdot \\
p_6: (A_6, A_{L6}) &= (A_5, A_{L56}) \\
c_{67} &= A_{L6} \\
p_7: (A_7, c_7) &= (A_6, c_{67}) \\
p_8: [(A_8, c_8):L_4] &= \pi(A_7, c_7)
\end{align*}
\]

L4
\[
\begin{align*}
p_9: (A_9, c_9) &= \Phi[(A_4, c_4):L_2, (A_8, c_8):L_3] \\
\cdot &= c_9, A_9 \\
p_{10}: () &= () \\
p_{11}: [\emptyset]:L_{\text{end}} &= \pi()
\end{align*}
\]
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

\[
\begin{align*}
L_1 & : A_{01} = \cdot \\
p_1: (A_1) = (A_{01}) & \\
p_{2,5}: [(A_2):L_2, (A_3):L_3] = \pi(A_1) \quad p_0: [()L_1] = \pi()
\end{align*}
\]

\[
\begin{align*}
L_2 & : c_{23} = \\
p_3: (A_3, c_3) = (A_2, c_{23}) & \\
p_4: [(A_4, c_4):L_4] = \pi(A_3, c_3)
\end{align*}
\]

\[
\begin{align*}
L_3 & : AL_{56} = \cdot \\
p_6: (A_6, AL_{56}) = (A_5, AL_{56}) & \\
c_{67} = AL_6 & \\
p_7: (A_7, c_{7}) = (A_6, c_{67}) & \\
p_8: [(A_8, c_8):L_4] = \pi(A_7, c_7)
\end{align*}
\]

\[
\begin{align*}
L_{4} & : p_9: (A_9, c_9) = \Phi[(A_4, c_4):L_2, (A_8, c_8):L_3] \\
& \quad \cdot = c_9, A_9 \\
p_{10}: () = () & \\
p_{11}: [()L_{\text{end}}] = \pi()
\end{align*}
\]

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:
- \( p_x \downarrow A \quad b \\ C \quad d \quad E \quad f \)
- \( p_{x+1} \downarrow X \quad Z \quad Y \)

Pieces:
- \( C \quad d \quad E \quad f \)
- \( A \quad b \)
Example

\begin{align*}
A_{01} = \bullet \\
p_1: (A_1) = (A_{01}) \\
p_{2,5}: [(A_2):L_2, (A_5):L_3] = \pi(A_1)
\end{align*}

\begin{align*}
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)
\end{align*}

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

\begin{align*}
p_9: (A_9, c_0) = \Phi [(A_4, c_4):L_2, (A_8, c_8):L_3] \\
\quad = c_0, A_9 \\
p_{10}: () = () \\
p_{11}: [():L_{end}] = \pi() 
\end{align*}
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

• 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

<table>
<thead>
<tr>
<th>Puzzle solver -&gt; Statement+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement -&gt; Rule</td>
</tr>
<tr>
<td>Condition -&gt; (Rule : Statement)</td>
</tr>
<tr>
<td>Rule -&gt;</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>Z</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>Y</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>Y</td>
</tr>
<tr>
<td>Y</td>
</tr>
</tbody>
</table>

**Puzzle solver success**

- A program succeeds iff all statements succeeds
- A rule $r$ succeeds in an area $a$ iff
  - $r$ matches $a$
  - pieces of the strategy of $r$ are available
- A condition ($r : s$) succeeds iff
  - $r$ succeeds or
  - $s$ succeeds
  - 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

\[
\begin{array}{ccc}
X & X \\
Z & & \\
\hline
s1 & & \\
\end{array} \quad \begin{array}{ccc}
X & Z \\
Y & Z \\
\hline
s2 & & \\
\end{array}
\]

(• Puzzle)

1. s1 matches \(a_1\) only
2. Apply \(s_1\) to \(a_1\) succeeds and returns this puzzle

\[
\begin{array}{ccc}
X & K \\
Q & & \\
\hline
a_1 & & \\
\end{array} \quad \begin{array}{ccc}
Y \\
Z & & \\
\hline
\end{array}
\]

3. \(s_2\) matches \(a_2\) only
4. Apply \(s_2\) to \(a_2\)
   A. Apply first rule of \(s_2\): fails
   B. Apply second rule of \(s_2\): success

Puzzle solved!
Program execution: another example

• A puzzle solver

1. $s_1$ matches $a_1$ only

2. Apply $s_1$ to $a_1$
   A. Apply first rule of $s_1$: success

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

4. Apply $s_2$ to $a_2$

5. Apply $s_2$ to $a_3$
Program execution: yet another example

• A puzzle solver

• Puzzle

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

2. Apply s1 to a1

3. s2 matches a2 and a3

4. Apply s2 to a2: fail
   No 1-size x pieces, we used them all in s1

Finding the right puzzle solver is the key!
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

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

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
### Solving type 0 puzzles

<table>
<thead>
<tr>
<th>Type</th>
<th>Board</th>
<th>Kinds of Pieces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 0</td>
<td>0</td>
<td>Y, X, Z</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>K-1</td>
<td>Y, X, Z</td>
</tr>
</tbody>
</table>

- **Type-0** board consists of a sequence of quadrants, where each quadrant can contain one of the pieces Y, X, or Z.

<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, X, Y, Z</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y, X, Y, Z</td>
</tr>
</tbody>
</table>

- **Type-1** board is more complex, with a grid of quadrants arranged in a 2x2 pattern, allowing for a variety of piece configurations.

<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, X, X, Z</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y, X, X, Z</td>
</tr>
</tbody>
</table>

- **Type-2** board is the most intricate, with a grid of quadrants arranged in a larger 3x3 pattern, accommodating multiple piece combinations.

Each type of puzzle has a unique board configuration and piece arrangement, allowing for different solving strategies.
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 puzzles 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

The board:

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

\[
L_1: \quad \text{AX} = \cdot \\
\hspace{1cm} p_0: \quad \text{branch } L_2, L_3
\]

\[
L_2: \quad \text{BL} = \cdot \\
\hspace{1cm} p_3: \quad \text{cxhr } BX, AX \\
\hspace{1cm} p_5: \quad AL = AL \\
\hspace{1cm} p_6: \quad AL = AL \\
\hspace{1cm} p_7: \quad AL = \text{BL} \\
\hspace{1cm} p_8: \quad \text{jump } L_4
\]

\[
L_3: \quad p_{10}: \quad AL, BX \\
\hspace{1cm} p_0: \quad \text{jump } L_{\text{end}}
\]
This lecture

Ideal

Thank you!