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
A compiler

Character stream (Source code)

Front-end

IR

Middle-end

IR

Back-end

Machine code
Task: From Variables to Registers

```
(:MyVeryImportantFunction
  (MyVar1 <- 2)
  (MyVar2 <- 40)
  (MyVar3 <- MyVar1)
  (MyVar3 += MyVar2)
  (print MyVar3)
)
```

No overlapping

Software

- MyVar1
- MyVar2
- MyVar3

Hardware

- r8
- r9
- r10

?
Register Allocation

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

... in significantly less time!

Generated-code run time

Compilation time

Equivalent quality of graph coloring

Ideal
Summary

• Graph coloring abstraction: Houston we have a problem

• Puzzle abstraction

• From a program to a collection of puzzles

• Solve puzzles

• From solved puzzles to assembly code
To register allocators: what are you doing?

(:MyVeryImportantFunction

(MyVar1 <- 2)
(MyVar2 <- 40)
(MyVar3 <- 0)
(MyVar3 += MyVar1)
(MyVar3 += MyVar2)
(print MyVar3)
)

- MyVar1 -> stack (spilled)
- MyVar2 -> r8
- MyVar3 -> r9
Graph coloring abstraction: a problem

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

• Graph coloring abstraction: Houston we have a problem

• Puzzle abstraction

• From a program to a collection of puzzles

• Solve puzzles

• From solved puzzles to assembly code
Puzzle Abstraction

- Puzzle = board (areas = registers) + pieces (variables)
  
  ![Diagram of board with pieces]

- 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 puzzle board has areas divided in two rows (soon will be clear why)

• A register determines the shape of the corresponding puzzle board.

Register aliasing determines the #columns

PowerPC
ARM integer registers

SPARC v8
ARM float registers

SPARC v9

x86, 8 integer registers, AX=EAX, SI=ESI, etc

SPARC V9, 8 quad-precision floating point registers
## Puzzle pieces accepted by boards

<table>
<thead>
<tr>
<th>Type</th>
<th>Board</th>
<th>Kinds of Pieces</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><img src="image" alt="Board 0" /></td>
<td><img src="image" alt="Pieces 0" /></td>
</tr>
<tr>
<td>...</td>
<td><img src="image" alt="Board K-1" /></td>
<td><img src="image" alt="Pieces 0" /></td>
</tr>
<tr>
<td><strong>Type-1</strong></td>
<td><img src="image" alt="Board Type-1" /></td>
<td><img src="image" alt="Pieces Type-1" /></td>
</tr>
<tr>
<td><strong>Type-2</strong></td>
<td><img src="image" alt="Board Type-2" /></td>
<td><img src="image" alt="Pieces Type-2" /></td>
</tr>
</tbody>
</table>
Summary

• Graph coloring abstraction: Houston we have a problem

• Puzzle abstraction

• 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
   B. Transform A into SSI form
   C. Insert in B parallel copies between every instruction pair

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}  

NOT SSA

SSA
Motivation for SSA

• Code analysis needs to represent facts at every program point

```c
float myF(float par1, float par2, float par3) {
    myVar1 = par1 * par2
    myVar2 = myVar1 + par3
    ret myVar2 }
```

• What if
  • There are a lot of facts and there are a lot of program points?
  • potentially takes a lot of space/time
Example

\[
\begin{align*}
x &= 3 \\
a > b \quad &\text{Predecessor} \\
y &= a + b \quad &\text{Successor} \\
z &= 2 \times y \\
w &= y + z \\
z &= w + x
\end{align*}
\]
Static Single Assignment (SSA)

Add **SSA edges** from definitions to uses
- No intervening statements define variable
- Safe to propagate facts about $x$ only along SSA edges
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

![Diagram showing the calculations and SSA rules](image)
Eliminating $\Phi$

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

\[
b_1 = c + 1 \quad b_2 = d + 1
\]

\[
b_3 = \Phi(b_1, b_2)
\]

If ($b_3 > N$)

\[
b_1 = c + 1
\]

\[
b_2 = d + 1
\]

\[
b_3 = b_1
\]

\[
b_3 = b_2
\]

If ($b_3 > N$)

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
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 live going out of the basic block.

---

**Basic block**: sequence of instructions with
- **only one** entry point and
- **only one** exit point.

**Not SSI**

BB1:
- L1
  - (myVar1 <- 5)
  - (myVar2 += myVar1)
  - (cjump myVar1 = myVar2 :L2)

BB2:
- L2
  - (c <- 10)

**SSI**

- \( \text{If } (b > 1) \)
- \( (c1, c2) = \pi(c) \)
- \( ... = c1 + 1 \)
- \( ... = c2 * 2 \)

- \( \text{If } (b > 1) \)
- \( \text{... } \)
- \( \text{... } \)
- \( \text{... } \)
SSA and SSI code

Not SSA and not SSI

SSA but not SSI

SSA and SSI
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
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

\begin{align*}
A_{01} &= \cdot \\
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*}
A_{L_{56}} &= \cdot \\
p_6: (A_6, A_{L_{56}}) &= (A_5, A_{L_{56}}) \\
c_{67} &= A_{L_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*}
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}: 0 &= 0 \\
p_{11}: [0]:L_{end} &= \pi(0)
\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

V1 (used later) = V2 (last use) + 3
r10 = r10 + 3
Example

L₁

$A_{01} = \cdot$
$p₁: (A₁) = (A_{01})$
$p₂,₅: [(A₂):L₂, (A₅):L₃] = \pi (A₁)$

L₂

$c_{23} =$
$p₃: (A₃,c₃) = (A₂,c_{23})$
$p₄: [(A₄,c₄):L₄] = \pi (A₃,c₃)$

L₃

$AL_{5₆} = \cdot$
$p₆: (A₆, AL₆) = (A₅, AL_{5₆})$
$c₆₇ = AL₆$
$p₇: (A₇,c₇) = (A₆,c₆₇)$
$p₈: [(A₈,c₈):L₄] = \pi (A₇,c₇)$

L₄

$p₉: (A₉, c₉) = \Phi [(A₄, c₄):L₂, (A₈, c₈):L₃]$
$\cdot = c₉, A₉$
$p₁₀: 0 = 0$
$p₁₁: [0]:L_{end} = \pi (0)$
Example

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

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

\[
A_{L_{5,6}} = \cdot \\
p_6: (A_6, A_{L_{6}}) = (A_5, A_{L_{5,6}}) \\
c_{67} = A_{L_6} \\
p_7: (A_7, c_7) = (A_6, c_{67}) \\
p_8: [(A_8, c_8):L_4] = \pi(A_7, c_7)
\]

\[
p_9: (A_9, c_9) = \Phi[(A_4, c_4):L_2, (A_8, c_8):L_3] \\
= c_9, A_9 \\
p_{10}: 0 = 0 \\
p_{11}: [0:L_{end}] = \pi(0)
\]