Register Allocation by Puzzle Solving

EECS 322: Compiler Construction

Simone Campanoni
Robby Findler
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

Hardware

MyVar1  MyVar2  MyVar3

r8  r9  r10
Register Allocation

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

Generated-code run time vs. Compilation time

Ideal

... in significantly less time!

Equivalent quality of graph coloring
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?

```plaintext
(: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

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

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
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)

• 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 into two rows (soon will be clear why)
- A register determinates the shape of the corresponding puzzle board.
- Register aliasing determines the columns

- PowerPC
- ARM integer registers
- SPARC v8
- ARM float registers
- SPARC v9
## Puzzle pieces accepted by boards

<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></td>
</tr>
<tr>
<td>Type-1</td>
<td></td>
<td>Y, X, Y, Z</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-2</td>
<td></td>
<td>Y, Y, Y, X, X, X</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></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
}
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

\[
x := 3 \rightarrow x := 3
\]

\[
x := 3 \rightarrow a > b
\]

\[
x := 3 \rightarrow y := a + b
\]

\[
x := 3 \rightarrow z := 2 \times y
\]

\[
x := 3 \rightarrow w := y + z
\]

\[
x := 3 \rightarrow z := w + x
\]

\[
x := 3 \rightarrow y := a - b
\]

\[
x := 3 \rightarrow y := y \times 10
\]

\[
x := 3 \rightarrow w := w + y
\]
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

```
\[
\begin{align*}
x_1 &:= 3 \\
a_1 &> b_1 \\
y_1 &:= a_1 + b_1 \\
z_1 &:= 2 * y_1 \\
w_2 &:= y_1 + z_1 \\
z_1 &:= w_2 + x_1 \\
y_2 &:= a_1 - b_1 \\
z_2 &:= 2 * y_2 \\
w_3 &:= w_1 + y_3
\end{align*}
\]
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

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

Not SSA

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

Still not SSA

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

SSA
Eliminating $\Phi$

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

$$b_1 = c + 1$$
$$b_2 = d + 1$$
$$b_3 = \Phi(b_1, b_2)$$
If ($b_3 > N$)

$$b_1 = c + 1$$
$$b_2 = d + 1$$
$$b_3 = b_1$$
If ($b_3 > N$)

Not SSA
Eliminating Φ in practice

• Copies performed at Φ 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 Φs
• 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

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

Not SSI

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

SSI

Basic block: sequence of instructions with
• only one entry point and
• only one exit point

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

BB2
  :L2
  (c <- 10)
```
SSA and SSI code

If \( b > 1 \)

\[
\begin{align*}
... &= c + 1 \\
... &= c \times 2 
\end{align*}
\]

Not SSA and not SSI

SSA but not SSI

SSA and SSI
Parallel copies

- Rename variables in parallel

\[
\begin{align*}
V &= X + Y \\
Z &= A + B \\
(V_1, X_1, Y_1, Z_1, A_1, B_1) &= (V, X, Y, Z, A, B) \\
V_1 &= X_1 + Y_1 \\
(V_2, X_2, Y_2, Z_2, A_2, B_2) &= (V_1, X_1, Y_1, Z_1, A_1, B_1) \\
Z_2 &= A_2 + B_2
\end{align*}
\]
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

(a) 

L₁
A = •
p₁: branch L₂, L₃

L₂

c = p₃: jump L₄

L₄
join L₂, L₃
p₉: • = c, A
p₁₀: jump L_end

(b) 

L₁
A₀₁ = •
p₁: (A₁) = (A₀₁)

L₂
AL = •
p₆: c = AL
p₇: jump L₄

L₃
AL₅₆ = •
p₆: (A₆, AL₆) = (A₅, AL₅₆)
c₆₇ = AL₆
p₇: (A₇, c₇) = (A₆, c₆₇)
p₈: [(A₈, c₈):L₄] = π(A₇, c₇)

L₄
p₉: (A₉, c₉) = Φ[(A₄, c₄):L₂, (A₈, c₈):L₃]
• = c₉, A₉
p₁₀: () = ()
p₁₁: [()]:L_end = π()
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

\[
\begin{align*}
L_1 & \quad A_{01} = \cdot \\
p_1: (A_1) = (A_{01}) \\
p_{2,5}: [(A_2):L_2, (A_5):L_3] = \pi(A_1) \\
p_0: [():L_1] = \pi()
\end{align*}
\]

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

\[
\begin{align*}
L_3 & \quad A_{L56} = \cdot \\
p_6: (A_6, A_{L56}) = (A_5, A_{L56}) \\
c_{67} = A_{L6} \\
p_7: (A_7,c_7) = (A_6,c_{67}) \\
p_8: [(A_8,c_8):L_4] = \pi(A_7,c_7)
\end{align*}
\]

\[
\begin{align*}
L_4 & \quad p_9: (A_9, c_0) = \Phi[(A_4, c_4):L_2, (A_8, c_8):L_3] \\
& \quad \cdot = c_0, A_9 \\
p_{10}: () = () \\
p_{11}: [():L_{end}] = \pi()
\end{align*}
\]
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

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

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

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

\[ \mathcal{A}_{L_{56}} = \cdot \]
\[ p_6 : (A_6, \mathcal{A}_{L_6}) = (A_5, \mathcal{A}_{L_{56}}) \]
\[ c_{67} = \mathcal{A}_{L_6} \]
\[ p_7 : (A_7, c_7) = (A_6, c_{67}) \]
\[ p_8 : [(A_8, c_8):L_4] = \pi(A_7, c_7) \]

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

\[ \begin{align*}
A_0 & = \bullet \\
A_1 & = A_0 \\
A_2 & = A_3 \\
A_3 & = A_4 \\
A_4 & = A_5 \\
A_5 & = A_6 \\
A_6 & = A_7 \\
A_7 & = A_8 \\
A_8 & = A_9 \\
\end{align*} \]

\[ \begin{align*}
p_1 &: (A_1) = (A_0) \\
p_2 &: (A_2) = (A_3) \\
p_3 &: (A_3) = (A_4) \\
p_4 &: (A_4) = (A_5) \\
p_5 &: (A_5) = (A_6) \\
p_6 &: (A_6) = (A_7) \\
p_7 &: (A_7) = (A_8) \\
p_8 &: (A_8) = (A_9) \\
\end{align*} \]

\[ \begin{align*}
\pi(A_1) & = \pi(A_0) \\
\pi(A_3) & = \pi(A_2) \\
\pi(A_4) & = \pi(A_3) \\
\pi(A_5) & = \pi(A_4) \\
\pi(A_6) & = \pi(A_5) \\
\pi(A_7) & = \pi(A_6) \\
\pi(A_8) & = \pi(A_7) \\
\pi(A_9) & = \pi(A_8) \\
\end{align*} \]
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
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 $a$
  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
  • $r$ succeeds or
  • $s$ succeeds
Solving type 1 puzzles: a visual language

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

<table>
<thead>
<tr>
<th>Rule -&gt;</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>z</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>z</td>
<td>z</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>y</td>
<td>x</td>
<td>x</td>
<td>z</td>
<td>z</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>y</td>
<td>x</td>
<td>x</td>
<td>z</td>
<td>z</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>z</td>
<td>z</td>
<td>z</td>
<td>z</td>
<td>z</td>
<td>z</td>
<td>z</td>
</tr>
</tbody>
</table>

**Puzzle solver execution**
- For each statement $s_1, ..., s_n$
  - 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

- Puzzle

1. s1 matches a1 only
2. Apply s1 to a1 succeeds and returns this puzzle
3. s2 matches a2 only
4. Apply s2 to a2
   A. Apply first rule of s2: fails
   B. Apply second rule of s2: success
Program execution: another example

- A puzzle solver

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

5. Apply s2 to a3

**Puzzle solved!**
Program execution: yet another example

- A puzzle solver

\[
\begin{array}{c}
\text{s1} \\
\begin{array}{c}
\text{x} \\
\text{x} \\
\text{X}
\end{array} \\
\end{array}
\quad \begin{array}{c}
\text{s2} \\
\begin{array}{c}
\text{X} \\
\text{Y}
\end{array}
\end{array}
\]

- Puzzle

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

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
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 a 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</th>
<th>Board</th>
<th>Kinds of Pieces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-0</td>
<td><img src="image" alt="Type-0 Board" /></td>
<td><img src="image" alt="Type-0 Pieces" /></td>
</tr>
<tr>
<td>Type-1</td>
<td><img src="image" alt="Type-1 Board" /></td>
<td><img src="image" alt="Type-1 Pieces" /></td>
</tr>
<tr>
<td>Type-2</td>
<td><img src="image" alt="Type-2 Board" /></td>
<td><img src="image" alt="Type-2 Pieces" /></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 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 \( v \)
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 solved puzzles to assembly code
Thank you!

Compilation time

Generated code run time

A

C

D

E

Equivalent quality of graph coloring

... in significantly less time!

Ideal

Today and last Wed. → B

Compilation time