Advanced compiler construction

Advanced graph coloring

Simone Campanoni
simonec@eecs.northwestern.edu
A coloring algorithm

Algorithm:
1. Repeatedly select a node and remove it from the graph, putting it on top of a stack

2. When the graph is empty, rebuild it
   • Select a color on each node as it comes back into the graph, making sure no adjacent nodes have the same color
   • If there are not enough colors, the algorithm fails
     • Spilling comes in here
     • Select the nodes (variables) you want to spill
Outline

• Coalescing and freezing

• Advanced register order

• Advanced spilling
Limitation of our basic approach

(:myF 0 0
  %v0 <- rdi
  %v1 <- %v0
  %v2 <- %v0
  rax <- %v0
  rax += %v1
  rax += %v2
  return
)

(:myF 0 0
  rax <- rdi
  rax += rdi
  rax += rdi
  return
)
Advanced heuristic: coalescing

(:myF 0 0
%v0 <- rdi
%v1 <- %v0
%v2 <- %v0
rax <- %v0
rax += %v1
rax += %v2
return)

Are they useful? (:myF 0 0
 rdi <- rdi
 rdi <- rdi
 r10 <- rdi
 rax <- rdi
 rax += rdi
 rax += r10
return)
Advanced heuristic: coalescing

(:myF 0 0
  %v0 <- rdi
  %v1 <- %v0
  %v2 <- %v0
  rax <- %v0
  rax += %v1
  rax += %v2
  return
)

(:myF 0 0
  r10 <- rdi
  rax <- rdi
  rax += rdi
  rax += r10
  return
)
Advanced heuristic: coalescing

(:myF 0 0
  %v0 <- rdi
  %v1 <- %v0
  %v2 <- %v0
  rax <- %v0
  rax += %v1
  rax += %v2
  return
)
Coalescing problem

• Coalescing can significantly increase the quality of the code
• Merging N nodes increases the degree of the resulting node
• This might generate a graph that requires more colors
  • More spills!
Coalescing: the potential problem

(:myF 3 0
  %v0 <- rdi
  %v0 += rdi
  %v0 += rsi
  %v0 += r10
  %v1 <- %v0
  %v2 <- %v0
  rax <- %v0
  rax += %v1
  rax += %v2
  return
)

• Graph coloring without coalescing succeeded!
• Let’s try to do coalescing before graph coloring
Coalescing: the potential problem

(:myF 3 0
   %v0 <- rdi
   %v0 += rdi
   %v0 += rsi
   %v0 += r10
   %v1 <- %v0
   %v2 <- %v0
   rax <- %v0
   rax += %v1
   rax += %v2
   return
)

FAIL
Coalescing problem

• Coalescing can significantly increase the quality of the code
• Merging N nodes increases the degree of the resulting node
• This might generate a graph that requires more colors
  • More spills!
• So when should we apply it?
• Two common conservative strategies:
  1. Briggs
  2. George
Briggs

Nodes a and b can be coalesced if the resulting node \( ab \) will have fewer than \( K \) neighbors of degree \( \geq K \)

• \( K = \) Number of general purpose registers

• This coalescing is guaranteed not to turn a \( K \)-colorable graph into a non-\( K \)-colorable graph
Nodes $a$ and $b$ can be coalesced if for every adjacent node $t$ of $a$, either
  • $(t, b)$ already exists or
  • $\text{Degree}(t) < K$
Graph coloring without coalescing

Code analysis

Interference graph, f

Simplify graph

Select

Spill
Graph coloring with coalescing

1. Code analysis
2. Interference graph, $f$
3. Tag nodes to be move-related
4. Simplify graph only for not-move-related nodes with degree < GP registers
5. Coalesce with Briggs and George
6. Select
7. Spill
Advanced heuristic: freeze move nodes

1. Tag nodes to be move-related
2. Simplify graph only for not-move-related nodes
3. Coalesce with Briggs and George
4. Freeze (give up coalescing some nodes)
5. Select
Outline

• Coalescing and freezing

• Advanced register order

• Advanced spilling
Example

(:myF
  0 8
  %myV1 <- 1
  %myV2 <- 1
  %myV3 <- 1
  %myV4 <- 1
  %myV5 <- 1
  %myV6 <- 1
  %myV7 <- 1
  %myV8 <- 1
  mem rsp 0 <- %myV1
  mem rsp 8 <- %myV2
  mem rsp 16 <- %myV3
  mem rsp 24 <- %myV4
  mem rsp 32 <- %myV5
  mem rsp 40 <- %myV6
  mem rsp 48 <- %myV7
  mem rsp 56 <- %myV8
  return
)
Registers

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Result</th>
<th>Caller save</th>
<th>Callee save</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdi</td>
<td>rax</td>
<td>r10</td>
<td>r12</td>
</tr>
<tr>
<td>rsi</td>
<td></td>
<td>r11</td>
<td>r13</td>
</tr>
<tr>
<td>rdx</td>
<td></td>
<td>r8</td>
<td>r14</td>
</tr>
<tr>
<td>rcx</td>
<td></td>
<td>r9</td>
<td>r15</td>
</tr>
<tr>
<td>r8</td>
<td></td>
<td>rax</td>
<td>rbp</td>
</tr>
<tr>
<td>r9</td>
<td></td>
<td>rcx</td>
<td>rbx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rdi</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rdx</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rdx</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rsi</td>
<td></td>
</tr>
</tbody>
</table>
Example

(:myF
  0 8
  %myV1 <- 1
  %myV2 <- 1
  %myV3 <- 1
  %myV4 <- 1
  %myV5 <- 1
  %myV6 <- 1
  %myV7 <- 1
  %myV8 <- 1

  Caller save
    r10
    r11
    r8
    r9
    rcx
    rdi
    rdx
    rsi
    rax

  mem rsp 0  <-  %myV1
  mem rsp 8  <-  %myV2
  mem rsp 16 <-  %myV3
  mem rsp 24 <-  %myV4
  mem rsp 32 <-  %myV5
  mem rsp 40 <-  %myV6
  mem rsp 48 <-  %myV7
  mem rsp 56 <-  %myV8

  return
)

We can color this graph without spilling
Example

(:myF
  0 9
  %myV1 <- 1
  %myV2 <- 1
  %myV3 <- 1
  %myV4 <- 1
  %myV5 <- 1
  %myV6 <- 1
  %myV7 <- 1
  %myV8 <- 1
  %myV9 <- 1

Caller save

  r10
  r11
  r8
  r9
  rcx
  rdi
  rdx
  rsi
  rax

mem rsp 64 <- %myV9
mem rsp 0  <- %myV1
mem rsp 8  <- %myV2
mem rsp 16 <- %myV3
mem rsp 24 <- %myV4
mem rsp 32 <- %myV5
mem rsp 40 <- %myV6
mem rsp 48 <- %myV7
mem rsp 56 <- %myV8
return
)

Will we color this graph without spilling?
Example

(:myF
  0 8
  %myV1 <- 1
  %myV2 <- 1
  %myV3 <- 1
  %myV4 <- 1
  %myV5 <- 1
  %myV6 <- 1
  %myV7 <- 1
  %myV8 <- 1

... // computation that uses myV* variables

Will we color this graph without spilling?

Which variables will spill?

Can we do better?

What about using callee save registers?

mem rsp -8 <- :ret
call :myF2 0
:ret
mem rsp 0 <- %myV1
mem rsp 8 <- %myV2
mem rsp 16 <- %myV3
mem rsp 24 <- %myV4
mem rsp 32 <- %myV5
mem rsp 40 <- %myV6
mem rsp 48 <- %myV7
mem rsp 56 <- %myV8
return
)
Advanced heuristics: register order

• Change the order of registers depending on the code in f
  • E.g., a lot of calls => prefer callee save registers
  • E.g., a few calls => prefer caller save registers

• This heuristic requires extra code analysis to count #calls
Advanced heuristic: node selection

• Idea: variables used the most at run-time should be in registers

• Approach: give priority to nodes (variables) used in loops

• This heuristic requires a code analysis usually found in middle-ends: loop identification
Outline

• Coalescing and freezing

• Advanced register order

• Advanced spilling
Advanced heuristic: spilling

• Spill a subset of variables at every iteration
  • E.g., 1 at a time

• After having spilled variables
  • Run the register allocation algorithm for spilled variables
  • This will save space in the stack (lower memory pressure)
  • 1 color = 1 stack location