Interference graph

Simone Campanoni
simone.campanoni@northwestern.edu
A graph-coloring register allocator structure

Liveness analysis

IN, OUT

Interferences analysis

Interference graph

Register allocator

Code analysis

Graph coloring

Spill

f without variables and with registers

spill(f, var, prefix)

f with var spilled
Outline

• What is the interference graph

• Algorithm to build the interference graph

• Calling convention
The interference graph

- The Graph coloring algorithm assigns variables to registers

\[
%\text{myVar1} \leftarrow 5 \quad \rightarrow \quad r10 \leftarrow 5
\]

- This transformation must preserve:
  - The original code semantics
  - The constraints of the target architecture (e.g., the second operand of the shift operation must be a constant or rcx)
  - These constraints are encoded in the interference graph

- Nodes: variables
- Edges: interferences

- Meaning of an edge: 2 connected nodes must use different registers
• Next we are going to learn the algorithm that automatically compute the interference graph

• The algorithm adds edges to the interference graph for different categories of constraints, one category at a time

• We will motivate each category of constraints by showing when the algorithm is incorrect if such a category is not considered
Generating the interference graph

- 1 node per variable
- GP registers are considered variables
- Connect each pair of variables that belong to the same IN or OUT set
- Connect a GP register to all other registers (even those not used by f)
- And ...

Is this correct?

Graph coloring

```
(@myF 0 0
%myVar1 <- 2
%myVar2 <- 40
%myVar3 <- %myVar1
%myVar4 <- 42
%myVar3 += %myVar2
print %myVar3
)
```

```
(@myF 0 0
r10 <- 2
r11 <- 40
r10 <- r10
r11 <- 42
r10 += r11
print r10
)```

```plaintext
%myVar1
%myVar2
%myVar3
%myVar4
```

```plaintext
r10
r11
%myVar1
%myVar2
%myVar3
%myVar4
```
Generating the interference graph (2)

- 1 node per variable
- GP registers are considered variables
- Connect each pair of variables that belong to the same IN or OUT set
- Connect a GP register to all other registers (even those not used by the function)
- Connect variables in KILL[i] with those in OUT[i]
  - Necessary for dead code that defines a variable

```plaintext
(@myF 0
 %myVar1 <- 2
 %myVar2 <- 40
 %myVar3 <- %myVar1
 %myVar4 <- 42
 %myVar3 += %myVar2
 print %myVar3
)
```

```plaintext
(@myF 0 1
 r10 <- 2
 r11 <- 40
 r10 <- r10
 mem rsp 0 <- 42
 r10 += r11
 print r10
)
```
Generating the interference graph (3)

• 1 node per variable
• GP registers are considered variables
• Connect each pair of variables that belong to the same IN or OUT set
• Connect a GP register to all other registers (even those not used by f)
• Connect variables in KILL[i] with those in OUT[i]
  • Necessary for dead code that defines a variable

```
(@myF 0
 %myVar1 <- 1 -{ }
 %myVar2 <- 2 -{ %myVar1, %myVar2 }
 %myVar2 <<= %myVar1 -{ }
 )
```

Is this correct?

```
(@myF 0 0
 r10 <- 1
 r11 <- 2
 r11 <<= r10
 )
```
Constrains in the target language L1

• The L1 instruction \( x \text{ sop } sx \) is limited to only shifting by the value of \( rcx \) (or by a constant)
• This must be encoded in the interference graph
• Add interference edges to disallow the illegal registers when building the interference graph
• For example, consider the following example:

\[
a <<= b
\]

we need to add edges between \( b \) and every register except \( rcx \)
This ensures \( b \) will end up in \( rcx \) (or spilled)
Generating the interference graph (3)

- 1 node per variable
- GP registers are considered variables
- Connect each pair of variables that belong to the same IN or OUT set
- Connect a GP register to all other registers (even those not used by f)
- Connect variables in KILL[i] with those in OUT[i]
  - Necessary for dead code that defines a variable
- Handle constrained arithmetic via extra edges
Outline

• What is the interference graph

• Algorithm to build the interference graph

• Calling convention
The relation between Interference graph, calling convention, and liveness analysis

• Finally, we can understand why we had the following rules baked within the Liveness analysis

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>call u N</td>
<td>{ u, args used}</td>
<td>{ caller save registers}</td>
</tr>
<tr>
<td>call RUNTIME N</td>
<td>{ args used}</td>
<td>{ caller save registers}</td>
</tr>
<tr>
<td>return</td>
<td>{ rax, callee save registers}</td>
<td>{ }</td>
</tr>
</tbody>
</table>
Let’s assume we don’t treat call and return instructions with special rules.

In other words, let’s assume we don’t embed the calling convention within the Liveness analysis.
If we don’t encode calling convention within liveness analysis

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>call u N</td>
<td>{ u}</td>
</tr>
<tr>
<td>call RUNTIME N</td>
<td>{ }</td>
</tr>
<tr>
<td>return</td>
<td>{ }</td>
</tr>
</tbody>
</table>
Code example

```plaintext
(@myF
  0
  %a <- 2    // 1
  rax <- %a  // 2
  return    // 3
)

• Are GEN and KILL sets correct?
```
Algorithm

for (each instruction $i$) {
    GEN[$i$] = ...
    KILL[$i$] = ...
}

for (each instruction $i$)  IN[$i$] = OUT[$i$] = { };
do{
    for (each instruction $i$){
        IN[$i$] = GEN[$i$] $\cup$ (OUT[$i$] $-$ KILL[$i$])
        OUT[$i$] = $\cup$ a successor of $i$ IN[$s$]
    }
} while (changes to any IN or OUT occur);
Code example

(@myF
  0
  %a <- 2    // 1
  rax <- %a  // 2
  return    // 3
)

• Are GEN and KILL sets correct?
for (each instruction $i$) {
    GEN[$i$] = ...
    KILL[$i$] = ...
}

for (each instruction $i$) IN[$i$] = OUT[$i$] = { }

do{
    for (each instruction $i$)
    {
        IN[$i$] = GEN[$i$] $\cup$ (OUT[$i$] $-$ KILL[$i$])
        OUT[$i$] = $\bigcup_{s \text{ a successor of } i} \text{IN}[s]$
    }
} while (changes to any IN or OUT occur);
Code example

```plaintext
(@myF
0
%a <- 2    // 1
rax <- %a  // 2
return     // 3
)
```

- Are GEN and KILL sets correct?

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN[i]</td>
<td>GEN[i] U (OUT[i] – KILL[i])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT[i]</td>
<td>U_{s a successor of i} IN[s]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

19
Code example

```plaintext
(@myF
  0
  %a <- 2 // 1
  rax <- %a // 2
  return // 3
)

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{%a}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>{%a}</td>
<td>{rax}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>{rax}</td>
<td>{}</td>
<td>{rax}</td>
<td>{}</td>
</tr>
</tbody>
</table>
```

- Are GEN and KILL sets correct?

\[
\begin{align*}
\text{IN}[i] &= \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \\
\text{OUT}[i] &= \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\end{align*}
\]
### Code example

```plaintext
(@myF
  0
  %a <- 2     // 1
  rax <- %a   // 2
  return      // 3
)
```

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{%a}</td>
<td>{ }</td>
<td>{}</td>
</tr>
<tr>
<td>{%a}</td>
<td>{rax}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>{rax}</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

- Are GEN and KILL sets correct?

\[
\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i])
\]
\[
\text{OUT}[i] = \bigcup_{s\text{ a successor of } i} \text{IN}[s]
\]
**Code example**

```plaintext
(@myF
  0
  %a <- 2   // 1
  rax <- %a // 2
  return    // 3
)
```

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>{ %a }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>{ %a }</td>
<td>{ rax }</td>
<td>{ %a }</td>
<td>{ rax }</td>
</tr>
<tr>
<td>{ rax }</td>
<td>{ }</td>
<td>{ rax }</td>
<td>{ }</td>
</tr>
</tbody>
</table>

- Are GEN and KILL sets correct?

\[
\begin{align*}
\text{IN}[i] &= \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \\
\text{OUT}[i] &= \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\end{align*}
\]
### Code example

```c
(@myF
0
%a <- 2  // 1
rax <- %a  // 2
return  // 3
)
```

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{%a}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>{%a}</td>
<td>{rax}</td>
<td>{%a}</td>
<td>{rax}</td>
</tr>
<tr>
<td>{rax}</td>
<td>{}</td>
<td>{rax}</td>
<td>{}</td>
</tr>
</tbody>
</table>

- Are GEN and KILL sets correct?  

\[
\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \\
\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\]
• Are GEN and KILL sets correct?

```
(@myF
  0
  %a <- 2  // 1
  rax <- %a  // 2
  return  // 3
)
```

\[ \text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \]

\[ \text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s] \]
Algorithm

for (each instruction $i$) {
    GEN[$i$] = ...
    KILL[$i$] = ...
}

for (each instruction $i$)  IN[$i$] = OUT[$i$] = { };  
do{
    for (each instruction $i$){
        IN[$i$] = GEN[$i$] U (OUT[$i$] − KILL[$i$])
        OUT[$i$] = $U_s$ a successor of $i$ IN[$s$]
    }
} while (changes to any IN or OUT occur);
Code example

(@myF
0
%a <- 2       // 1   
ra <- %a       // 2   
return        // 3   
)

• Are GEN and KILL sets correct?

\[
\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \\
\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\]
Algorithm

for (each instruction $i$) {
    GEN[$i$] = ...
    KILL[$i$] = ...
}
for (each instruction $i$)  IN[$i$] = OUT[$i$] = { };
do{
    for (each instruction $i$){
        IN[$i$] = GEN[$i$] $\cup$ (OUT[$i$] $-$ KILL[$i$])
        OUT[$i$] = $\bigcup_{s \text{ a successor of } i} \text{IN}[s]$
    }
} while (changes to any IN or OUT occur);
### Code example

```plaintext
(@myF
  0
  %a <- 2  // 1
  rax <- %a  // 2
  return  // 3
)
```

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{%a}</td>
<td>{}</td>
<td>{%a}</td>
</tr>
<tr>
<td>{%a}</td>
<td>{rax}</td>
<td>{%a}</td>
<td>{rax}</td>
</tr>
<tr>
<td>{rax}</td>
<td>{}</td>
<td>{rax}</td>
<td>{}</td>
</tr>
</tbody>
</table>
```
Steps

1. Compute IN and OUT sets

2. Compute interference graph from IN and OUT sets
• Graph coloring can assign r12 to %a
Code example

(@myF
  0
  r12 <- 2  // 1
  rax <- r12 // 2
  return     // 3
)

• Are GEN and KILL sets correct?
• Graph coloring can assign r12 to %a
• Is there any problem?
# 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>
Code example

(@myF
  0
  r12 <- 2    // 1
  rax <- r12   // 2
  return       // 3
)

• Are GEN and KILL sets correct?
• Graph coloring can assign r12 to %a
• Is there any problem?

• The calling convention counts as definitions and uses
• When adding them as such, we automatically enforce the calling convention

\[
\text{rax} \quad \text{r12} \\
\text{r10} \quad \%a
\]
Calling convention in GEN/KILL

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>call u N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call RUNTIME N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>{ rax, callee save registers}</td>
<td>{ }</td>
</tr>
</tbody>
</table>
Return instruction in a 2 registers CPU

(@myF 0
  %a <- 2
  return
)

Callee-save: r12

Caller-save: r10

w/o calling convention

%a
r10

Graph coloring

r12

w/o calling convention

%a
r10

Graph coloring

r12

w/ calling convention

%a
r10

Graph coloring

r12

w/ calling convention

%a
r10

Graph coloring

r10

(@myF 0
  r10 <- 2
  return
)

(@myF 0
  r12 <- 2
  return
)
### Calling convention in GEN/KILL

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>call u N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call RUNTIME N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>{ rax, callee save registers}</td>
<td>{ }</td>
</tr>
</tbody>
</table>
Call instructions

• Which register should we use for %a?
  r10

• Is it correct? (r10 is a caller save register)

(@myF 0
  %a <- 2
call @f 0
  %a *= %a
  rax <- %a
  return)

(@myF 0
  r10 <- 2
call @f 0
  r10 *= r10
  rax <- r10
  return)
Calling convention in GEN/KILL

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>call u N</td>
<td>{ u, args used}</td>
</tr>
<tr>
<td></td>
<td>{ caller save registers}</td>
</tr>
<tr>
<td>call RUNTIME N</td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>{ rax, callee save registers}</td>
</tr>
<tr>
<td></td>
<td>{ }</td>
</tr>
</tbody>
</table>
Homework #2

• Compute the interference graph of an L2 function given as input

• Implement the spiller (see Spilling.pdf) for an L2 function given as input

• Implement the full L2 compiler (including the graph coloring algorithm, see Graph_coloring.pdf)
Homework #2

• Compute the interference graph of an L2 function given as input

```c
(*myF
  0
  %myVar1 <= 5
  %myVar2 <= 0
  %myVar2 %= %myVar1
  return
)
```

test/interference/test1.L2f

Your work needs to print to std::cout

A node in the interference graph

%myVar1 %myVar2 r12 r13 r14 r15 rax rbp rbx

Nodes connected with the first one (the order between them doesn’t matter)

rsi r10 r11 r12 ... rbp rbx rcx rdi rdx

test/interference/test1.L2f.out
Testing the interference graph of your homework #2

• Under L2/tests/interference there are the tests you have to pass
• To test:
  • To check all tests: make test_interference
  • To check one test: ./interference test/interference/test1.L2f
• Check out each input/output for each test if you have doubts
  • test/interference/test1.L2f
  • test/interference/test1.L2f.out
A graph-coloring register allocator structure

- Liveness analysis
- Interferences analysis
- Graph coloring
- Code analysis
- Spill

f with var spilled
spill(f, var, prefix)
f without variables and with registers
Always have faith in your ability

Success will come your way eventually

Best of luck!