Interference graph

Simone Campanoni
simonec@eecs.northwestern.edu
A graph-coloring register allocator structure

Liveness analysis

Interferences analysis

IN, OUT

Interference graph

Register allocator

Code analysis

Graph coloring

Spill

\( f \) with \( \text{var spilled} \)

\( \text{spill}(f, \text{var, prefix}) \)

\( f \) without variables and with registers
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
  %myVar1 <- 5  ➔  r10 <- 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 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 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

\[
\begin{align*}
\text{(:myF 0 0} & \\
%\text{myVar1} & \leftarrow 2 & \{ \} \\
%\text{myVar2} & \leftarrow 40 & \{ %\text{myVar1} \} \\
%\text{myVar3} & \leftarrow %\text{myVar1} & \{ %\text{myVar1}, %\text{myVar2} \} \\
%\text{myVar4} & \leftarrow 42 & \{ %\text{myVar1}, %\text{myVar2} \} \\
%\text{myVar3} & += %\text{myVar2} & \{ %\text{myVar1}, %\text{myVar2} \} \\
\text{print %myVar3} & \leftarrow 1 \\
\end{align*}
\]
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 f)
• 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
)

!(: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 \texttt{x sop sx} is limited to only shifting by the value of \texttt{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:
  \[
  a \leftarrow\ll b
  \]
  we need to add edges between \texttt{b} and every register except \texttt{rcx}
  This ensures \texttt{b} will end up in \texttt{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

```plaintext
(:myF 0 %myVar1 <- 1 %myVar2 <- 2 %myVar2 <= %myVar1 )
```

```plaintext
(:myF 0
  rcx <- 1
  r11 <- 2
  r11 <<= rcx
)
```
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, \text{ 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.
Code example

(: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[$i$] 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
  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?
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

```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?

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

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

- Are GEN and KILL sets correct?

\[
\begin{array}{|c|c|c|c|c|}
\hline
& GEN & KILL & IN & OUT \\
\hline
\text{IN} & \{\} & \{\} & \{\} & \{\} \\
\hline
\text{OUT} & \{\} & \{\} & \{\} & \{\} \\
\hline
\text{IN}[i] & = & \text{GEN}[i] & \cup & \text{OUT}[i] - \text{KILL}[i] \\
\text{OUT}[i] & = & \cup_{s \text{ a successor of } i} \text{IN}[s] \\
\hline
\end{array}
\]
### Code example

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

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{}</td>
<td>{%a}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>%a &lt;- 2</td>
<td>{%a}</td>
<td>{rax}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>rax &lt;- %a</td>
<td>{rax}</td>
<td>{}</td>
<td>{rax}</td>
<td>{}</td>
</tr>
<tr>
<td>return</td>
<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

```
(: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?

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

\[(::myF\]
\[0\]
\[%a \leftarrow 2 \quad \quad // 1\]
\[\text{rax} \leftarrow %a \quad \quad // 2\]
\[\text{return} \quad \quad \quad \quad // 3\]
\[)\]

- 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>1</td>
<td>{}</td>
<td>%{a}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>2</td>
<td>%{a}</td>
<td>{rax}</td>
<td>%{a}</td>
<td>{rax}</td>
</tr>
<tr>
<td>3</td>
<td>{rax}</td>
<td>{}</td>
<td>{rax}</td>
<td>{}</td>
</tr>
</tbody>
</table>

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

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

<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>

- Are GEN and KILL sets correct?

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

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{GEN} & \text{KILL} & \text{IN} & \text{OUT} \\
\hline
\{\} & \{%a\} & \{\} & \{\} \\
\{%a\} & \{rax\} & \{%a\} & \{rax\} \\
\{rax\} & \{\} & \{rax\} & \{\} \\
\hline
\end{array}
\]

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

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

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

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

- Graph coloring can assign r12 to %a

![Graph coloring diagram]

- 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

Arguments
- rdi
- rsi
- rdx
- rcx
- r8
- r9

Result
- rax

Caller save
- r10
- r11
- r8
- r9
- rax
- rcx
- rdi
- rdx
- rsi

Callee save
- r12
- r13
- r14
- r15
- rbp
- rbx
Code example

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

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

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

\[ \text{rax} \rightarrow \text{r12} \]

\[ \text{r10} \quad \text{%a} \]
## 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
r12

w/ calling convention

%a
r12

Graph coloring

(:myF 0
  r12 <- 2
  return)

Graph coloring

(:myF 0
  r10 <- 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>Function</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></td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>{ rax, callee save registers}</td>
<td>{ }</td>
</tr>
</tbody>
</table>
Homework #2

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

```cpp
(:myF
  0
  %myVar1 <- 5
  %myVar2 <- 0
  %myVar2 += %myVar1
  return
)
```

%myVar1 %myVar2 r12 r13 r14 r15 rax rbp rbx

The order between rows doesn’t matter

Your work needs to print to std::cout

A node in the interference graph

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

rsi r10 r11 r12 ... rbp rbx rcx rdi rdx
	est/interference/test1.L2f

test/interference/test1.L2f.out

• Implement the spiller (see Spilling.pdf)
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
- Interference graph

Register allocator
- Code analysis
- Graph coloring

Spill
- spill(f, var, prefix)
- f with var spilled

f without variables and with registers

f with

IN, OUT

f