Liveness analysis

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

Register allocator

Code analysis

Graph coloring

Spill

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

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

\( f \text{ without variables and with registers} \)
Outline

• Code analyses required

• Liveness analysis

• Calling convention
Task: From Variables to Registers

```
(:MyVeryImportantFunction

%MyVar1 <- 2
%MyVar2 <- 40
%MyVar3 <- %MyVar1
%MyVar3 += %MyVar2
print %MyVar3
)
```

- Why can we map MyVar1 and MyVar3 to r8?
- Why can’t we map MyVar1 and MyVar2 to r8?

We built the interference graph.

To compute it automatically, we need the liveness analysis.
Liveness analysis

**Goal:**
Identify the variables whose values might be used in the future just before and just after a given instruction $i$, for every $i$ in a function $f$

```markdown
(:myF 0 0
 %myVar1 <- 2
 %myVar2 <- 40
 %myVar3 <- %myVar1
 %myVar3 += %myVar2
 rax <- %myVar3
 return
)
```

IN (just before) and OUT (just after) sets

<table>
<thead>
<tr>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>{myVar1}</td>
<td>{myVar1, myVar2}</td>
</tr>
<tr>
<td>{myVar3, myVar2}</td>
<td>{myVar3}</td>
</tr>
<tr>
<td>{myVar3}</td>
<td>{ }</td>
</tr>
<tr>
<td>{ }</td>
<td>{myVar1}</td>
</tr>
</tbody>
</table>

Live ranges

Interference graph

Graph coloring

:myF w/o variables
Steps

1. Compute IN and OUT sets

2. Compute interference graph from IN and OUT sets
Generating the interference graph

• 1 node per variable or general-purpose (GP) register
• Connect each pair of variables that belong to the same IN or OUT set
• Connect a GP register to all other GP registers
• ... (see Interference graph slides for other steps)

• Meaning of an edge: 2 connected nodes must use different registers
Outline

• Code analyses required

• Liveness analysis

• Calling convention
Variables in the liveness analysis

• GP 64-bit registers are seen as variables for the liveness analysis
  • rsp is not included

• Every time we say “variable” in the context of liveness analysis, we mean either L2 variables or GP 64-bit registers

• IN and OUT sets of the liveness analysis includes variables
  • Hence, they include L2 variables or GP 64-bit registers
  • IN[i] = {r10}
Execution path

Let \( i \) be an instruction, we need to identify the set of variables with values that will be used just before and just after \( i \)
Liveness analysis

A variable is **live** at a particular point in the program if its value at that point will be used in the future (dead, otherwise)

- To compute liveness at a given point, we need to look into the future
- A variable \( v \) is live at a given point of a program \( p \) if
  - Exist a directed path from \( p \) to a use of \( v \) and
  - that path does not contain any definition of \( v \)

\[
\begin{align*}
%v1 & \leftarrow 5 \\
%v3 & \leftarrow 3 \\
%v2 & \leftarrow %v1 \\
return
\end{align*}
\]

\[
\begin{align*}
%v1 & \leftarrow 5 \\
%v1 & \leftarrow 3 \\
%v2 & \leftarrow %v1 \\
return
\end{align*}
\]
Liveness analysis algorithm

1. Identify which variables are define and which ones are read (used) for each instruction
   • GEN and KILL sets

2. Specify how instructions transmit live values around the program
   • How to compute IN and OUT sets from GEN and KILL sets

3. Iterate (2) until nothing (i.e., IN and OUT set) changes
   • Notice that (1) is performed only once!
   • GEN and KILL sets are constants and, therefore, path independent!
GEN and KILL sets

• GEN\[i\] = \{all variables read (used) by instruction \(i\)\}
  \%
  myVar3 \leftarrow \%myVar1 \quad // \text{GEN}\[i\] = \{myVar1\}

• KILL\[i\] = \{all variables defined by instruction \(i\)\}
  \%
  myVar3 \leftarrow \%myVar1 \quad // \text{KILL}\[i\] = \{myVar3\}

\%
myVar3 += \%myVar1
KILL[i] = \{myVar3\} \quad \text{GEN}[i] = \{myVar1, myVar3\}

% is dropped
GEN and KILL sets: more examples

- $\text{GEN}[i] = \{\text{all variables read (used) by instruction } i\}$
- $\text{KILL}[i] = \{\text{all variables defined by instruction } i\}$

```plaintext
rdi++
KILL[i] = \{rdi\}
GEN[i] = \{rdi\}
```
GEN and KILL sets: more examples

• GEN[i] = \{all variables read (used) by instruction i\}
• KILL[i] = \{all variables defined by instruction i\}

cjump rdi <= %v2 :true :false
KILL[i] = \{\}
GEN[i] = \{rdi, v2\}
Liveness analysis algorithm

1. Define which variables are define and which ones are read (used) for each instruction
   • GEN and KILL sets

2. Specify how instructions transmit live values around the program
   • How to compute IN and OUT sets from GEN and KILL sets

3. Iterate (2) until nothing changes
IN and OUT sets

• \( \text{IN}[i] = \{ \text{all variables live right before instruction } i \} \)
  \[ \text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \]

• \( \text{OUT}[i] = \{ \text{all variables live right after instruction } i \} \)
  \[ \text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s] \]

\[ i: \%v2 <- \%v1 \]
If \( \text{OUT}[i] = {} \) then
\[ \text{IN}[i] = \{ \%v1 \} \]

\[ i: \%v2 <- \%v1 \]
If \( \text{OUT}[i] = \{ \%v2 \} \) then
\[ \text{IN}[i] = \{ \%v1 \} \]

\[ i: \text{cjump } \%v = 1 :s1 :s2 \]
\[ i+1: :s1 \]
\[ i+j: :s2 \]
If \( \text{IN}[i+1] = \{ \%v1 \} \)
If \( \text{IN}[i+j] = \{ \%v2 \} \)
Then \( \text{OUT}[i] = \{ \%v1, \%v2 \} \)
Algorithm

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

- Code analyses required

- Liveness analysis

- Calling convention
Code example

(:myF
  0 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

```
(:myF
  0 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>{:myF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%a &lt;- 2</td>
<td>{}</td>
<td>{a}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>rax &lt;- %a</td>
<td>{a}</td>
<td>{rax}</td>
<td>{}</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?
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 0
%a <- 2  // 1
rax <- %a  // 2
return  // 3
)

• Are GEN and KILL sets correct?

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

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

\[
\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\]
## Code example

```
(:myF
0 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>IN</td>
<td>{ }</td>
<td>{a}</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>OUT</td>
<td>{a}</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>IN</td>
<td>{ }</td>
<td>{ }</td>
<td>{rax}</td>
<td>{ }</td>
</tr>
<tr>
<td>OUT</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>IN</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>OUT</td>
<td>{ }</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 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>{a}</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>
<tr>
<td>{}</td>
<td>{}</td>
<td>{ra}x</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 0
  %a <- 2 // 1
  rax <- %a // 2
  return // 3
)

• Are GEN and KILL sets correct?

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

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

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

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

• Are GEN and KILL sets correct?

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

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

- Are GEN and KILL sets correct?

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

\[
\text{OUT}[i] = \bigcup_s \text{a successor of } i \text{ IN}[s]
\]
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 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>{a}</td>
<td></td>
<td>{ }</td>
<td>{a}</td>
</tr>
<tr>
<td>{a}</td>
<td>{rax}</td>
<td>{a}</td>
<td>{rax}</td>
</tr>
</tbody>
</table>
Steps

1. Compute IN and OUT sets

2. Compute interference graph from IN and OUT sets
Generating the interference graph

• 1 node per variable or register

• Connect each pair of variables that belong to the same IN or OUT set
• Connect a register to all other registers
• ... (see Interference graph slides for other steps)

• Meaning of an edge: 2 connected nodes must use different registers
Code example

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

- Graph coloring can assign r12 to a
Code example

(:myF
  0 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 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?
# 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 0
 %a <- 2
 return
 )

Callee-save: r12

Caller-save: r10

Graph coloring

(:myF 0 0
 (r12 <- 2)
 (return)
 )

Graph coloring

(a - r12
 r10)

(a - r12
 r10)

(a - r12
 r10)

(a - r12
 r10)
## 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 0
 %a <- 2
 call :f 0
 %a *= %a
 rax <- %a
 return)

(:myF 0 0
 r10 <- 2
 call :f 0
 r10 *= r10
 rax <- r10
 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>{ 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>
# 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>{ u,&lt;br&gt;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,&lt;br&gt;callee save registers}</td>
<td>{ }</td>
</tr>
</tbody>
</table>
Homework #1

• Compute the IN and OUT sets of all instructions of an L2 function given as input

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

Your work

(in
  (r13 r15 rax rbp r12 rbx )
  (r13 r15 rax rbp r12 myVar1 rbx )
  (r13 r15 rax rbp myVar2 r12 myVar1 rbx )
  (r13 r15 rax rbp r12 rbx )
) (out
  (r13 r15 rax rbp r12 myVar1 rbx )
  (r13 r15 rax rbp myVar2 r12 myVar1 rbx )
  (r13 r15 rax rbp r12 rbx )
  ( )
))
Testing your homework #1

• Under L2/tests/liveness there are the tests you have to pass

• Two new compiler arguments: -l –g
  • Check L2/src/compiler.cpp

• To test: make test_liveness

• Check out each input/output for each test if you have doubts
  • test/liveness/test1.L2f
  • test/liveness/test1.L2f.out