Compiler Design

Lecture 16: Liveness Analysis

Christophe Dubach
Winter 2022

Some material from Prof. Michelle Strout, CS553, Colorado State University.

Timestamp: 2022/03/08 14:36:00
Example of generated MIPS code (using virtual registers):

```mips
.data
x: .space 4
y: .space 4
.text
la v0, x
lw v1, (v0)
add v2, v0, v1
la v3, y
lw v4, (v3)
sub v5, v4, v2
add v6, v2, v4
sw v5, (v0)
sw v6, (v3)
```

After “proper” register allocation (possible output):

```mips
.data
x: .space 4
y: .space 4
.text
la $t0, x
lw $t1, ($t0)
add $t2, $t0, $t1
la $t3, y
lw $t4, ($t3)
sub $t5, $t4, $t2
add $t6, $t2, $t4
sw $t5, ($t0)
sw $t6, ($t3)
```

What if less than 7 architectural registers available for allocation?

- Need to know which values is going to be used in the future.
### Definition

A variable (virtual register) is **live** at some point in the program if it has previously been **defined** by an instruction and will be **used** by an instruction in the future. It is **dead** otherwise.

💡 Two variables can use the same architectural register if they are never used at the same time, *i.e.* never simultaneously live.

⇒ Register allocation use liveness information.
Example:

```plaintext
.data
x: .space 4
y: .space 4
.text
    la   v0, x
    lw   v1, (v0)
    add  v2, v1, v1
    la   v3, y
    lw   v4, (v3)
    sub  v5, v4, v2
    add  v6, v2, v4
    sw   v5, (v0)
    sw   v6, (v3)
```

<table>
<thead>
<tr>
<th>Live after instruction:</th>
<th>v0</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0 v2 v3 v4 v5 v6 v3 v6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Question: how many architectural registers are needed?
Computing liveness is more complicated in the presence of control flow (e.g. loops, if-then-else).

**Assembly pseudo-code:**

```
    a = 0
L1:    b = a + 1
      c = c + b
      a = b*2
      if (a<9) goto L1
    return c
```

Question: what is the live range of b?

To answer this question we need to understand the *dynamic flow* of the program execution.

---

¹We illustrate concepts at a slightly higher level than assembly from this point on.
Control-Flow Graph (CFG)

Concept invented in 1970 by:

Frances Allen (1932–2020), IBM, (1st woman to receive Turing Award in 2006!)

\[
\begin{align*}
a & = 0 \\
L1: & \quad b = a + 1 \\
& \quad c = c + b \\
& \quad a = b \times 2 \\
if \ (a<9) & \quad goto \ L1 \\
& \quad return \ c
\end{align*}
\]

Directed graph:

- entry
  - 1: \(a = 0\)
  - 2: \(b = a + 1\)
  - 3: \(c = c + b\)
  - 4: \(a = b \times 2\)
  - 5: \(a < 9\)
    - True
      - 6: return \(c\)
    - False
      - 6: return \(c\)
What is the live range of $b$?

- $b$ is used in statement 4, so $b$ is live on the $3 \rightarrow 4$ edge.
- Since statement 3 does not define $b$, $b$ is also live on the $2 \rightarrow 3$ edge.
- Statement 2 defines $b$, so any value of $b$ on the $1 \rightarrow 2$ and $5 \rightarrow 2$ edges are not needed, so $b$ is dead along these edges.

$b$ live range is $2 \rightarrow 3 \rightarrow 4$. 
Live range of \(a\):
- \(1 \rightarrow 2\) and \(4 \rightarrow 5 \rightarrow 2\)

Live range of \(b\):
- \(2 \rightarrow 3 \rightarrow 4\)

Live range of \(c\):
- \(entry \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 2\)
  and \(5 \rightarrow 6\)

⚠️ Since \(a\) and \(b\) never simultaneously live, can share a register.
Terminology

Flow Graph

• a Control Flow Graph (CFG) has out-edges that leads to successor nodes and in-edges that come from predecessor nodes

• \( \text{pred}(n) = \text{set of all predecessors of node } n \)

• \( \text{succ}(n) = \text{set of all successors of node } n \)

Examples

• Out-edges of node 5: 5 \( \rightarrow \) 6 and 5 \( \rightarrow \) 2

• \( \text{succ}(5) = \{2,6\} \)

• \( \text{pred}(5) = \{4\} \)

• \( \text{pred}(2) = \{1,5\} \)
Uses and Defs

Def (definition)

- A write of a value to a variable
- \( \text{def}(v) = \text{set of CFG nodes that define variable } v \) 
- \( \text{def}(n) = \text{set of variables defined at node } n \)

Use

- A read of a variable’s value
- \( \text{use}(v) = \text{set of CFG nodes that use variable } v \) 
- \( \text{use}(n) = \text{set of variables used at node } n \)
A variable \( v \) is live on a CFG edge if

- \( \exists \) a directed path from that edge to a use of \( v \) (node \( \in \) use(b)) and
- that path does not go through any def of \( v \) (nodes \( \notin \) def(v)).
Computing Liveness
Flow of Liveness

Data-flow

• Liveness of variables is a property that flows through the edges of the CFG

Direction of flow

• Liveness flows *backward* in the CFG: behaviour of future nodes determines liveness at a given node
Example: flow of liveness for a

Example: flow of liveness for b

entry

1: a = 0

2: b = a + 1

3: c = c + b

4: a = b * 2

5: a < 9

6: return c

True

False

entry

1: a = 0

2: b = a + 1

3: c = c + b

4: a = b * 2

5: a < 9

6: return c

True

False
Liveness at Nodes

We have liveness on edges
  • before and after each node

Two more definitions:
  • A variable is live-out at a node if it is live on any of that node’s out-edges
  • A variable is live-in at a node if it is live on any of that node’s in-edges
Computing Liveness

Rules for computing liveness

1. Generate liveness:
   \[ v \in \text{use}(n) \Rightarrow v \in \text{LIVE}_{in}(n) \]

2. Push liveness across edges:
   \[ v \in \text{LIVE}_{in}(n) \Rightarrow \forall p \in \text{pred}(n) \forall v \in \text{LIVE}_{out}(p) \]

3. Push liveness across nodes:
   \[ v \in \text{LIVE}_{out}(n) \land v \notin \text{def}(n) \Rightarrow v \in \text{LIVE}_{in}(n) \]

Data-flow equations

\[ \text{LIVE}_{in}(n) = \text{use}(n) \cup (\text{LIVE}_{out}(n) - \text{def}(n)) \]

\[ \text{LIVE}_{out}(n) = \bigcup_{s \in \text{succ}(n)} \text{LIVE}_{in}(s) \]
Solving the Data-flow equations

1: for all node \( n \in \text{CFG} \) do
2: \( \text{LIVE}_{\text{in}}(n) = \emptyset \)
3: \( \text{LIVE}_{\text{out}}(n) = \emptyset \)
4: end for
5: repeat
6: for all node \( n \in \text{CFG} \) do
7: \( \text{LIVE}'_{\text{in}}(n) = \text{LIVE}_{\text{in}}(n) \)
8: \( \text{LIVE}'_{\text{out}}(n) = \text{LIVE}_{\text{out}}(n) \)
9: \( \text{LIVE}_{\text{in}}(n) = \text{use}(n) \cup (\text{LIVE}_{\text{out}}(n) - \text{def}(n)) \)
10: \( \text{LIVE}_{\text{out}}(n) = \bigcup_{s \in \text{succ}(n)} \text{LIVE}_{\text{in}}(s) \)
11: end for
12: until \( \text{LIVE}'_{\text{in}}(n) = \text{LIVE}_{\text{in}}(n) \land \text{LIVE}'_{\text{out}}(n) = \text{LIVE}_{\text{out}}(n) \forall n \)

This is a fix-point algorithm for iterative liveness analysis.
### Data-flow equations

\[
\text{LIVE}_{\text{in}}(n) = \text{use}(n) \cup (\text{LIVE}_{\text{out}}(n) - \text{def}(n))
\]
\[
\text{LIVE}_{\text{out}}(n) = \bigcup_{s \in \text{succ}(n)} \text{LIVE}_{\text{in}}(s)
\]
There is something inefficient about this process.

For instance, consider the 3 → 4 edge in the graph:
- \( \text{LIVE}_{out}(4) \) is used to compute \( \text{LIVE}_{in}(4) \)
- \( \text{LIVE}_{in}(4) \) is used to compute \( \text{LIVE}_{out}(3) \)

💡 The algorithm would converge faster if we process the nodes backwards.
Backward Liveness Analysis

1: for all node n ∈ CFG do
2: \( \text{LIVE}_{in}(n) = \emptyset \)
3: \( \text{LIVE}_{out}(n) = \emptyset \)
4: end for
5: repeat
6: for all node n ∈ CFG in “reverse DFS visited order” do
7: \( \text{LIVE}'_{in}(n) = \text{LIVE}_{in}(n) \)
8: \( \text{LIVE}'_{out}(n) = \text{LIVE}_{out}(n) \)
9: \( \text{LIVE}_{out}(n) = \bigcup_{s \in \text{succ}(n)} \text{LIVE}_{in}(s) \)
10: \( \text{LIVE}_{in}(n) = \text{use}(n) \cup (\text{LIVE}_{out}(n) - \text{def}(n)) \)
11: end for
12: until \( \text{LIVE}'_{in}(n) = \text{LIVE}_{in}(n) \land \text{LIVE}'_{out}(n) = \text{LIVE}_{out}(n) \forall n \)
Example with Backward Liveness Analysis

Converges in only 3 iterations!

Data-flow equations

\[
\begin{align*}
\text{LIVE}_{out}(n) &= \bigcup_{s \in \text{succ}(n)} \text{LIVE}_{in}(s) \\
\text{LIVE}_{in}(n) &= \text{use}(n) \cup (\text{LIVE}_{out}(n) - \text{def}(n))
\end{align*}
\]
More performance considerations

**Basic Block**

A straight sequence of assembly instruction which (usually) finishes with a branch/jump instruction.

Key property: Either *all* the instructions in the sequence execute or none execute.

Can significantly decrease the size that a CFG occupies in memory by grouping nodes that have a single predecessor and a single successor into basic blocks.

The instructions in a basic block can be simply represented as a list (rather than a graph).
Example

No basic blocks:

entry
1: a = 0
2: b = a + 1
3: c = c + b
4: a = b * 2
5: a < 9
True
6: return c
False

entry

With basic blocks:

entry
1: a = 0
2: b = a + 1
c = c + b
a = b * 2
a < 9
True
3: return c
False
• Proper register allocation