Norwegian University of Science and Technology

Compiler Construction

Lecture 19–5: Very busy expressions and summary of data flow analyses

Week of 2020-03-30

Michael Engel

Overview

- Data-flow analyses
  - Very busy expressions
  - May- and must-analyses
  - Common features and categorization

Busy expressions

- An expression e is **busy** at a program point if and only if
  - an evaluation of e exists along some path $w_i, \ldots, w_j$ starting at program point w_i
  - no operation of any operand of e exists before its evaluation along the path (e.g., the operands are unchanged)

- If an expression is found to be busy at some program point, it is definitely going to be used in ***some*** path following that point

Very busy expressions

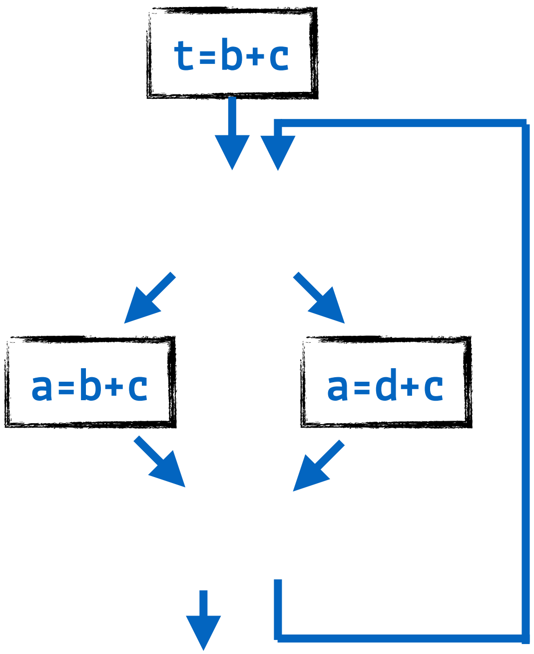
- An expression is **very busy** at some program point if it will definitely be evaluated before its value changes
  - At a program point w_i, an expression is very busy if it is busy along **all** paths starting at w_i

- Dataflow analysis can **approximate** the set of very busy expressions for all program points
- The result of that analysis can then be used to perform ***code hoisting***:
  the computation of a very busy expression can be performed at the earliest point where it is busy
  - this optimization doesn't (necessarily) reduce time, but code space

Busy expressions example

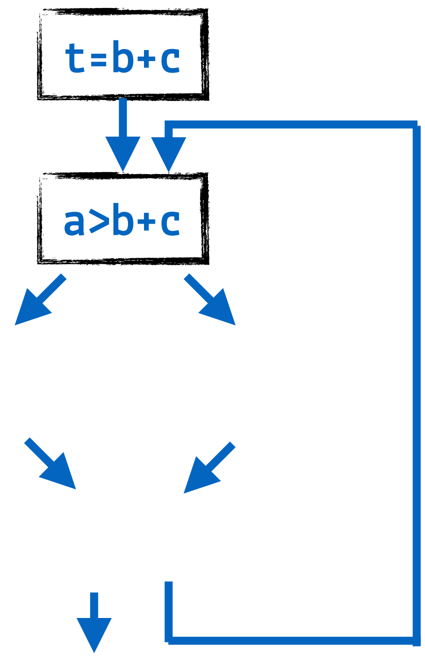
`b+c` is **not** very busy at loop entrance

```
t=b+c;
for (…) {
  if (…) a=b+c;
  else   a=d+c;
}
```



`b+c` ***is*** very busy at loop entrance

```
t=b+c;
for (…) {
  if (a>b+c) x=1;
  else       x=0;
}
```



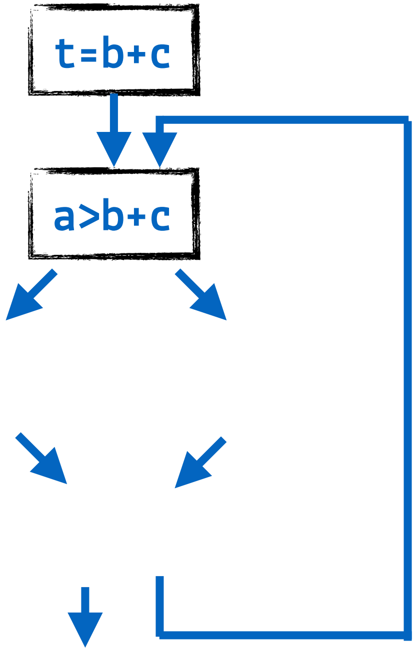
Optimization: code hoisting

- Dataflow analysis can **approximate** the set of very busy expressions for all program points
- If an expression is found to be very busy at w_i, we can move its evaluation to that node
- The result of that analysis can then be used to perform an optimization called ***code hoisting***:
  - the computation of a very busy expression can be performed at the earliest point where it is busy
  - it doesn't (necessarily) reduce time, but code space
- Useful for ***loop invariant code motion***
- If an expression is **invariant** in a loop and is also **very busy**, we know it must be used in the future
- Hence evaluation outside the loop must be worthwhile

Optimization example

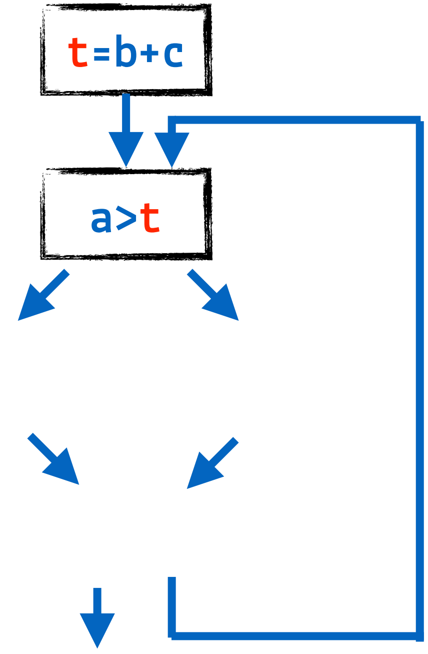
`b+c` ***is*** very busy at loop entrance

```
t=b+c;
for (…) {
  if (a>b+c) x=1;
  else       x=0;
}
```



Evaluate `b+c` ***once*** before loop:

```
t=b+c;
for (…) {
  if (a>t) x=1;
  else     x=0;
}
```



Very busy expressions: flow equations

- We can derive the following data flow equations:

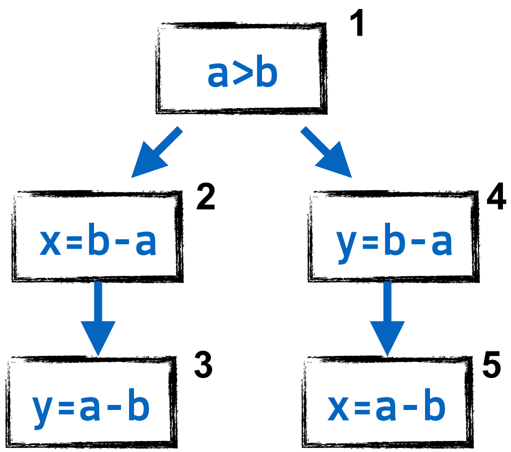
$$\mathtt{Out}_n = \begin{cases} \emptyset & \text{if } n \text{ is final block} \\ \bigcap_{p \in \mathtt{succ}(n)} \mathtt{In}_p & \text{otherwise} \end{cases}$$

$$\mathtt{In}_n = (\mathtt{Out}_n - \mathtt{Kill}_n) \cup \mathtt{Gen}_n$$

```
In1 = Out1
In2 = Out2 ∪ {b-a}
In3 = {a-b}
In4 = Out4 ∪ {b-a}
In5 = {a-b}

Out1 = In2 ∩ In4
Out2 = In3
Out3 = ø
Out4 = In5
Out5 = ø
```

- Example:

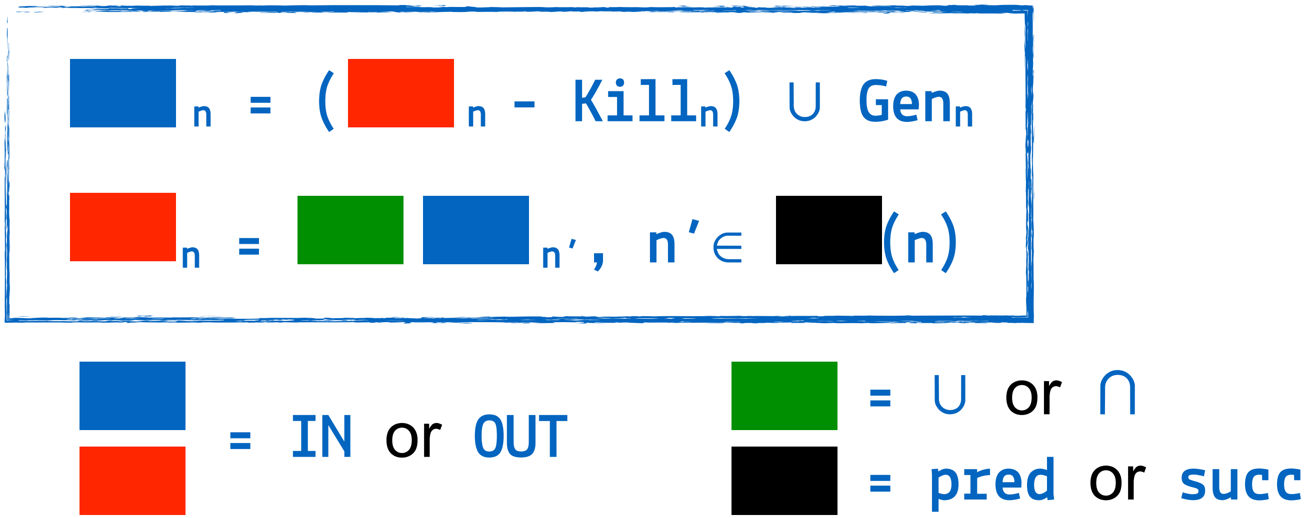


	$\mathtt{Kill}_n$	$\mathtt{Gen}_n$
1	ø	ø
2	ø	{b-a}
3	ø	{a-b}
4	ø	{b-a}
5	ø	{a-b}

	$\mathtt{In}_n$	$\mathtt{Out}_n$
1	{a-b,b-a}	{a-b,b-a}
2	{a-b,b-a}	{b-a}
3	{a-b}	ø
4	{a-b,b-a}	{b-a}
5	{a-b}	ø

A common analysis pattern

- Common pattern for the data-flow analyses we discussed:



- Two choices exist:
  - perform a forward or backward analysis? and
  - whether the analysis computes ∪ or ∩ sets

May and must analyses

- An analysis is said to compute "**may**" facts if those facts hold along **some path** in the control-flow graph
- In contrast, an analysis is said to compute "**must**" facts if those facts hold along **all paths**
- Accordingly, the use of the join operation is ∪ is called "may" analysis and ∩ is a "must"-analysis
- We can now categorize our data-flow analyses according to the data-flow equations used:

	may	**must**
forward	reaching definitions	available expressions
backward	live variables	very busy expressions