Lec-19

29/10/19

Lec - 20

$a = x + y$

$a = 17$

$c = x + y$

SSA form
different version numbers whenever a variablo is modified

$a_0 = x + y$

$a_1 = 17$

$c_0 = x + y$

Value numbering system

$a_0^3 = x^1 + y^2$

$a_1^4 = 17^4$

$c_0^3 = x^1 + y^2$

Value numbering system

$x \leftarrow 1$

$y \leftarrow 2$

$a_0 \leftarrow 3$

$a_1 \leftarrow$ ~~3~~ 4

$c_0 \leftarrow 3$

$1 \rightarrow x$

$2 \rightarrow y$

$3 \rightarrow a_0, c_0$

$4 \rightarrow a_1$

$\langle +, 1, 2 \rangle \longrightarrow 3$

$[17 \longrightarrow 4] \Rightarrow$ This can be put in the code

$a = x + y$

$c = a$

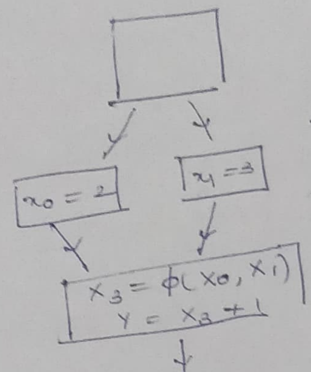
$a = 17$

can work

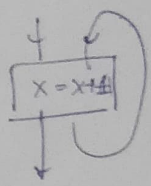
Superlocal value numbering

~~m y x a + b~~

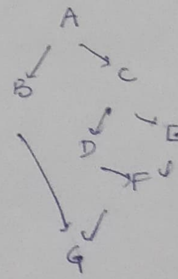
Increase scopo



→ Multiple assignments reach a use, we use a ϕ ~~[illegible]~~ + new assignment



Extended basic block
↳ One leader + followers
each follower has one pred only
and a BB ∈ only 1 EBB



EBB₁ : A, B, C, D, E

EBB₂ : F

EBB₃ : G

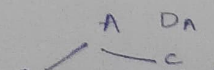
Idea:

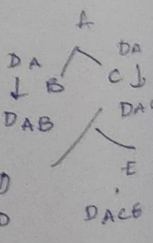
Every EBB is a tree, with a root node.

Value numbering on EBB

· Let D_A have all 3 DS

for B in EBB₁, starts with D_A as initial state



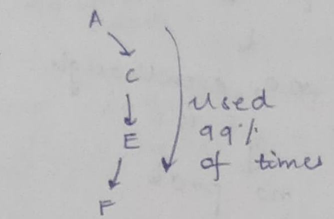


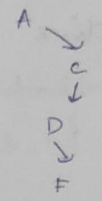
! But the blocks have to be in SSA form. SSA makes analysis and optimization

· A new EBB uses null state

$D_{ACD} \wedge D_{ACE} = D_{AC}$ ⇒ For F to be visited, A and C have to be visited
↑
merging

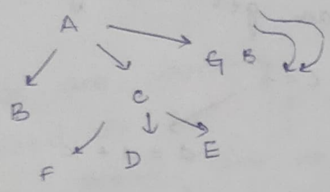
Dynamic compilation →





⇒ Heavily optimized ⇒ If another path is used ⇒ Use unoptimized code

Dominators:



$C \leftarrow D_A$

$B \leftarrow D_A$ as init state

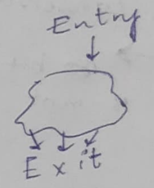
$G \leftarrow D_A$

$D \leftarrow D_{AC}$

$F \leftarrow D_{AC}$

Global optimization: Optimization at a fn level:-

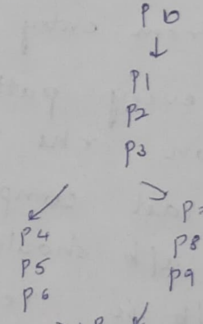
Program points ≠ Labels, given by coder, part of syntax



Execution trace
↳ Sequence of program points

$S_0 \xrightarrow{P_0} S_1 \xrightarrow{P_1} S_2$ - - - $\xrightarrow{P_i} S_i$

· When we want to claim a property of a program at a prog. points it has to be true for all traces.

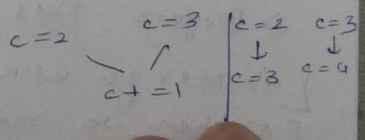


c = 10
p:
c = c + 1

— Const-folding
Constant propogation → replace const var, by value
(Inter-procedural) const prop → Do the same along all paths
↓
across fn calls

· If the compiler figures out a path in which a constant prop can be done and another where the value changes and p1 >> p2 (prob) ⇒ Hotspot to p1 and use ~~p2~~ unoptimized code, when p2 is needed

— Constant prop: cloning → can cause program to exceed cache memory making it slower



1/11/19

Lec-21

- Conservative → is c ~~a~~ constant → yes, eventhough c is not → May Causes incorrect solution

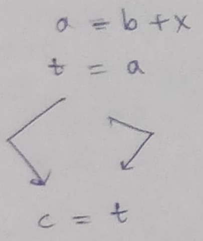
  ↓ no, eventhough c is → no prob

Available expression:

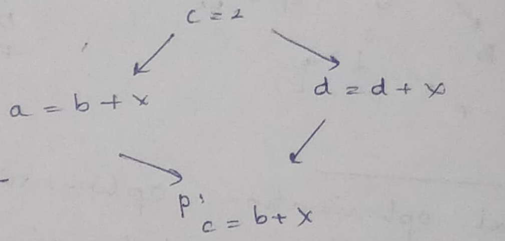
a = b + x → In SSA form

$a_0 = b_0 + x_0$ → No prob (SSA form)

so no prob if a_0, b_0, x_0 can get modified

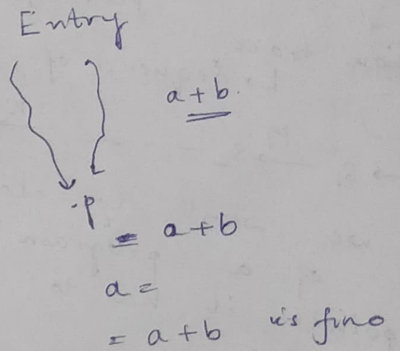


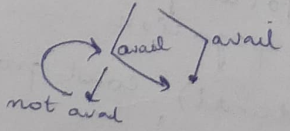
Available along one path only
So, conservatively ↳ ~~ca~~ b + x is not available



- Paths from entry to p,
- Along every path from E to P, a+b on rhs.
- After last computation of a + b, , a/b should not be in lhs
- If a/b is replaced along a path before p.

⇒





- Loop ⇒ ∞ - paths

Live variable analysis: → Useful for register allocation

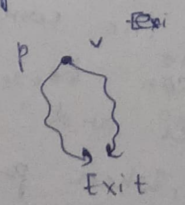
1> paths from p --> exit

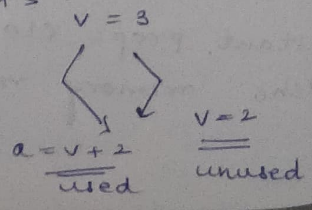
2> ∃, atleast one path where ~~a~~ v is used on RHS

3> Before this use, no re-definition on LHS

[4> Merge using union]

[5> when we merge with union, we start with an underestimate]





v = v + 2
live

v = 2

{ } } → Dead code automatically
no where used

Uday Khedkar

If we have straight line sequence of code, finding live vars

Live(Po) = (Live (P1) - {a}) ∪ {b, x}

```
Po I1 a = b + x
P1 I2 y = a + z
P2 I3
P4
```

↑ as we go up a is live at P1 and not live at Po

Live (P3) = { }.

If we are working on a fn level, and global vars are used ⇒ So, if global vars are used ⇒ we have to consider them live ⇒ We, have to assume, Live (P3) = global vars

Po b = b + x → Live

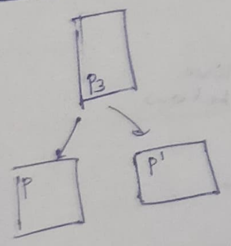
P1 y = b + x

Liv e (Po) = $\underline{\underline{\text{Live}(P_1) - \{b\}}}$ ∪ $\underline{\underline{\{b, x\}}}$ → instruction level *

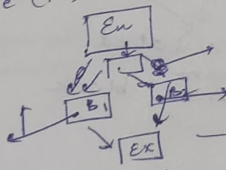
Lost gained

- Live variable analysis is a back-foard dataflow analysis

Live (P3) = Live (P) ∪ Live (p')



If CFG is DAG, ⇒



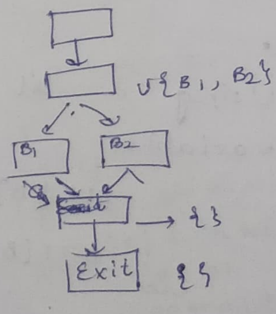
∪ {B1, B2}

{ } ↑

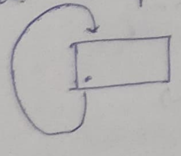
{ }

→ all glob vars | NULL set

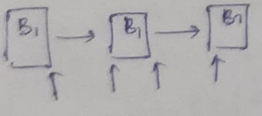
Live (Exit) = { }



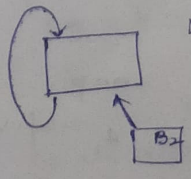
for - loops.



→ vars at beginning of basic block needs end of basic block → needs beginning needs end



- We use a fixed point algorithm



Live(B1) = Live (B1) ∪ Live(B2)
initially
{}

↓
Run until stabilization

IN[B] → set of variables live at the beginning
OUT[B] → set of vars live at the end

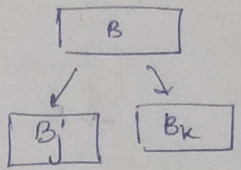
Given OUT, in can be estimated

IN [B] = (OUT[B] - KILL[B]) ∪ Gen [B]
* → Block level

```
redef   use
b  =  (b) + x
redef (y) = b + 3
          use
x  =  (y) +3
```

Gen [B] = { b, x }

$OUT[B] = \bigcup_{S \text{ is successor of } B} IN[S]$



For each block 2 conditions
+ 1 exit block condition

- If graph is DAG → no prob, start from bottom, go to top

$2x + 3y = 4$

$3x + 2y = 7$

$x = \frac{4 - 3y}{2}$

$y = \frac{7 - 3x}{2}$

x y

$$x = \frac{4 - 3\left[\frac{7-3x}{2}\right]}{2}$$

$$x = 2 - \frac{21}{4} + \frac{9x}{4}$$

5/11/19

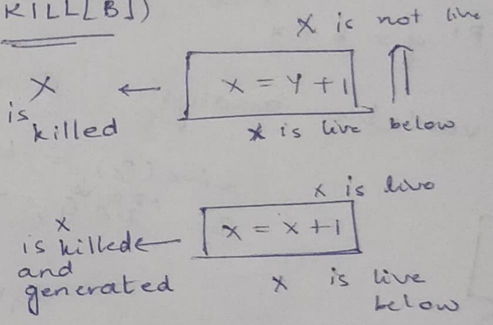
Lec-22

P
I
P'

$Live(P) = f(I, LIVE(P'))$

$IN[B] = \underline{GEN[B]} \cup (OUT[B] - \underline{KILL[B]})$

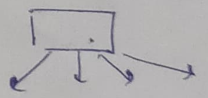
Calculated from Basic block



- Using a bit vector for live variable

$IN[B] = Gen \mid (out \;\&\; \sim KILL)$

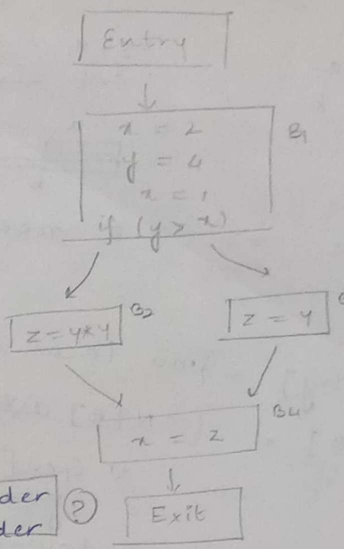
$OUT[B] = in_{S1} \mid in_{S2} \mid \ldots \mid in_{Sn}$



- If, no circular dependencies ⇒ No loops ⇒ [Practically, only in trivial fns, like ~~loops~~ get(), set()]
⇓
Toposort and find IN, GEN sets

- Circular Loop
↳ Need a fixed point iter method.

	In	out	I	O	I	O	I	O
1	/	/	y	/	y	/	z	/
2	{}	y	{y}	{z}	{y}	{z}	{z}	



IN =

. Order affects: If DAG ⇒ reverse topo sort ⇒ Least iters to converge

If not DAG ⇒ $\boxed{\text{reverse postorder order}}$ ⇒ Least iters to converge ?

. Convergence ↳ We are not going to change {0, 1}.

O (|V| |B| + 1)

IN [B1] = {a, b, c, t} ⇒ IN [Exit] = {} ; t never dies, but}

glob-var

~~does~~ Problem is because confluence operator is union union never kills

B2 ↙ B5 d is not used at all But d survives

Available Expression Analysis

1) Find rln b/w In & Out

2> Find rln with pred

~~In [B]~~ In [Entry] = { no local vars only glob vars, }

Out[B] = $\cap$ In [pi] sucessor Si

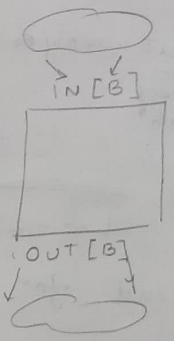
~~Out [B]~~ Out [B] = ~~Out [B]~~ $\cap$ In[B] { Gen – kill }

redefined

[{b + c} ∪ {AVAIL(p')}]

– {ele has 'a' ~~as~~ an operand

works for p a p'



p
a = b + c
p'

avail (p')
= [{b + c} ~~∪~~ {AVAIL p'}
– {e| e has 'a' as an operand}]

p
*q = b + c ←
p'

Pointers handling
Points to / alias analysis.

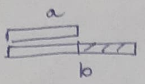
If no points to at p', we have to assume q could be pointing to any variable

Pointer arithmetic is even worse

. When higher opt, are used ~~gdb~~ src code and original code, would be very different ⇒ gdb tells that is not there random shit in your code

```
union{
    int a
  float b
   }s
```

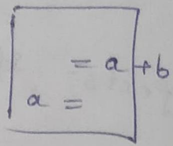
s.a → updates
s.b → partly updated



p
s.a = b+c ⇒ s.b needs to be killed to (unavailable)
p'

In [Entry] = {no local vars, only glob vars}

Out [B] = (IN [B] - kill [B])
∪ Gen[B]
↓
downwardly expressed
operands used
x not defined



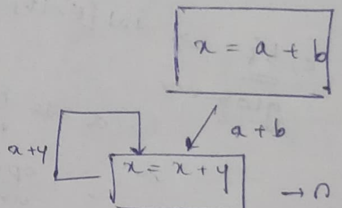
IN [B] = $\bigcap_{\text{pred}(B)}$ out [P]

- All expressions as {bit vector}

B1	B2	B3
In | 0	I | 0	I | 0
{} {a+b, a*b}		

∪ May analysis
atleast one path

∩ Must - analysis
every path



→ ∩ will never allow a+b

Out {B1} = {}
Out{1

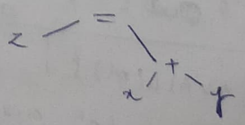
Lec - 23

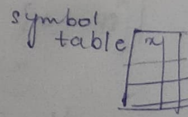
8/11/19

- llvm is the backend for many compilers

LLvm essentials

- Start with straight line sequence
- Then, loops, - -
-





Compilers :

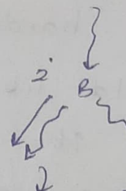
- Not only prog. skills, but the ability to look ahead & decide on use of a DS & its effect"
- Eventhough theory exists, and accessible to all compiler makers, the design decisions can determine, whether an optimization can be used practically

x86-64 is a super scalar processor

→ 1> Multiple instr at same time

2> If ~~[illegible]~~ true/false dependencies

3> → Branch prediction



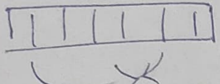
assumed that this path works

↓

~~[illegible]~~ if not roll back

- VLIW → Very long Instr word arch

Itanium uses

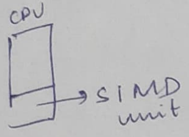


& indep subinstr

→ are executed ||ly → The compiler writer has to pack the sub-instructions ~~[illegible]~~, so that there are no deps.

Most of the slots ~~[illegible]~~ will be filled most times

- Now compiler is exploiting cores of CPU, to run faster



SIMD unit → vector processor and GPUs.

- When power supply is lost, ~~[illegible]~~ transactions stop & roll backs have to happen. But instead ~~[illegible]~~ store the transaction in a non-volatile memory

---

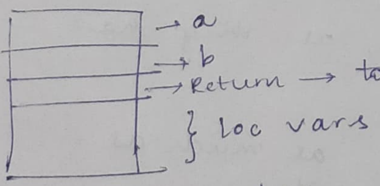
Simple code generation:

- Memory for global variables are in a global datasegment

↓ survives all fn calls, (prog exec time)

- Local vars are restricted to the ~~[illegible]~~ fn scope → stack is used, an activation record is created

↳ ~~[illegible]~~ where vars can be created



esp → top pointer

ebp → base of the act. record

↳ extended base pointer →

- data $x = y + z$

space allocated

mov, eax [X] → assembler transforms [x] into address

$x = y + z$

↳ ~~[illegible]~~ compute intensity $= 1/3$ (1 +, y, z read)

and 3 → high ~~[illegible]~~ time taking memory r/w

- We would like to improve compute intensity.

---

x = *ptr

eax = ptr

mov ebx, [eax]

mov [ebp-4] ebx

Mem → Reg → Mem

Why can't we ~~[illegible]~~ move memory to memory → Processor doesn't support → Instr that → >1 mem

Bcoz, a hardware design is hard for it

- We would like to remove unnecessary memory access.
- So, it is assumed that there are ∞ many registers called virtual registers. (vr)

$t_1 = a + b$ move vr1, [a]

Pointers can screw up code generation

↳ If &a is accessed, keep it in memory

---

arrays, pointers are not promoted

```
x = x + 1
temp = call foo  ——→ can change x (if global)       x changes in
                     If register allocation         foo
                     on fn level x ↔ vr1            x ↔ vr2
x = 2 * x
                ——→ If x is local, no prob
```

Lec-24

12/11/19

```
x = y + z  }  LD R1, @y  —→ replaced with
              LD R2, @z  —→ adress by assembler
                                for global vars
              Add R2, R1
              ST @x, R2     and by compiler for local vars
                                              ↳ 4(sp)
                                                4(gp)
                                                 global pointer
                                                 bp for
                                                 global data
                                                 section
```

For non x-86 2 source one destination operand ⇒

```
Add R3, R2, R1
ST @x, R3
```

- gcc, llvm → retargettable compilers need to make compromises on IR optimization, as they have to be applicable for all archs
- But icc, has one arch ⇒ so optimize as much as possible

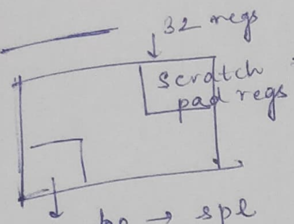
```
if flag goto L    LD R1 @flag
                  cmp R1 0
                  BEQ L
                      ↳ replaced with
                        target address
```

- 2 loads and 1 store for every operation
- Compiler has to ensure that traffic for main memory is reduced

vRi mem(local) vRk

↓ local variable don't want to promote y to mem, due to some complicatoon

In CISC, one operand can be from memory



⇒ convention followed by compiler writer

gb, bp → spl purpose regs

→ Assume 4 physical registers

vR1 - 4	r1 ↔ vr1
vR2 - 4	r2 → vr2
vR3 - 2	r3 → vr3
vR4 - 2	r4 → vr4

r5, r6 → scratch padregs

spill code → need to reduce it

- Suppose a variable is used many times in the start and ~~few times~~ nomore ~~notin~~ in later time. ⇒ we are not deallocating

. Farthest in the future → LRU approx → Page replacement

$t_1 = x - y$
↓ ↓ ↓
r_3 $r_1 \to r_2$

vr4 ← ✗
vr5 ← grab a reg from one of the variables | spill code for some variable

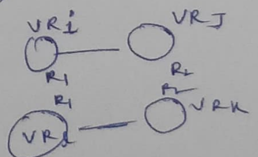
Bottom-up register allocation

↳ Unlike in OS| future is unfurled before us

↳ still need scratchpad regs. (for unsafe variables]

Live-range is calculated for all vr's

vr9, vr7 → live ranges overlap ⇒ so, need different regs.

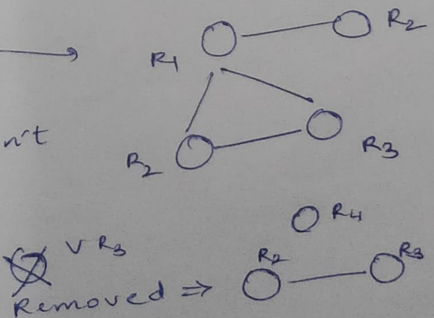


Interference Graph

If 3 regs, can we allocate them without spill code

If 2 regs for this →

Spill free code → color the graph, so that adj nodes don't have the same color



→ A variable with a big live range, if removed ⇒ can ~~remove~~ make it as ~~color~~ small as 2 color

- farthest in future works in basic blocks, but fails in control flow.

- Matrix A, B, C vs loops ( )

  A = B + C

  ↑

  Ease of prog analysis