# CS 4120
# Introduction to Compilers

Andrew Myers

Cornell University

Lecture 20: Live Variable Analysis

Lecturer: Maks Orlovich

14 Oct 09

---

# Problem

- Abstract assembly contains arbitrarily many registers $t_i$
- Want to replace all such nodes with register nodes for `e[a-d]x`, `e[sd]i`, `(ebp)`
- Local variables allocated to TEMP's too
- Only 6-7 usable registers: need to allocate multiple $t_i$ to each register
- For each statement, need to know which variables are *live* to reuse registers

---

# Using scope

- Observation: temporaries, variables have bounded scope in program
- Simple idea: use information about program scope to decide which variables are live
- Problem: overestimates liveness

```
{  int b = a + 2;
   int c = b*b;
   int d = c + 1;
   return d; }
```

← b is live

← c is live, b is not

← what is live here?

---

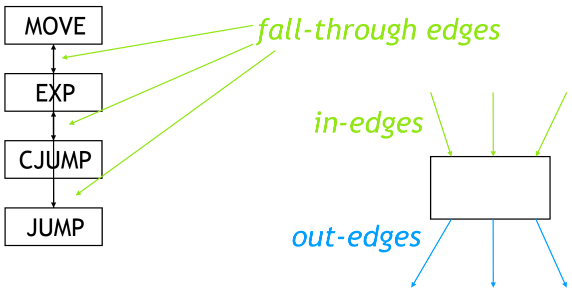
# Live variable analysis

- Goal: for each statement, identify which temporaries are live
- Analysis will be *conservative* (may over-estimate liveness, will never under-estimate)

But more *precise* than simple scope analysis (will estimate fewer live temporaries)

# Control Flow Graph

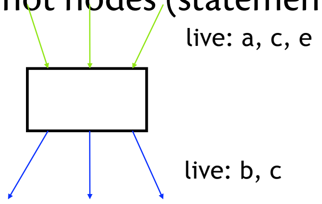
- Canonical IR forms *control flow graph* (*CFG*) : statements are nodes; jumps, fall-throughs are edges



---

# Liveness

- Liveness is associated with *edges* of control flow graph, not nodes (statements)



- Same register can be used for different temporaries manipulated by one stmt

---

# Example

a = b + 1

⇩

MOVE(TEMP(ta), TEMP(tb) + 1)

⇩

mov ta, tb
add ta, 1

Live: tb
mov ta, tb
add ta,1
Live: ta (maybe)

Register allocation: ta ⇒ eax, tb ⇒ eax
~~mov eax, eax~~
add eax, 1

---

# Use/Def

- Every statement *uses* some set of variables (reads from them) and *defines* some set of variables (writes to them)
- For statement *s* define:
  - *use*[*s*] : set of variables used by *s*
  - *def* [*s*] : set of variables defined by *s*
- Example:

| | | |
|---|---|---|
| a = b + c | *use* = b,c | *def* = a |
| a = a + 1 | *use* = a | *def* = a |

# Liveness

Variable *v* is *live* on edge *e* if:

There is

- a node *n* in the CFG that uses it *and*
- a directed path from *e* to *n* passing through no *def*

How to compute efficiently?

How to use?

# Simple algorithm: Backtracing

"variable v is *live* on edge *e* if there is a node *n* in CFG that uses it *and* a directed path from *e* to *n* passing through no *def*"

*(Slow) algorithm*: Try all paths from each *use* of a variable, tracing *backward* in the control flow graph until a *def* node or previously visited node is reached. Mark variable live on each edge traversed.

# Dataflow Analysis

- *Idea*: compute liveness for all variables simultaneously
- Approach: define *equations* that must be satisfied by any liveness determination
- Solve equations by iteratively converging on solution
- Instance of general technique for computing program properties: *dataflow analysis*

# Abstract Assembly

- Abstract assembly = assembly code w/ infinite register set
- Canonical intermediate code = abstract assembly code – except for expression trees
- MOVE($e_1$, $e_2$) ⇒ `mov e1, e2`
- JUMP($e$) ⇒ `jmp e`
- CJUMP($e$,$l$) ⇒ `cmp e1, e2`
  `[jne|je|jgt|…] l`
- CALL($e$, $e_1$,…) ⇒ `push e1`; … ; `call e`
- LABEL($l$) ⇒ `l:`

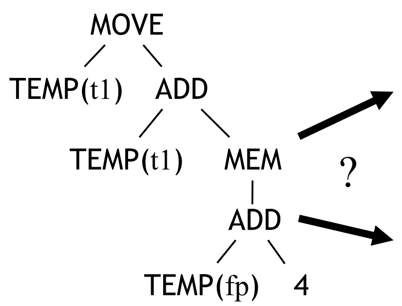
# Instruction selection

- Conversion to abstract assembly is problem of *instruction selection* for a single IR statement node
- Full abstract assembly code: glue translated instructions from each of the statements
- Problem: more than one way to translate a given statement. How to choose?

---

# Example

MOVE(TEMP(t1), TEMP(t1) + MEM(TEMP(FP)+4))



```
mov t2, fp
add t2, 4
mov t3,[t2]
add t1, t3
```

```
add t1,[fp + 4]
```

---

# Pentium ISA

- Need to map IR tree to actual machine instructions – need to know how instructions work
- Pentium is *two-address* CISC architecture
- Typical instruction has

  *opcode* (`mov`, `add`, `sub`, `shl`, `shr`, `mul`, `div`, `jmp`, `jcc`, `push`, `pop`, `test`, `enter`, `leave`, &c.)

  – *destination* (`r`, `[r]`, `[k]`, `[r+k]`, `[r1+r2]`, `[r1+w*r2]`, `[r1+w*r2+k]`)

  (**may also be an operand)**

  – *source* (any legal destination, or a constant)

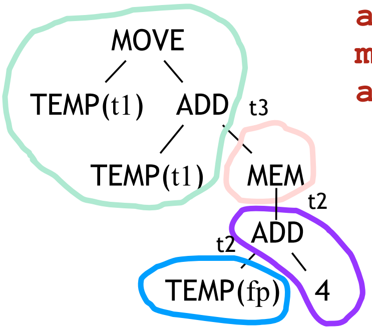
```
                         opcode   dest   src
mov eax,1                    add ebx,ecx
sub esi,[ebp]  add [ecx+16*edi],edi
je label1                    jmp [fp+4]
```

---

# Tiling

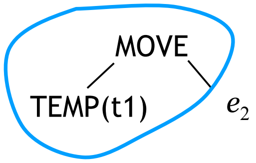
- Idea: each Pentium instruction performs computation for a piece of the IR tree: a *tile*

```
mov t2, ebp
add t2, 4
mov t3,[t2]
add t1, t3
```

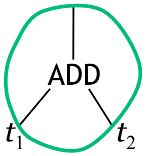


- Tiles connected by new temporary registers (t2, t3) that hold result of tile

# Some tiles

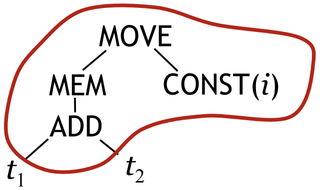


```
mov t1, t2
```



```
mov tf, t1
add tf, t2
```

(`tf` a *fresh* temporary)



```
mov [t1+t2], i
```

---

# Problem

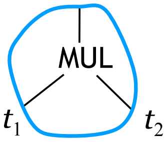
- How to pick tiles that cover IR statement tree with minimum execution time?
- Need a good selection of tiles
  - small tiles to make sure we can tile every tree
  - large tiles for efficiency
- Usually want to pick large tiles: fewer instructions
- Pentium: RISC core instructions take 1 cycle, other instructions may take more

```
add [ecx+4], eax   ⇔   mov edx,[ecx+4]
                       add edx,eax
                       mov [ecx+4],eax
```

---

# An annoying instruction

- Pentium mul instruction multiples single operand by **eax**, puts result in **eax** (low 32 bits), **edx** (high 32 bits)
- Solution: add extra **mov** instructions, let register allocation deal with **edx** overwrite

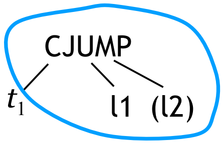


```
mov eax, t1
mul t2
mov tf, eax
```

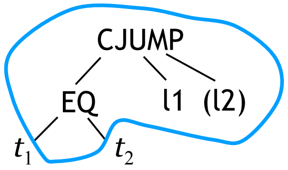
---

# Branches

- How to tile a conditional jump?
- Fold comparison operator into tile



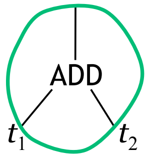
```
test t1
 jnz l1
```



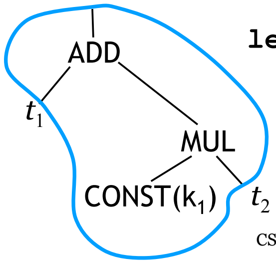
```
cmp t1, t2
je l1
```

# More handy tiles

**`lea`** instruction computes a memory address but doesn't actually load from memory



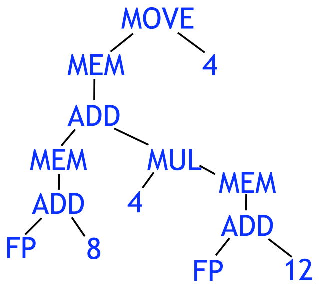
$\texttt{lea } t_f\texttt{, [}t_1\texttt{+}t_2\texttt{]}$ ($t_f$ a *fresh* temporary)



$\texttt{lea } t_f\texttt{, [}t_1\texttt{+}k_1\texttt{*}t_2\texttt{]}$ ($k_1$ `one of 2,4,8,16`)

# Greedy tiling

- Assume larger tiles = better
- Greedy algorithm: start from top of tree and use largest tile that matches tree
- Tile remaining subtrees recursively



# How good is it?

*Very* rough approximation on modern pipelined architectures: execution time is number of tiles

Greedy tiling (Appel: "maximal munch") finds an *optimal* but not necessarily *optimum* tiling: cannot combine two tiles into a lower-cost tile

- We *can* find the optimum tiling using dynamic programming!

# Dataflow values

$use[n]$ : set of variables used by $n$

$def[n]$ : set of variables defined by $n$

$in[n]$ : variables live on entry to $n$

$out[n]$ : variables live on exit from $n$

Clearly: $in[n] \supseteq use[n]$

What other constraints are there?

# Dataflow constraints

$in[n] \supseteq use[n]$

- A variable must be live on entry to $n$ if it is used by the statement itself

$in[n] \supseteq out[n] - def[n]$

- If a variable is live on output and the statement does not define it, it must be live on input too

$out[n] \supseteq in[n']$ if $n' \in succ[n]$

- if live on input to n', must be live on output from n

---

# Iterative dataflow analysis

- Initial assignment to $in[n]$, $out[n]$ is empty set ∅ : will not satisfy constraints

$$in[n] \supseteq use[n]$$
$$in[n] \supseteq out[n] - def[n]$$
$$out[n] \supseteq in[n'] \quad \text{if} \quad n' \in succ[n]$$

- Idea: iteratively re-compute in[n], out[n] when forced to by constraints. Live variable sets will increase monotonically.
- Dataflow equations:

$$in'[n] = use[n] \cup (out[n] - def[n])$$
$$out'[n] = \bigcup_{n' \in succ[n]} in[n']$$

---

# Complete algorithm

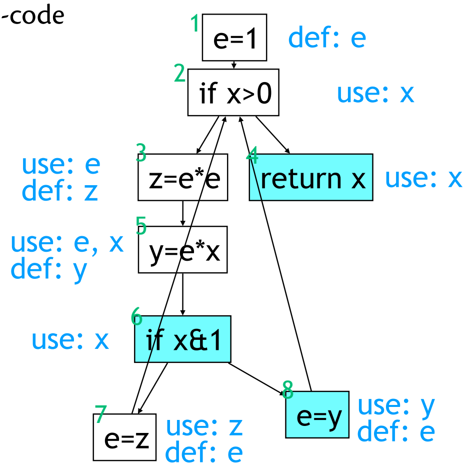
```
for all n, in[n] = out[n] = ∅
repeat until no change
    for all n
            $out[n] = \bigcup_{n' \in succ[n]} in[n']$
            $in[n] = use[n] \cup (out[n] - def[n])$
    end
end
```

- Finds *fixed point* of in, out equations
- Problem: does extra work recomputing in, out values when no change can happen

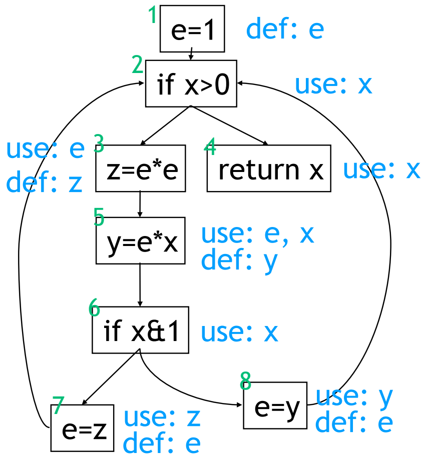
---

# Example

- For simplicity: pseudo-code



# Example



2: in={x}
3: in={e}
4: in={x}
5: in={e,x}
6: in={x}
7: out={x}, in={x,z}
8: out={x}, in={x,y}
1: out={x}, in={x}
2: out={e,x}, in={e,x}
3: out={e,x}, in={e,x}
5: out={x}, in={e,x}
6: out={x,y,z}, in={x,y,z}
7: out={e,x}, in={x,z}
8: out={e,x}, in={x,y}
1: out={e,x}, in={x}
5: out={x,y,z}, in={e,x,z}
3: out={e,x,z}, in={e,x}
*all equations satisfied*

# Faster algorithm

- Information only propagates between nodes because of this equation:

$$out[n] = \bigcup_{n' \in succ\ [n]} in[n']$$

- Node is updated from its successors
  - If successors haven't changed, no need to apply equation for node
  - Should start with nodes at "end" and work backward

# Worklist algorithm

- Idea: keep track of nodes that might need to be updated in *worklist* **:** FIFO queue

```
for all n, in[n] = out[n] = Ø
w = { set of all nodes }
repeat until w empty
    n = w.pop( )
    out[n] = $\bigcup_{n' \in succ\ [n]} in[n']$
    in[n] = use[n] ∪ (out[n] — def [n])
    if change to in[n],
        for all predecessors m of n, w.push(m)
end
```