**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

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

- $\text{b is live}$
- $\text{c is live, b is not}$
- $\text{what is live here?}$

**Live variable analysis**

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

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

![Diagram of control flow graph with nodes and edges labeled]

**Liveness**

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

![Example liveness diagram]

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

**Example**

```plaintext
a = b + 1
MOVE(TEMP(ta), TEMP(tb) + 1)
mov ta, tb
add ta, 1
```

Live: ta (maybe)

Register allocation: ta ⇒ eax, tb ⇒ eax

```plaintext
mov eax, ebx
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:
  ```plaintext
  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?

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
• $\text{MOVE}(e_v, e_2) \Rightarrow \text{mov } e_1, e_2$
• $\text{JUMP}(e) \Rightarrow \text{jmp } e$
• $\text{CJUMP}(e, l) \Rightarrow \text{cmp } e_1, e_2$
• [jne|je|jgt|... $l$
• $\text{CALL}(e, e_1,...) \Rightarrow \text{push } e_1;...;\text{call } e$
• $\text{LABEL}(l) \Rightarrow 1$

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.
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],[r+k],[r+r2],
    [r1*w*r2],[r1+w*r2+k])
    (may also be an operand)
  - source (any legal destination, or a constant)

Tiling

• Idea: each Pentium instruction performs computation for a piece of the IR tree: a tile
  - Tiles connected by new temporary registers (t2, t3) that hold result of tile
Some tiles

- **MOVE**
  - `mov t1, t2`

- **ADD**
  - `mov t_f, t1` (t_f a fresh temporary)
  - `add t_f, t2`

- **MEM**
  - `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 t_f, eax
```

Branches

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

lea instruction computes a memory address but doesn’t actually load from memory

\[
\text{lea } t_\epsilon, [t_1 + t_2] \quad (t_\epsilon \text{ a fresh temporary})
\]

\[
\text{lea } t_\epsilon, [t_1 + k_1 \cdot t_2] \quad (k_1 \text{ one of } 2, 4, 8, 16)
\]

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

- \( \text{use}[n] \): set of variables used by \( n \)
- \( \text{def}[n] \): set of variables defined by \( n \)
- \( \text{in}[n] \): variables live on entry to \( n \)
- \( \text{out}[n] \): variables live on exit from \( n \)

Clearly: \( \text{in}[n] \supseteq \text{use}[n] \)

What other constraints are there?
**Dataflow constraints**

\[ \text{in}[n] \supseteq \text{use}[n] \]

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

\[ \text{in}[n] \supseteq \text{out}[n] - \text{def} [n] \]

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

\[ \text{out}[n] \supseteq \text{in}[n'] \quad \text{if} \quad n' \in \text{succ} [n] \]

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

**Complete algorithm**

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

- Finds fixed point of \( \text{in}, \text{out} \) equations
- Problem: does extra work recomputing \( \text{in}, \text{out} \) values when no change can happen

**Iterative dataflow analysis**

- Initial assignment to \( \text{in}[n] \), \( \text{out}[n] \) is empty set \( \emptyset \) : will not satisfy constraints

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

- Idea: iteratively re-compute \( \text{in}[n] \), \( \text{out}[n] \) when forced to by constraints. Live variable sets will increase monotonically.

- Dataflow equations:

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

**Example**

- For simplicity: pseudo-code

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

- Finds fixed point of \( \text{in}, \text{out} \) equations
- Problem: does extra work recomputing \( \text{in}, \text{out} \) values when no change can happen
**Example**

```
Example

1: e=1
2: if x>0
3: z=e*e
4: y=e*x
5: e=z
6: if x&1
7: e=y
8: return x

---

**Faster algorithm**

- Information only propagates between nodes because of this equation:
  
  $\text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n']$

- 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

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