Software Programmable DSP Platform Analysis

Episode 5, Tuesday 19 April 2005, Ingredients

Liveness Analysis

Control-Flow Graphs
Definition & Use
Calculation of Liveness
Interference Graphs

Register Allocation

Coloring by Simplification
Spilling

CI6x Compiler Intrinsics

From Abstract To Concrete Registers

- Instruction selection has left us with an assembly program that uses abstract registers (unboundedly many).
- But target architecture only has a small fixed set of registers...
- We want to map numerous temporaries (TEMP) into as few concrete registers as possible.
- Obviously we can only assign the same register to two temporaries, if we do not need both of them at the same time.

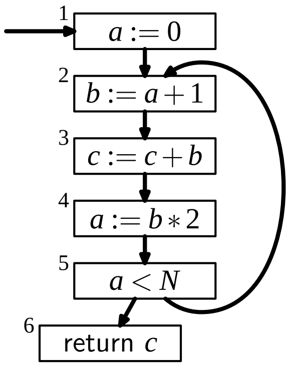
Liveness Analysis

- Identify temporaries that cannot be active at the same time.
- This is achieved by liveness analysis.
- Liveness analysis works on control flow graphs.
- In practice the flow graph is created from the abstract machine program,
- but for clarity of presentation we shall use simple language of expressions and assignments in this lecture.

Control-Flow Graphs

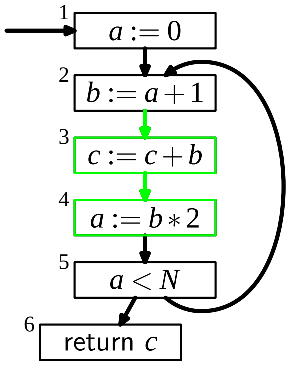
- Each statement is a node
- An edge from node x to y if statement x can be directly followed by y during execution.

```
       $a \leftarrow 0$
$L_1$: $b \leftarrow a+1$
       $c \leftarrow c+b$
       $a \leftarrow b*2$
       if $a < N$ goto $L_1$
       return $c$
```

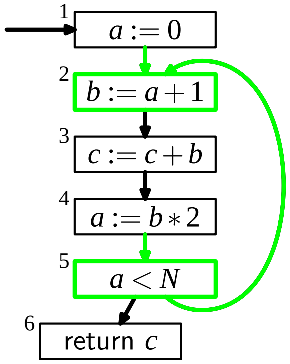


Live Variable

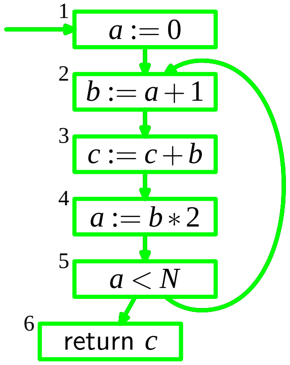
A variable is live at a given program point if its current value may be used in later execution.



- *b* is live in node 4.
- 4 does not define *b* so *b* is live on input to 4.
- 3 does not define *b* so *b* is live in 3 and on all edges incoming.
- 3 defines *b* and does not use it. *b* is not live in 2. Live range of *b* is $\{2 \to 3, 3 \to 4\}$.

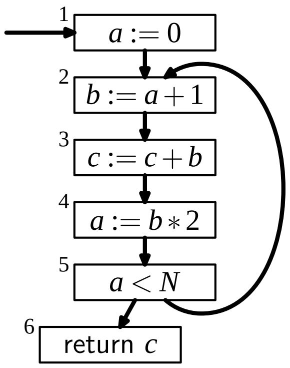


- *a* is live in nodes $2, 5$.
- Take one step back.
- 4 and 1 kill *a*.
- Live range of *a* is $\{1 \to 2, 4 \to 5 \to 2\}$.
- Note: the value of *a* in node 3 is completely useless.



- *c* is used in $3, 6$
- One step back.
- Another one back.
- Note that *c* is live both on into and out from 3, as it is both defined and used in 3.
- *c* is live on entry to 1. If *c* is not a parameter, then this is a bug (uninitialized variable).

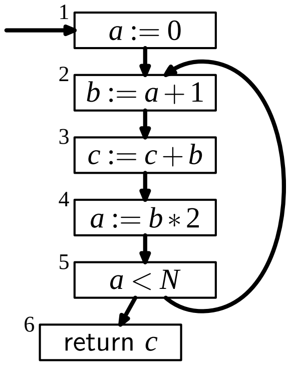
- *out-edges*$[n]$: all edges that lead to a successor node of *n*.
- *in-edges*$[n]$: all edges that lead from a predecessor node of *n*.
- *pred*$[n]$: set of all predecessors of *n*.
- *succ*$[n]$: set of all successors of *n*.



$$\textit{out-edges}[5] = \{5 \to 6, 5 \to 2\} \qquad \textit{succ}[5] = \{2,6\}$$
$$\textit{in-edges}[2] = \{5 \to 2, 1 \to 2\} \qquad \textit{pred}[2] = \{1,5\}$$

Definition & Use

- An assignment to a variable x **defines** x.
- An occurrence of x on the right hand side of the assignment is called a **use** of x.



- $\textit{def}(3) = \{c\}$
- $\textit{use}(3) = \{b,c\}$

Liveness

Variable x is **live** on the given edge if there exists a directed path from that edge to a use that does not go through any def.

X is **live-in** in node n if it is live on any of its *in-edges*.

X is **live-out** in node n if it is live on any of its *out-edges*.

Calculation of Liveness

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

- initialize all $\textit{in}[n]$ and $\textit{out}[n]$ sets to be empty
- compute new sets interpreting equality like assignments
- repeat the previous step until no growth is observed in the sets.

The result for our running example is

node	*live-in*	*live-out*
1	c	ac
2	ca	bc
3	bc	bc
4	bc	ac
5	ac	ac
6	c	

Interference Graphs

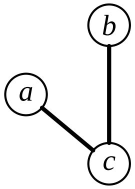
- Variables *a* and *b* are in interference if *a* and *b* cannot be allocated in the same memory space (a register).
- Overlapping live ranges cause interference.
- Architecture constraints may cause interferences (for example registers participating in some instruction cannot be from two different register files).

The following are our live ranges:

node	*live-in*	*live-out*
1	c	ac
2	ca	bc
3	bc	bc
4	bc	ac
5	ac	ac
6	c	

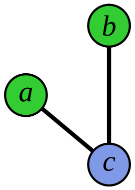
- We can see from this that *a* interferes with *b*
- and *b* interferes with *c*,
- but *a* does not interfere with *c*.

The same information presented as an *interference graph*:



Register Allocation

Assign as few platform registers to many temporaries: do this by assigning a minimal number of colors to nodes of interference graph, such that any neighboring vertices have different colors.



A and *b* have been allocated in the same register.

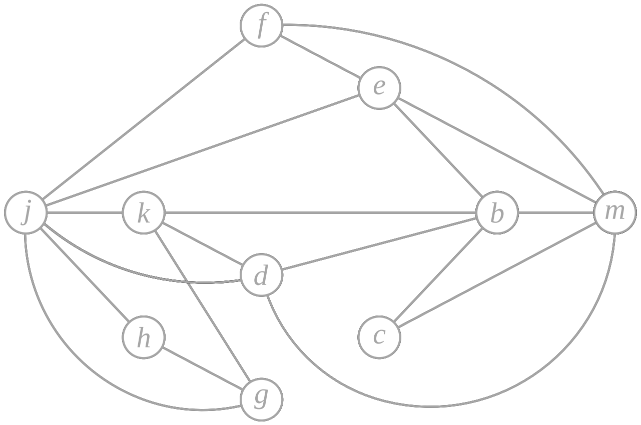
Coloring by Simplification

[Kempe 1879]

This is a coloring algorithm based on heuristics (i.e. does not guarantee optimality):

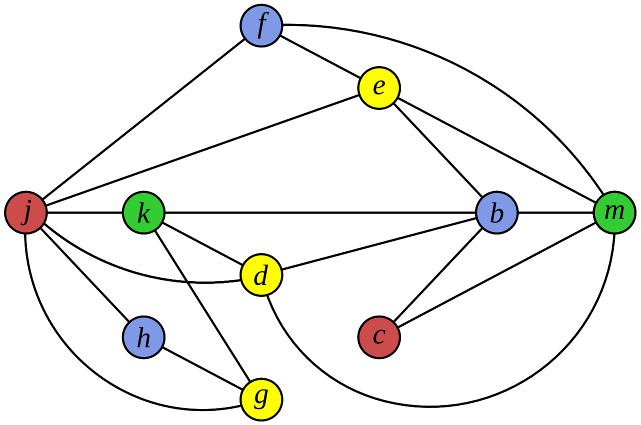
- Assume K registers (colors) are available.
- Find a node m with less than K neighbors.
- Remove m from the graph (it will be easy to add it and color, since it has less than K members).
- Repeat previous step until you end up with isolated nodes.
- Assign them the first color,
- and add nodes back to the graph in the reverse order, adding colors on the fly.

Simplifying



stack: g h k d j e f b c m (source: Appel, p. 237–238)

Selecting



stack: g h k d j e f b c m (source: Appel, p. 237–238)

Spilling

- Colouring by simplification may fail if the interference graph is not k-colourable.
- If all nodes left in the graph have degrees higher than k, an arbitrary node n has to be removed from the graph (potential spill).
- But since the algorithm cannot be really sure if this is a real spill, we put the node on the stack hoping that we can still colour this with just k colours during selection.
- If selection manages to colour n then fine.
- If neighbours of n already use k: **actual spill**.
- n has to be stored in memory.

- We ignore the spill during the main run and continue to find all other spills.
- Code is rewritten to fetch and store from memory for each definition and use.
- Then liveness analysis and colouring has to be rerun, as the interference graph has changed (the new code uses new temporaries).
- Usually this process succeeds after one or two iterations.

On Choosing Colors

- A local variable that is not live across the call should be allocated to the caller save registers (so only choose from a subsset of colours).
- Similarly a local variable that is live across several calls should be stored in a callee save register to avoid multiple saves.
- Register allocation for threes (side-effect free expressions) can be done much more efficiently, see Appel p. 257.

Compiler Intrinsics [cl6x specific!]

- Intrinsics are special functions that map directly to inlined C67x instructions.
- They look like a function call.
- Name starts with an underscore.
- Instrinsics are directly compiled to special instructions.
- Exhaustive list available in section 2.4.1 of spr198.(Programmer's Guide).

Saturated Addition in Standard C

```
int sadd(int a, int b) {
  int r;
  r = a + b;
  if (((a^b) & 0x80000000) == 0) {
    if ((r^a) & 0x80000000) {
      r = (a<0) ? 0x80000000:0x7fffffff;
    }
  }
  return r; }
```

Many, many cycles...

Saturated Addition Intrinsic

In Cl6x you can achieve the same effect with:

```
r = _sadd(a,b);
```

- translated directly to SADD instruction [spru189,3-108]
- no stack frame, entry code, exit code
- efficient execution (1 cycle)
- disadvantage: portability suffers (but C implementations are provided for workstation testing, profiling and compilations with other compilers).

How is this implemented?

Most likely

- intrinsics are already recognized at the abstract syntax level,
- and then are represented in IR by special IR-instructions.
- this isntructions map directly to target instructions during instruction selection,
- so code for function calls is not generated.