Register Allocation (Continued)

March 21, 2001

CS 132: Compiler Design

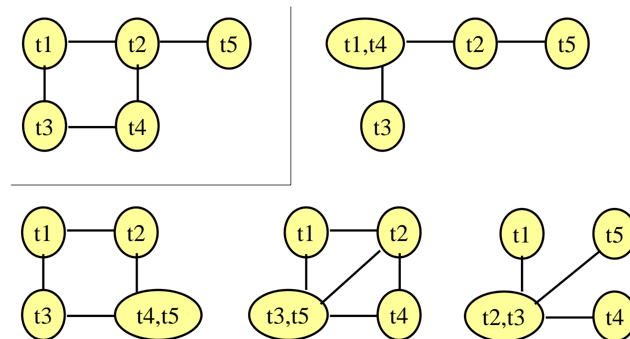
Handling Move/Copy Instructions

- Suppose code contains the move `a ← b`.
  - Graph construction algorithm avoids unnecessary interference between source and destination of moves.
  - Other instructions may force interference)
- Suppose `a` and `b` turn out not to interfere.
  - If possible, we'd like to put both `a` and `b` in the same register
  - Then the copy is a no-op and can be eliminated.

Coalescing

- Declaring that a and b must be allocated the same register equivalent to merging their nodes in the interference graph.
  - Nodes get coalesced into a single node
  - Hence this process is called *coalescing*
- Single node replaces two nodes
  - Interferes with any node that either of the two original nodes interfered with.

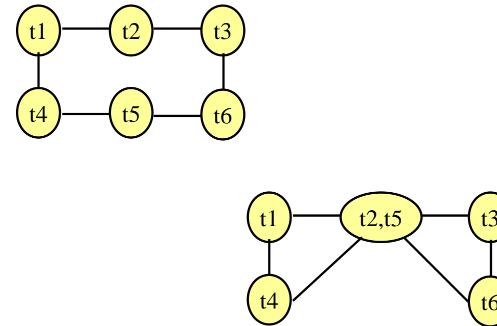
Coalescing Example



Aggressive Coalescing

- Any two non-interfering nodes can be coalesced.
- Chaitin:
  - Coalesce the source and destination of every copy, as long as they don't interfere.
- But is this a good idea?

Another Coalescing Example



Coalescing Strategies

- Spill code is usually more expensive than a move, because it involves memory access
- Hence many allocators are very conservative about coalescing
  - Only coalesce if we can guarantee that it does not increase the number of registers required to color the graph.

Conservative Coalescing Heuristics

- Briggs:
  - Two nodes can be coalesced if the resulting node has $<k$ neighbors with $\geq k$ edges.
- George:
  - Two nodes `a` and `b` can be coalesced if, for every neighbor `t` of `a`, either t is a neighbor of `b`, or else `t` has degree $<k$.

Biased Coloring

- We can apply the same "optimism" trick to coloring as we did with spilling.
  - Some nodes can be safely coalesced but the conservative heuristics can't guarantee it safe.
  - So, don't coalesce, but when choosing a color for a node, look to see if there's one that would eliminate a move operation.
- Independent of any coalescing phase.

Graph Coloring with Traditional Coalescing

1. Build the interference graph.
2. Apply conservative coalescing to move-related nodes.
3. Repeatedly remove nodes from the graph (preferentially removing nodes of degree $<k$).
4. Color the simplified nodes in reverse order.
5. Insert spill code and repeat, if necessary.
6. Replace references to temporaries with references to registers.

Iterated Coalescing

- Observation [George and Appel]
  - It's safe to do coalescing even after some of the graph nodes have been removed.
  - After nodes are removed, the conservative coalescing heuristics might apply in more instances.

Iterated Coalescing

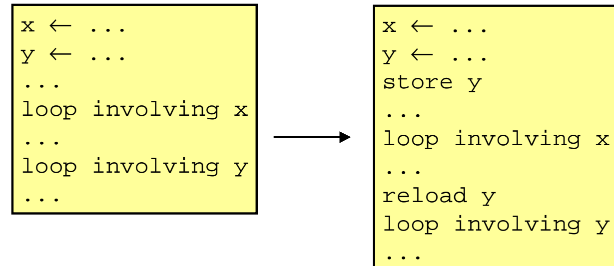
- When simplifying the graph, do the following (ordered by decreasing priority)
  1. Remove a non move-related node of degree $<k$
  2. Conservatively coalesce pair of move-related nodes
  3. Remove a move-related node of degree $<k$
     - Give up on trying to coalesce it.
  4. Pull out a high-degree node
     - as a potential spill
- Note: Coalescing may create non move-related nodes with low degree, etc.

Live Range Splitting

- Clearly not always optimal to require that a variable be either always in the same register or always in memory.
- *Live range splitting* breaks the lifetime of a variable into separate pieces, each of which can be separately allocated.

Live Range Splitting Example

- Assume only one register available for `x` and `y`.



Live Range Splitting

- Idea is at least a decade old [Chow and Hennessy 90] but still problematic
  - Which live ranges to split?
  - Where to insert splits?
- Still an open problem how to do this well (without substantially increasing allocation time)
  - Some positive results in research papers, but I don't know of any dominant approach.

Stepping Back

- What are we actually applying the register allocation algorithm to?
  - A single expression
  - A single basic block: *local* allocation.
  - An entire procedure: *global* allocation.
  - Many procedures: *interprocedural* allocation.

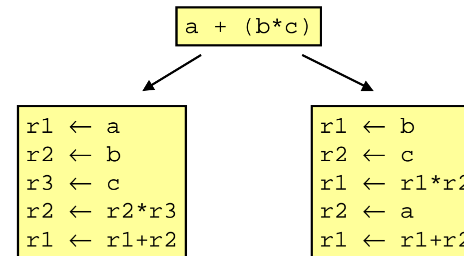
Sethi-Ullman Numbering

- Assume that all variables in a complex expression are stored in memory.

`a + (b*c)`

How many registers are required to evaluate it (assuming no side effects)?

Sethi-Ullman Example



Interference Revisited

- What does the interference graph look like?

```
a ← x+y
b ← f(x,3)
c ← a+1
```

Interprocedural Allocation

- What benefit is there by considering several procedures (or the whole program) at once?
  - Permits non-standard calling conventions
  - More precise caller-save/callee-save distinction.