Low-Level Issues

Last lecture

- Interprocedural analysis

Today

- Start low-level issues
- Register allocation

Later

- More register allocation
- Instruction scheduling

Register Allocation

Problem

- Assign an unbounded number of **symbolic** registers to a fixed number of **architectural** registers (which might get renamed by the hardware to some number of **physical** registers)
- Simultaneously live data must be assigned to different architectural registers

Goal

- Minimize overhead of accessing data
  - Memory operations (loads & stores)
  - Register moves

Scope of Register Allocation

Expression

Local

Loop

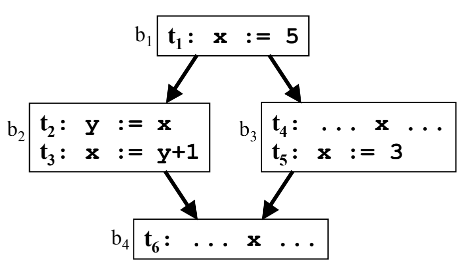
Global

Interprocedural

Granularity of Allocation

What **is allocated to registers?**

- Variables
- Live ranges/Webs (*i.e.,* du-chains with common uses)
- Values (*i.e.,* definitions; same as variables with SSA & copy propagation)



Variables: 2 (**x** & **y**)
Live Ranges/Web: 3 ($t_1 \to t_2, t_4$; $t_2 \to t_3$; $t_3, t_5 \to t_6$)
Values: 4 (t_1, t_2, t_3, t_5, $\phi(t_3, t_5)$)

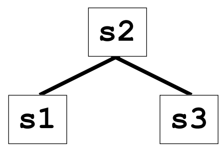
What are the tradeoffs?

Each allocation unit is given a symbolic register name (*e.g.,* `s1`, `s2`, *etc.*)

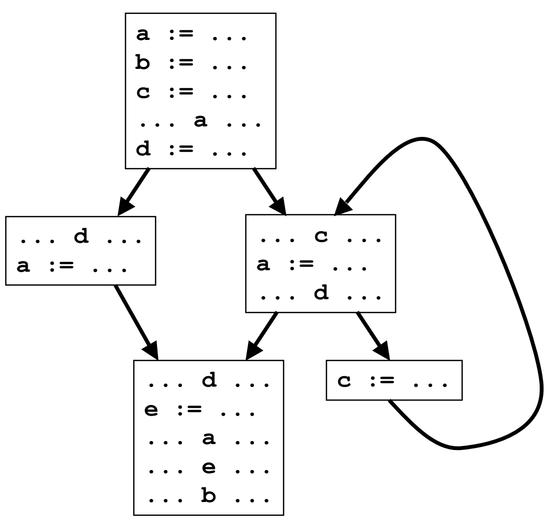
Global Register Allocation by Graph Coloring

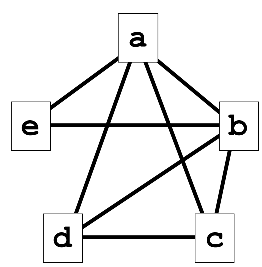
Idea [Cocke 71], First allocator [Chaitin 81]

1. Construct **interference graph** $G=(N,E)$
   - Represents notion of "simultaneously live"
   - Nodes are units of allocation (*e.g.,* variables, live ranges/webs)
   - $\exists$ edge $(n_1,n_2) \in E$ if n_1 and n_2 are simultaneously live
   - Symmetric (not reflexive nor transitive)
2. Find ***k*-coloring** of G (for k registers)
   - Adjacent nodes can't have same color
3. **Allocate** the same register to all allocation units of the same color
   - Adjacent nodes must be allocated to distinct registers

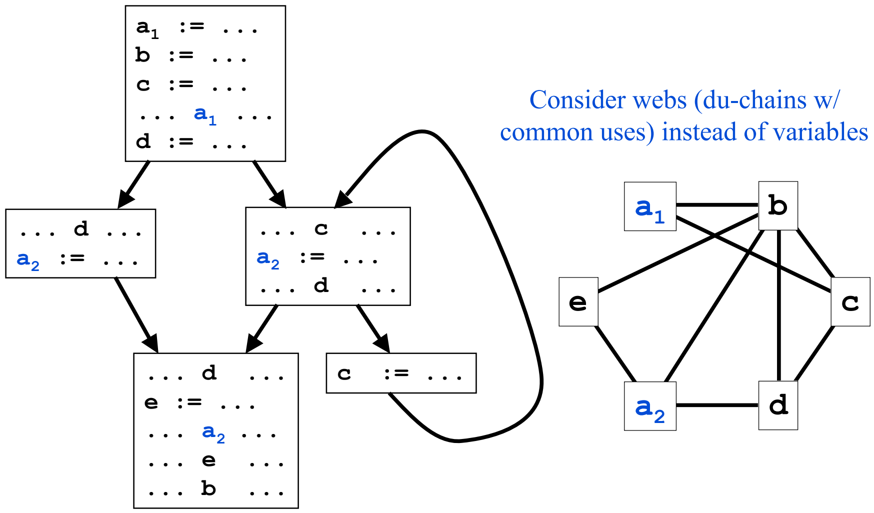


Interference Graph Example (Variables)





Interference Graph Example (Webs)



Computing the Interference Graph

Use results of live variable analysis

> **for each** symbolic-register $\mathbf{s_i}$ **do**
>   **for each** symbolic-register $\mathbf{s_j}$ ($\mathbf{j < i}$) **do**
>   **for each** def $\in$ {definitions of $\mathbf{s_i}$} **do**
>   **if** ($\mathbf{s_j}$ is live at def) **then**
> $E \leftarrow E \cup (\mathbf{s_i}, \mathbf{s_j})$

Coalescing

Move instructions

- Code generation can produce unnecessary move instructions

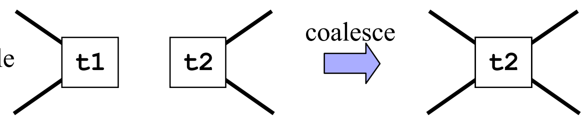
  `mov t1, t2`
- If we can assign `t1` and `t2` to the same register, we can eliminate the move

Idea

- If `t1` and `t2` are not connected in the interference graph, **coalesce** them into a single variable

Problem

- Coalescing can increase the number of edges and make a graph uncolorable
- Limit coalescing to avoid uncolorable graphs



Allocating Registers Using the Interference Graph

***K*-coloring**

- Color graph nodes using up to *k* colors
- Adjacent nodes must have different colors

Allocating to *k* registers ≡ finding a *k*-coloring of the interference graph

- Adjacent nodes must be allocated to distinct registers

But. . .

- Optimal graph coloring is NP-complete
  - Register allocation is NP-complete, too (must approximate)
- What if we can't *k*-color a graph? (must **spill**)

Spilling

If we can't find a k-coloring of the interference graph

- Spill variables (nodes) until the graph is colorable

Choosing variables to spill

- Choose least frequently accessed variables
- Break ties by choosing nodes with the most conflicts in the interference graph
- Yes, these are heuristics!

Weighted Interference Graph

Goal

- Weight(s) = $\sum_{\forall \text{ references } r \text{ of } s} f(r)$ $f(r)$ is execution frequency of r

Static approximation

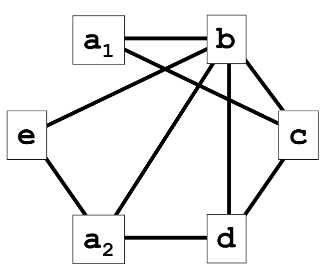
- Use some reasonable scheme to rank variables
- One possibility
  - Weight(s) = 1
  - Nodes after branch: ½ weight of branch
  - Nodes in loop: 10 × weight of nodes outside loop

Simple Greedy Algorithm for Register Allocation

for each $n \in N$ **do** { select n in decreasing order of weight }
    if n can be colored **then**
        do it { reserve a register for n }
    else
        Remove n (and its edges) from graph { allocate n to stack (spill) }

Example

Attempt to 3-color this graph (, ,)

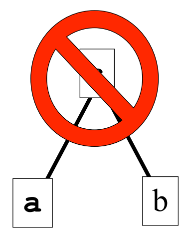


Weighted order:
a_1
b
c
d
a_2
e

What if you use a different weighting?

Example

Attempt to 2-color this graph (,)



Weighted order:
`a`
`b`
`c`

Improvement #1: Simplification Phase [Chaitin 81]

Idea

- Nodes with < k neighbors are guaranteed colorable

Remove them from the graph first

- Reduces the degree of the remaining nodes

Must spill only when all remaining nodes have degree ≥ k

Algorithm [Chaitin81]

while interference graph not empty **do**
    while ∃ a node *n* with < *k* neighbors **do**
        Remove *n* from the graph   simplify
        Push *n* on a stack
    if any nodes remain in the graph **then** { *blocked* with >= k edges }
        Pick a node *n* to spill   { lowest spill-cost or }   spill
        Add *n* to spill set   { highest degree }
        Remove *n* from the graph
if spill set not empty **then**
    Insert spill code for all spilled nodes   { store after def; load before use }
    Reconstruct interference graph & start over
while stack not empty **do**
        Pop node *n* from stack   color
        Allocate *n* to a register

More on Spilling

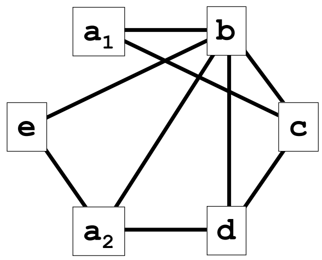
Chaitin's algorithm restarts the whole process on spill
- Necessary, because spill code (loads/stores) uses registers
- Okay, because it usually only happens a couple times

Alternative
- Reserve 2-3 registers for spilling
- Don't need to start over
- But have fewer registers to work with

Example

Attempt to 3-color this graph (, ,)

Stack:
d
c
b
a_2
a_1
e



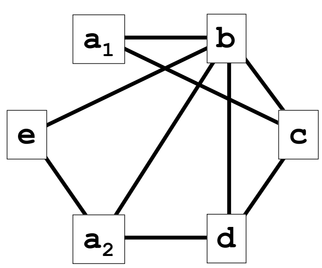
Weighted order:
e
a_1
a_2
b
c
d

Example

Attempt to 2-color this graph (,)

Spill Set:
e
a_1
a_2
b

Stack:
d
c

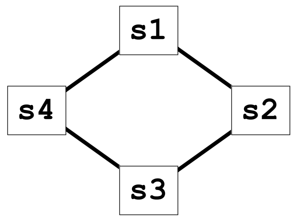


Weighted order:
e
a_1
a_2
b
c
d

Many nodes remain uncolored even though we could clearly do better

The Problem: Worst Case Assumptions

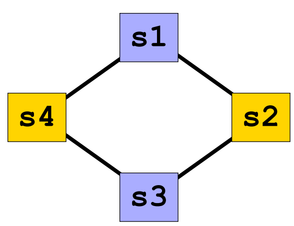
Is the following graph 2-colorable?



Clearly 2-colorable

- But Chaitin's algorithm leads to an immediate block and spill
- The algorithm assumes the worst case, namely, that all neighbors will be assigned a different color

Improvement #2: Optimistic Spilling [Briggs 89]



Idea

- Some neighbors might get the same color
- Nodes with *k* neighbors **might** be colorable
- Blocking does not imply that spilling is necessary
  - Push blocked nodes on stack (rather than place in spill set)
  - Check colorability upon popping the stack, when more information is available

Defer decision

Algorithm [Briggs *et al.* 89]

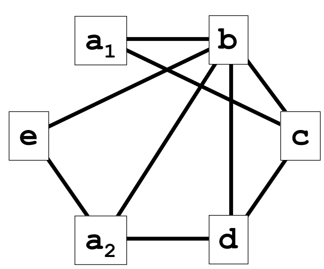
while interference graph not empty **do**
    while ∃ a node *n* with < *k* neighbors **do** } simplify
        Remove *n* from the graph
        Push *n* on a stack
    if any nodes remain in the graph **then** { *blocked* with >= k edges }
        Pick a node *n* to spill { lowest spill-cost/highest degree }
        Push *n* on stack } defer decision
        Remove *n* from the graph
while stack not empty **do**
    Pop node *n* from stack } make decision
    if n is colorable **then**
        Allocate *n* to a register
    else
        Insert spill code for *n* { Store after def; load before use }
        Reconstruct interference graph & start over

Example

Attempt to 2-color this graph (,)

Stack:
d
c
b*
a_2*
a_1*
e*

*** blocked node**



Weighted order:
e
a_1
a_2
b
c
d

Improvement #3: Live Range Splitting [Chow & Hennessy 84]

Idea

- Start with variables as our allocation unit
- When a variable can't be allocated, split it into multiple subranges for separate allocation
- Selective spilling: put some subranges in registers, some in memory
- Insert memory operations at boundaries

Why is this a good idea?

Improvement #4: Rematerialization [Chaitin 82]&[Briggs 84]

Idea

- Selectively re-compute values rather than loading from memory
- "Reverse CSE"

Easy case

- Value can be computed in single instruction, and
- All operands are available

Examples

- Constants
- Addresses of global variables
- Addresses of local variables (on stack)

Next Time

Lecture

- More register allocation
  - Allocation across procedure calls