Traditional Uses of Compilers

Last lecture
- Optimizing OO languages

Today
- Start low-level issues
- Register allocation

Register Allocation

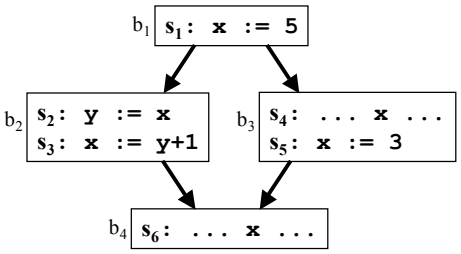
Problem
- Assign an unbounded number of **symbolic** registers to a small, fixed number of **architectural** registers (which might get renamed by the hardware to some number of **physical** registers)
- Simultaneously live data must be assigned to different architectural registers

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

Granularity of Allocation

***What* is allocated to registers?**

- Variables
- Live ranges (*i.e.,* set of basic blocks in which a variable is live)
- Values (*i.e.,* definitions; same as variables with SSA)
- Webs (*i.e.,* du-chains with common uses)



Variables: 2 (**x** & **y**)
Live ranges: 2 ($\{b_1,b_2,b_3,b_4\}$, $\{b_2\}$)
Values: 4 (s_1, s_2, s_3, s_5, ϕ (s_3,s_5))
Web: 3 ($s_1 \rightarrow s_2,s_4$; $s_2 \rightarrow s_3$; $s_3,s_5 \rightarrow s_6$)

What are the tradeoffs?

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

Scope of Register Allocation

Expression

Local

Loop

Global

Interprocedural

Local Register Allocation for Loops

Idea

- Estimate the benefit of allocating variables in basic blocks or loops
- Allocate variables with greatest benefit to registers
- Estimates are a function of execution frequency (from profiles, heuristics)

Surprisingly effective!

- IBM 360/370 Fortran H compiler (1968)

Local Register Allocation for Loops (cont)

Definitions

- *ldcost*: Cost (time) of load instruction
- *stcost*: Cost of store instruction
- *mvcost*: Cost of register-to-register transfer instruction
- *usesave*: Savings (time) for each use of variable in a register vs. memory
- *defsave*: Savings for each assignment of variable in a register vs. memory
- Static counts for variable v: u_i, d_i, l_i, s_i (l_i and s_i are 0 or 1)

Benefit of allocating variable v to a register in block b_i is

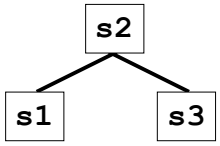
$$netsave(v,i) = u_i \cdot usesave + d_i \cdot defsave - l_i \cdot ldcost - s_i \cdot stcost$$

$$benefit(v,L) = 10^{depth(L)} \sum_{i \in blocks(L)} netsave(v,i)$$

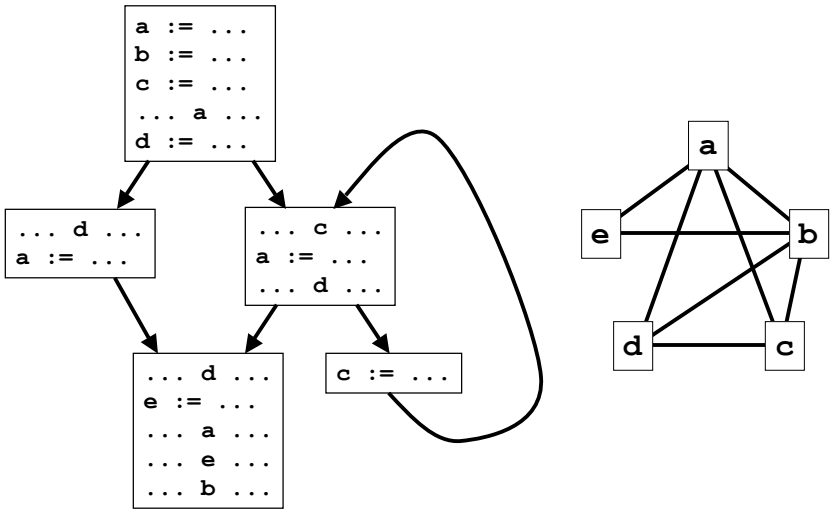
Global Register Allocation by Graph Coloring

Idea

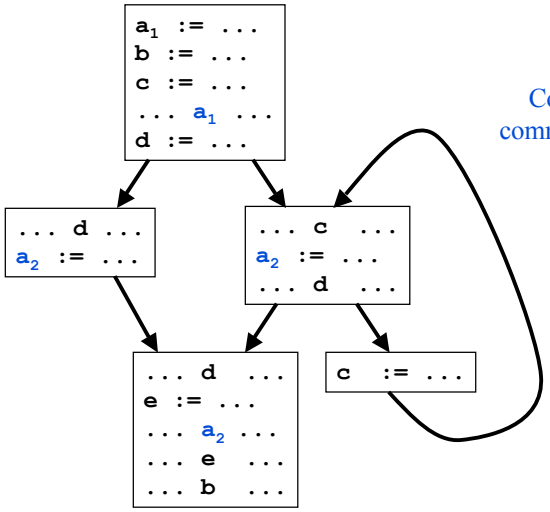
1. Construct **interference graph** *G=(N,E)*
   - Represents notion of "simultaneously live"
   - Nodes are units of allocation (*e.g.,* variables, live ranges, webs)
   - ∃ edge $(n_1,n_2) \in E$ if n_1 and n_2 are simultaneously live
   - Symmetric (not reflexive nor transitive)
2. Find ***k*-coloring** 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



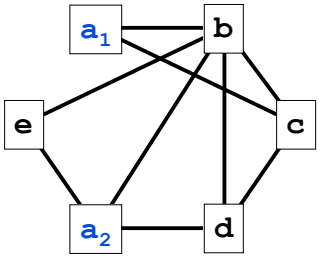
Interference Graph Example (Variables)



Interference Graph Example (Webs)



Consider webs (du-chains w/ common uses) instead of variables



Computing the Interference Graph

Use results of live variable analysis

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

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) to stack 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

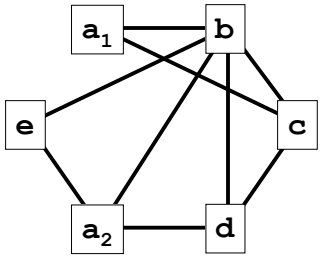
```
for each n ∈ N do                    { select n in increasing order of weight }
    if n can be colored then
        do it                                  { reserve a register for n }
    else
        Remove n (and its edges) from graph    { allocate n to stack (spill) }
```

Note

- Reserve 2-3 temp registers for manipulating data on stack

Example

Attempt to 3-color this graph (, ,)



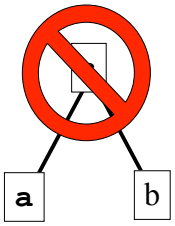
Weighted order:
a_1
b
c
d
a_2
e

What if you use a different weighting?

Problems with this approach?

Example

Attempt to 2-color this graph (,)



Weighted order:
a
b
c

Improvement #1: Simplification Phase

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 [Chaitin82]

while interference graph not empty **do**
  while ∃ a node *n* with < *k* neighbors **do**
    Remove *n* from the graph
    Push *n* on a stack
  } simplify
  if any nodes remain in the graph **then** { *blocked* with >= k edges }
    Pick a node *n* to spill { lowest spill-cost or }
    Add *n* to spill set { highest degree }
    Remove *n* from the graph
  } spill
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
  Allocate *n* to a register
} color

More on Spilling

Chaitin's algorithm restarts the whole process on spill

- Necessary, because spill code (loads/stores) uses registers
- Okay, because restarts usually only happen a couple times

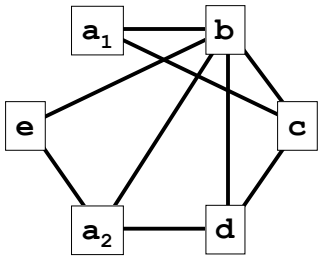
Alternative

- 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

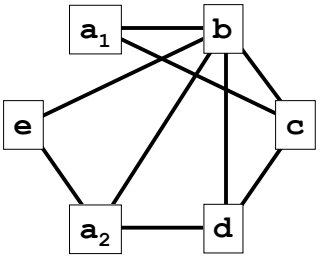
How do we order the nodes here?

Example

Attempt to 2-color this graph (,)

Spill Set:
e
a_1
a_2
b

Stack:
d
c

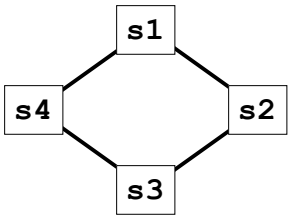


Weighted order:
e
a_1
a_2
b
c
d

Many nodes remain uncolored even though we could clearly do better

The Problem: Worst Case Assumptions

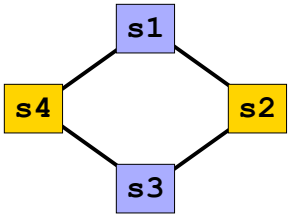
Is the following graph 2-colorable?



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



Idea

- Some neighbors might get the same color
- So 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]

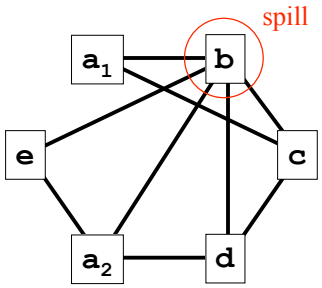
while interference graph not empty **do**
  while ∃ a node *n* with < *k* neighbors **do**
    Remove *n* from the graph
    Push *n* on a stack
  } simplify
  if any nodes remain in the graph **then** { *blocked* with >= k edges }
    Pick a node *n* to spill { lowest spill-cost/highest degree }
    Push *n* on stack
    Remove *n* from the graph
  } defer decision
while stack not empty **do**
  Pop node *n* from stack
  if n is colorable **then**
    Allocate *n* to a register
  } make decision
  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

Idea

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

Easy case

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

Examples

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

Coalescing

Move instructions

- Code generation can produce unnecessary move instructions

```
mov t1, t2
```

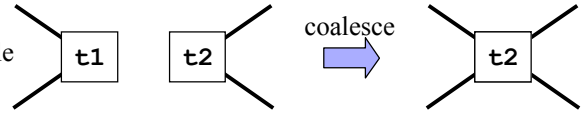
- If we can assign `t1` and `t2` to the same register, we can eliminate the move

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



Next Time

Lecture

- More register allocation
  - Allocation across procedure calls