Register Allocation

The problem:

assign machine resources (registers, stack locations)
to hold run-time data

Constraint:

simultaneously live data allocated to different locations

Goal:

minimize overhead of stack loads & stores
and register moves

Interference graph

Represent notion of "simultaneously live" using **interference graph**

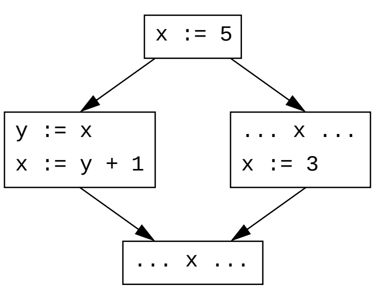
- nodes are "units of allocation"
- n_1 is linked by an edge to n_2 if n_1 and n_2 are simultaneously live at some program point
- symmetric, not reflexive, not transitive

Two adjacent nodes must be allocated to distinct locations

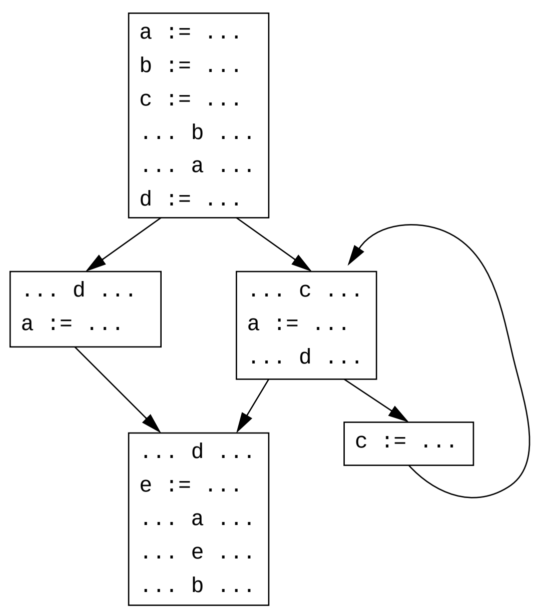
Units of allocation

What are the units of allocation?

- variables?
- separate def/use chains (**live ranges**)?
- values?
  - i.e., variables, in SSA form after copy propagation



A bigger example



Computing interference graph

Construct as side-effect of live variables analysis

- backwards iterative dfa algorithm

Flow function: identify defs & last uses

$LV_{\texttt{x := ...y...}}$:

$LV_{\texttt{if ...}}$:

Allocating registers using interference graph

Allocating variables to *k* registers is equivalent to finding a *k*-coloring of the interference graph

k-coloring: color nodes of graph using up to *k* colors, adjacent nodes have different colors

- optimal graph coloring: NP-complete

Spilling

If can't find *k*-coloring of interference graph, must **spill** some variables to stack, until the resulting interference graph is *k*-colorable

Which to spill?

- least frequently accessed variables
- most conflicting variables (nodes with highest out-degree)

Weighted interference graph:

weight(*n*) =
sum over all references (uses and defs) *r* of *n*:
execution frequency of *r*

Try to spill nodes with lowest weight and highest out-degree, if forced to spill

Static frequency estimates

Initial node: weight = 1

Nodes after branch: 1/2 weight of branch

Nodes in loop: 10x nodes outside loop

Dynamic profiles could give better frequency estimates

Just need heuristic ranking of variables

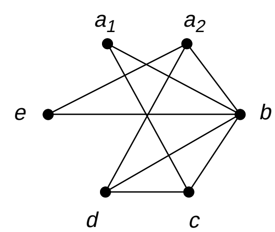
Simple greedy allocation algorithm

For all nodes, in decreasing order of weight:

- try to allocate node to a register, if possible
- if not, allocate to a stack location

Reserve 2-3 scratch registers to use when manipulating nodes allocated to stack locations

Example



Weight Order:

c
d
a_2
b
a_1
e

Assume 3 registers available

Improvement #1: add simplification phase

[Chaitin 82]

Key idea:

nodes with < *k* neighbors can be allocated after all their neighbors, but still guaranteed a register

So remove them from the graph first

- reduces the degree of the remaining nodes

Must resort to spilling only when all remaining nodes have degree $\geq k$

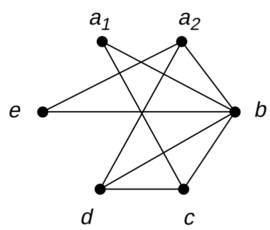
The algorithm

while interference graph not empty:
  while there exists a node with < *k* neighbors:
    remove it from the graph
    push it on a stack
  if all remaining nodes have *k* neighbors, then **blocked**:
    pick a node to spill
      (choose node with lowest (spill cost/degree))
    remove node from graph
    add to spill set

if any nodes in spill set:
  insert spill code for all spilled nodes
    (insert stores after defs, loads before uses)
  reconstruct interference graph, start over

while stack not empty:
  pop node from stack
  allocate to register

Example

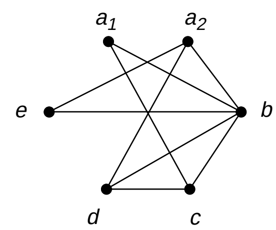


Weight Order:

c
d
a_2
b
a_1
e

Assume 3 registers available

Example



Weight Order:

c
d
a_2
b
a_1
e

Assume **2** registers available

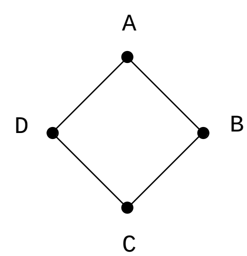
"Subsumption"

Twist in Chaitin's algorithm:
if see `x := y`, where x & y not simultaneously live,
then merge live ranges & eliminate all such copies

+ avoids generating code for simple copies
– can introduce extra spilling

If allocate values instead of variables or live ranges,
then subsumption happens implicitly

An annoying case



If only 2 registers available $\Rightarrow$ blocked immediately, must spill

Improvement #2: blocked doesn't mean spill

[Briggs *et al.* 89]

Key idea:
just because a node has *k* neighbors
doesn't mean it will need to be spilled
(neighbors may get overlapping colors)

Algorithm:

Like Chaitin, except:

- when removing blocked node, just push onto stack ("optimistic spilling")
- when done removing nodes:
  - pop nodes off stack and see if they can be allocated
  - really spill only if it can't be allocated at this stage

Other miscellaneous enhancements

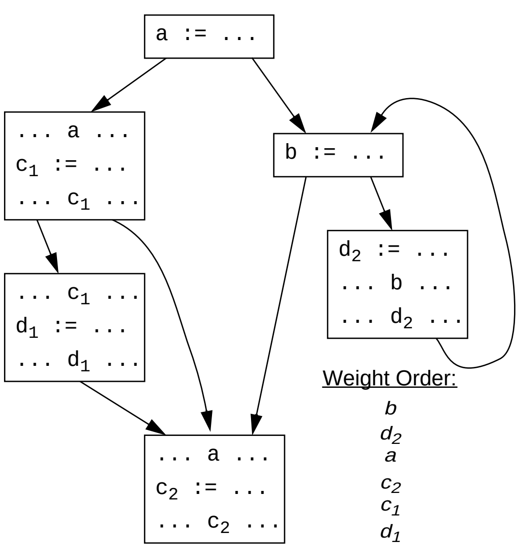
Improvement #3: live range splitting

Priority-Based Coloring [Chow & Hennessy 84]

Key idea: if a variable can't be allocated to a register,
try to split it into multiple subranges that can be allocated separately

- move instructions inserted at split points
- some live range pieces in registers, some in memory
  ⇒ selective spilling

Example



Assume **2** registers available

Improvement #4: rematerialization

Idea: instead of *reloading* value from memory,
recompute it instead,
if recomputation is cheaper than reloading

Simple strategy: choose rematerialization over spilling, if

- can recompute a value in a single instruction, and
- all operands will always be available

Examples:

- constants
- address of global var
- address of var in stack frame

Performance results

[Briggs *et al.* 94]

E.g.

For some procedure:

XXX spill instructions before
YYY spill instructions after

YYY is Z% smaller than XXX

- Z ranges between -2% and 48% for “optimistic spilling”
- Z ranges between -26% and 33% for rematerialization

Optimistic spilling a good heuristic
Mixed results for rematerialization

Register allocation and calls

Simple approach: calling conventions

More sophisticated: interprocedural register allocation

Calling conventions

Goals:

- fast calls
  - pass *k* arguments in registers, result in register
- language-independent
- support debugger, profiler, etc.

Problematic language features:

- varargs
- passing/returning aggregates
- returning multiple values
- exceptions, `setjmp`/`longjmp`

Callee-save vs. caller-save registers

Need a convention at calls for which registers managed by caller (**caller-save**) and which managed by callee (**callee-save**)

- SPARC has **hardware-save** registers, too

Caller-save:

- caller must save/restore any caller-save registers live across calls
- callee is free to use these registers w/o any overhead

Callee-save:

- callee must save/restore any callee-save registers it uses
- caller is free to use these registers, even across calls

Hardware-save:

- caller and callee can use freely

A problem with callee-save registers

Run-time utilities (e.g. `longjmp`) and
programming environment tools (e.g. debugger)
need to be able to find contents of registers relative to a particular stack frame

Caller-save registers are on stack in stack frame at known place

Callee-save registers?

Impact on register allocator

How should register allocator deal w/ calling conventions?

Simple: calling-convention-oblivious register allocation

- spill all live caller-save registers before call, restore after call
- save all callee-save registers at entry, restore at return

Better: calling-convention-aware register allocation

- incorporate preferred registers for formals, actuals
- call kills caller-save registers
  - allocator knows to avoid these registers, save/restore code turns into normal spills
  - live-range splitting particularly useful to split var into before call/during call/after call segments
- entry is def of all callee-save registers, exit is use
  - allocator knows must spill these registers if used in proc

Exploiting calling convention

Calling-convention-aware register allocator
can customize its usage to use “cheaper” registers

- leaf routines (try to) use only caller-save registers
- routines with calls use callee-save registers for variables live across calls

Poor man’s interprocedural register allocation

Rich man’s interprocedural register allocation

Allocate registers across calls to minimize overlap between caller and callee subgraph

Allocate global variables to registers over entire program

Could do compile-time interprocedural register allocation

+ gains most benefit
– might be expensive
– might require lots of recompilation after programming change

Or, could do link-time re-allocation

+ low compile-time cost
+ little impact on separate compilation
– cost at link time
– probably less effective

Wall's link-time register allocator

[Wall 86]

Compiler does local allocation + planning for linker

- generates call graph info
- generates variable usage info for each proc
- generates **register actions**
  executed by linker if variable allocated to register

Linker does interprocedural allocation & patches compiled code

- determines interference graph among variables
- picks best additional variables to allocate to registers
- executes register actions for those vars to patch compiled code

Register actions

Describe changes to code if given var allocated to a register

OPx(`var`): replace operand x with reg allocated to `var`
RESULT(`var`): replace result with reg allocated to `var`
REMOVE(`var`): delete instruction if `var` allocated to a reg

Use: for each variable *`var`*

- `r := load` *`var`*: REMOVE(*`var`*)
- `rk := ri op rj:`
  OP1(*`var`*) if *`var`* loaded into ri,
  OP2(*`var`*) if *`var`* loaded into rj,
  RESULT(*`var`*) if *`var`* stored from rk,
- `store` *`var`* `:= r`: REMOVE(*`var`*)

Example

Source code:

```
w = (x + y) * z;
```

original code	register actions			
	x	y	z	w
`r1 := load x`	REMOVE			
`r2 := load y`		REMOVE		
`r3 := r1 + r2`	OP1	OP2		
`r4 := load z`			REMOVE	
`r5 := r3 * r4`			OP2	RESULT
`store w := r5`				REMOVE

A problem

What if loaded value is still live after an overwriting store?

Example: `w = y++ * z;`

original code	register actions		
	y	z	w
`r1 := load y`	REMOVE		
`r2 := r1 + 1`	OP1, RESULT		
`store y := r2`	REMOVE		
`r2 := load z`		REMOVE	
`r1 := r1 * r2`	OP1	OP2	RESULT
`store w := r1`			REMOVE

These register actions are broken, if y in a register!

```
ry := ry + 1
r2 := load z
r1 := ry * r2   // ry reads updated y value, not original
store w := r1
```

Solution

Need two more actions:
LOAD(`var`): replace load with move from reg holding `var`
STORE(`var`): replace store with move to reg holding `var`

Use LOAD(`var`) instead of REMOVE(`var`) if
`var` is stored into while result of load is still live
Use STORE(`var`) instead of REMOVE(`var`) if
rhs is stored into more than one variable

Example: `w = x = y++ * z;`

original code	register actions			
	x	**y**	**z**	**w**
`r1 := load y`		LOAD		
`r2 := r1 + 1`		RESULT		
`store y := r2`		REMOVE		
`r2 := load z`			REMOVE	
`r1 := r1 * r2`			OP2	RESULT
`store x := r1`	STORE			OP1
`store w := r1`				REMOVE

Link-time operations

Construct weighted call graph from compiler tables
- weights can come from static estimates or profile info
- each proc annotated with list of used local vars

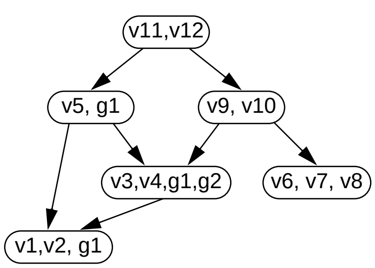
Traverse call graph bottom-up, assigning locals to **groups**
(a kind of interference graph)
- no simultaneously-live locals in same group
- each global in its own group
- group weighted by sum of members' weights
- recursion & indirect calls pose complications

Allocate groups to registers in decreasing order of weight

Run register actions during code relocation to improve code

Example

Call graph:



Groups:

Possible improvements

Use real profile data to construct weights

Do intraprocedural register allocation at compile-time

Track liveness info for vars at each call site
Track intraprocedural interference graph

Use real interference graph to run link-time allocation

Results

DECWRL Titan RISC processor: 64 registers

Basic experiment:

- local compile-time allocation uses 8 registers
- interprocedural link-time allocator uses 52 registers
- simple static frequency estimates
- smallish benchmark programs

⇒ 10-25% speed-up over local allocation alone

Small improvements (0-6%) with real profile data

Small improvements (0-5%) if use intraprocedural allocation too

- more pronounced for larger, real benchmarks

Less benefit if fewer registers available for global allocation
e.g. 5-20% for 8 global registers

Link-time + local better than intraprocedural register allocation