Instruction Scheduling

Reorder instructions to better fit target machine's pipeline

- fill control transfer delay slots
- avoid using result of multi-cycle operations too early
  - loads, floating point operations, ...
- schedule code for VLIW, superscalar machines
  - coordinate multiple instructions to fit available machine resources

Techniques:

- **list scheduling**, in a basic block
- **trace scheduling** across conditional branches
- **software pipelining**, across loop iterations

Loop unrolling often can help scheduling

Register allocation can hurt scheduling

List scheduling

[Gibbons & Muchnick 86]

Schedule a basic block...

- obeying data dependencies
- avoiding interlocks

Previous work: exponential, $O(n^4)$ algorithms

This work: $O(n^2)$ algorithm, simple

Pipeline model

Hazards considered:

- load followed by use of target of load
- store followed by a load
- load followed by
  ALU op or load/store with address calculation

```
r2 := r1 + 1
sp := sp - 12
*A := r0
r3 := *(sp+4)
r4 := *(sp+8)
sp := sp - 8
*sp := r2
r5 := *A
r4 := r0 + 1
```


Step 1: construct data dependence graph

Convert linear basic block into a DAG representing data dependencies

Loads & stores assumed to alias,
except that different offsets from common base reg (e.g. `sp`) do not alias

```
① r2 := r1 + 1
② sp := sp - 12
③ *A := r0
④ r3 := *(sp+4)
⑤ r4 := *(sp+8)
⑥ sp := sp - 8
⑦ *sp := r2
⑧ r5 := *A
⑨ r4 := r0 + 1
```

Step 2: traverse dependency graph, emitting code

Maintain set of **candidate nodes**
whose data dependence predecessors have been emitted

candidates := roots of DAG
while |candidates| > 0 do
select **best** available candidate node
emit it
remove it from the DAG
add any new root nodes to candidate set

Best node:

1. doesn't interlock with previous instruction, or
2. does interlock with an immediate successor node, or
3. has the most immediate successor nodes, or
4. is along the longest path to the leaves of the DAG

[Previous work used lookahead in DAG to guide choice
⇒ complex and slow (worst case)]

Results

["Effectiveness of a Machine-Level Global Optimizer",
Johnson & Miller, PLDI '86]

Compiling small benchmark programs: 7% improvement

Register Allocation

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

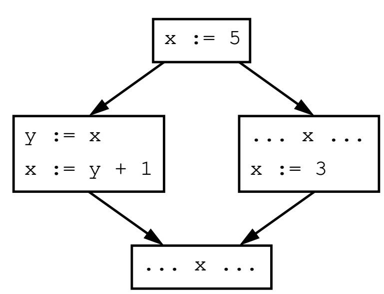
Constraint:
simultaneously live data allocated to different locations

Goal:
minimize overhead of stack loads & stores
and register moves

Granularity of allocation

What is allocated to registers/stack locations?

- variables?
- values?
  - same as variables after SSA form + copy propagation
- separate def/use chains (**live ranges**)?



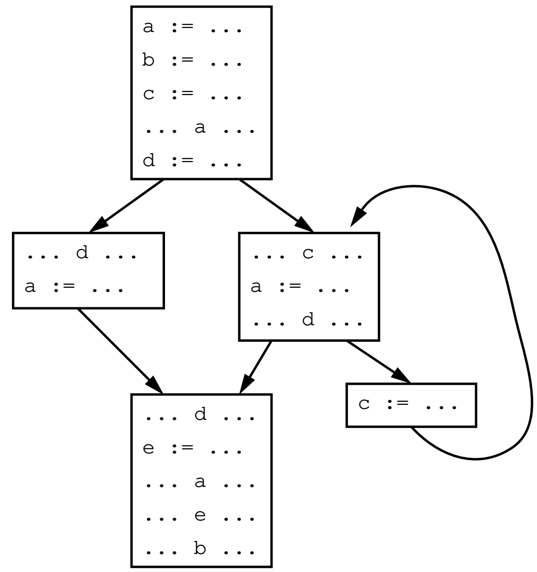
Interference graph

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

Example



Computing interference graph

Construct as side-effect of live variables analysis

- backwards iterative dfa algorithm

Flow function: identify defs & last uses

E.g.

```
a := ... b ...
```

Merge function:

E.g.

liveVars1 **merge** liveVars2

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

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, 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

Just need heuristic ranking of variables

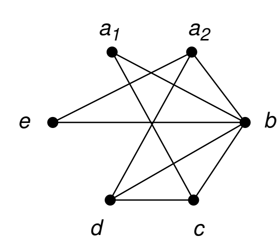
Simple greedy allocation algorithm

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 temp 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, and 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 $\geq k$

Chaitin allocates live ranges to registers

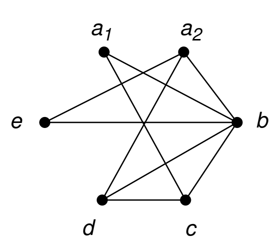
The algorithm

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

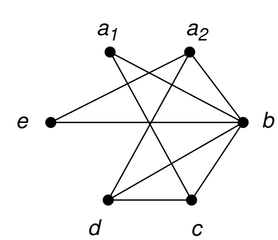


<u>Weight Order:</u>

c
d
a_2
b
a_1
e

Assume 3 registers available

Example



<u>Weight Order:</u>

c
d
a_2
b
a_1
e

Assume **2** registers available

Spilling

Chaitin restarts algorithm if anything spilled

- loads & stores of spilled value need registers for short time

Alternative:
reserve a couple temp registers for use by spill code

+ no need to start over if anything spilled
– lose 2-3 registers that might help avoid spilling

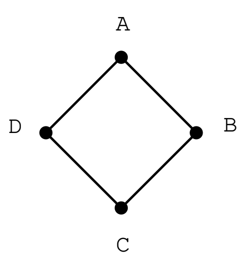
Subsumption

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

- can introduce extra spilling, though

If allocate values instead of variables,
then subsumption happens automatically

An annoying case



If only 2 registers available ⇒ 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

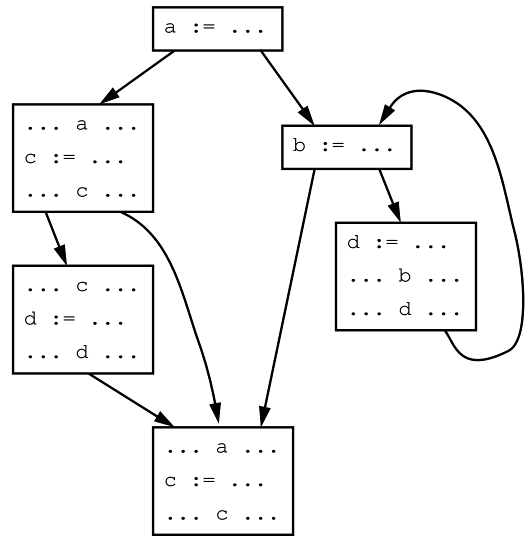
Improvement #3: live range splitting

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

- some live range pieces in registers, some in memory
  ⇒ selective spilling
- insert load, store, register move instructions at boundaries

Example



Improvement #4: rematerialization

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

Easy case:

- can recompute a value in a single instruction (or so)
  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

What's the conclusion?

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
- `setjmp`/`longjmp`

Other issues:
- exceptions, non-local returns, etc.
- non-LIFO activation records

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

Callee-save:
- callee must save/restore any callee-save registers it uses
- caller is free to use these registers

Hardware-save:
- caller and callee can use freely

Impact on register allocator

How should register allocator deal w/ calling conventions?

Simple approach:
- spill all live caller-save registers before call, restore after call
- save all callee-save registers at entry, restore at return
- register allocation insensitive to calls

Better:
- 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

Problems 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?

Exploiting calling convention

Register allocator can customize its usage to use "cheaper" registers

- leaf routines 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 action(s)
`r1 := load x`	REMOVE(`x`)
`r2 := load y`	REMOVE(`y`)
`r3 := r1 + r2`	OP1(`x`), OP2(`y`)
`r4 := load z`	REMOVE(`z`)
`r5 := r3 * r4`	OP2(`z`), RESULT(`w`)
`store w := r5`	REMOVE(`w`)

A problem

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

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

original code	register action(s)
`r1 := load y`	REMOVE(`y`)
`r2 := r1 + 1`	OP1(`y`), RESULT(`y`)
`store y := r2`	REMOVE(`y`)
`r2 := load z`	REMOVE(`z`)
`r1 := r1 + r2`	OP1(`y`), OP2(`z`), RESULT(`w`)
`store w := r1`	REMOVE(`w`)

Extensions

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 action(s)
`r1 := load y`	LOAD(y)
`r2 := r1 + 1`	RESULT(y)
`store y := r2`	REMOVE(y)
`r2 := load z`	REMOVE(z)
`r1 := r1 * r2`	OP2(z), RESULT(w)
`store x := r1`	STORE(x), OP1(w)
`store w := r1`	REMOVE(w)

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)

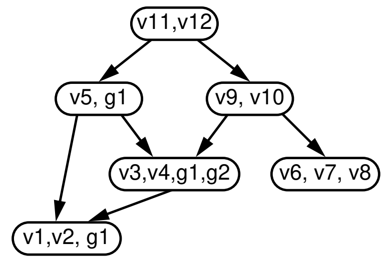
- no simultaneously-live locals in same group
- each global in its own group
- group weighted by sum of members' weights
- recursion & indirect calls pose complications

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

$\Rightarrow$ 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