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

OP`x`(`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

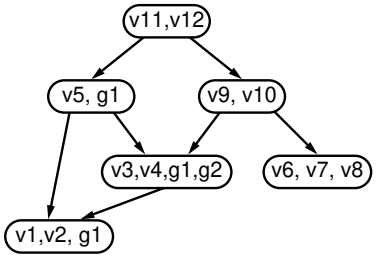
Traverse call graph bottom-up, assigning locals to **groups** (a kind of interference graph)
- no simultaneously-live locals in same group
- each global in its own group
- group weighted by sum of members' weights
- recursion & indirect calls pose complications

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