

More Register Allocation

Last time

- Register allocation
 - Global allocation via graph coloring

Today

- More register allocation
 - Procedure calls
 - Interprocedural

Next time?

Register Allocation and Procedure Calls

Problem

- Register values may change across procedure calls
- The allocator must be sensitive to this

Two approaches

- Work within a well-defined calling convention } Make “local” decisions
- Use interprocedural allocation } Make “global” decisions

Calling Conventions

Goals

- Fast calls (pass arguments in registers, minimal register saving/restoring)
- Language-independent
- Support debugging, profiling, *etc.*

Complicating Issues

- Varargs
- Passing/returning aggregates
- Exceptions, non-local returns
 - `setjmp()`/`longjmp()`
- Non-LIFO activation records

Architecture Review: Caller- and Callee-Saved Registers

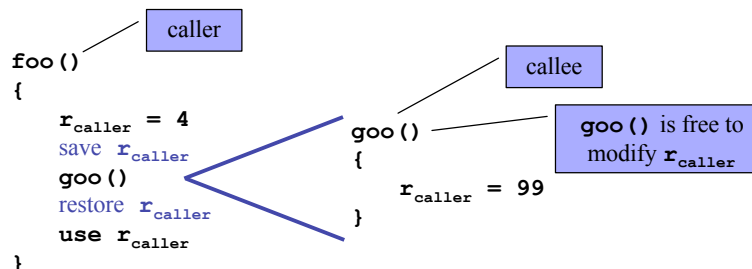
Partition registers into two categories

- Caller-saved
- Callee-saved

Caller-saved registers

- Caller must save/restore these registers when live across call
- Callee is free to use them

Example

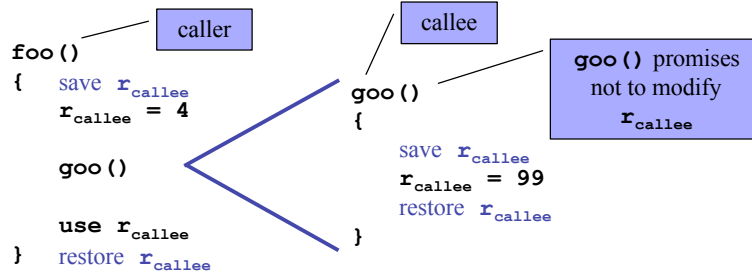


Architecture Review: Caller- and Callee-Saved Registers

Callee-saved registers

- Callee must save/restore these registers when it uses them
- Caller expects callee to not change them

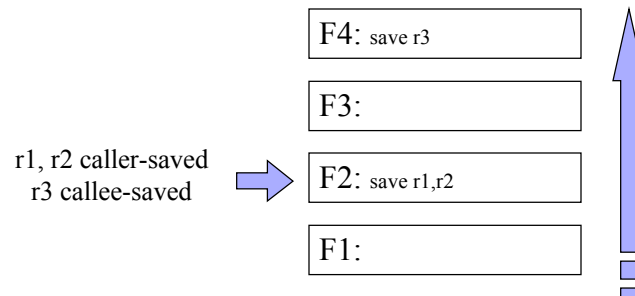
Example



Aside: Problem with Callee-Saved Registers

Run-time systems (e.g., `setjmp()`/`longjmp()`) and debuggers) need to know register values in any stack frame

- Caller-saved registers are on stack frame at known location
- Callee-saved registers?



Register Allocation and Calling Conventions

Insensitive register allocation

- Save all live caller-saved registers before call; restore after
- Save all used callee-saved registers at procedure entry; restore at return
- Suboptimal

```
foo ()  
{  
    t = ...  
    ... = t  
    s = ...  
    f ()  
    g ()  
    ... = s  
}
```

A variable that is not live across calls should go in caller-saved registers

A variable that is live across multiple calls should go in callee-saved registers

Sensitive register allocation

- Encode calling convention constraints in the IR and interference graph
- How? Use precolored nodes

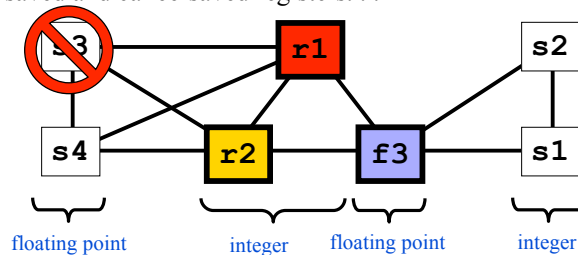
Precolored Nodes

Add architectural registers to interference graph

- Precolored (mutually interfering)
- Not simplifiable (infinite degree)
- Not spillable

Express allocation constraints

- Integers usually can't be stored in floating point registers
- Some instructions can only store result in certain registers
- Caller-saved and callee-saved registers. . .



Precolored Nodes and Calling Conventions

Callee-saved registers

- Treat entry as def of all callee-saved registers
- Treat exit as use of them all
- Allocator must “spill” callee-saved registers to use them

Encourage use of callee-saved regs

```
foo ()  
{  
  def (r3)  
  use (r3)  
}
```

Live range of callee-saved registers

Encourage use of callee-saved regs

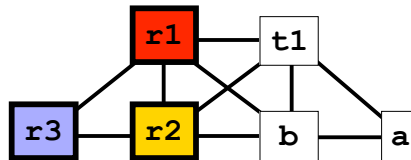
Caller-saved registers

- Variables live across call interfere with all caller-saved registers
- Splitting can be used (before/during/after call segments)

Example

```
foo () :  
  def (r3)  
  t1 := r3  
  a := ...  
  b := ...  
  ... a ...  
  call goo  
  ... b ...  
  r3 := t1  
  use (r3)  
  return
```

r1, r2 caller-saved
r3 callee-saved



Summary

Callee/caller-saved register allocation

- Can be effective
- Still making “local” decisions (may save registers when we don’t need to)

Let’s broaden the scope. . .

Interprocedural Register Allocation

Wouldn’t it be nice to. . .

- Allocate registers across calls to minimize unnecessary saves/restores?
- Allocate global variables to registers over entire program?

Compile-time interprocedural register allocation?

- + Could have great performance
- Might require lots of recompilation after changes (no separate compilation?)
- Might be expensive

Link-time interprocedural re-allocation?

- + Low compile-time cost
- + Little impact on separate compilation
- Link-time cost

Wall's Link-time Register Allocator [Wall 86]

Overall strategy

- Compiler uses 8 registers for local register allocation
- Linker controls allocation of remaining 52 registers

Compiler does local allocation & *planning* for linker

- Load all values at beginning of each basic block;
store all values at end of each basic block
- Generate call graph information
- Generate variable usage information for each procedure
- Generate **register actions**

Linker does interprocedural allocation & patches compiled code

- Generates "interference graph" among variables
- Picks best variables to allocate to registers
- Executes register actions for allocated variables to patch code

Register Actions

Describe code patch if particular variable allocated to a register

- **REMOVE(var)**: Delete instruction if **var** allocated to a register
- **OPx(var)**: Replace op x with register that was allocated to **var**
- **RESULT(var)**: Replace result with register allocated to **var**

Usage

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

Example

w := (x + y) * z

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)

Another Example

w := y++ * z

Suppose **y** is allocated to register **r5**

	REMOVE(y)
r5 := r5 + 1	OP1(y), RESULT(y)
	REMOVE(y)
r2 := load z	REMOVE(z)
r1 := r5 * r2	OP1(y), OP2(z), RESULT(w)
store w := r1	REMOVE(w)

Problem

- Loaded value is still live after store overwrites it
- Post-incremented value of **y** is lost if **y** is allocated to register
- We need two registers to hold the two values of **y**

Extension

More actions

- **LOAD(var)**: Replace load with move from the register holding **var**
- **STORE(var)**: Replace store with move to the register holding **var**

LOAD(var)

- Use instead of **REMOVE(var)** if **var** is stored into while result of load is still live

STORE(var)

- Use instead of **REMOVE(var)** if source is stored into more than one variable

Example Revisited

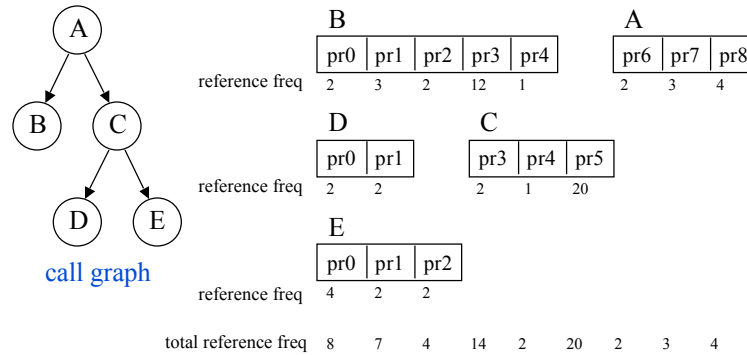
w := x := y++ * z

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

Deciding Which Variables to Promote to Registers

Steps

- Use bottom-up algorithm to assign pseudo registers
- Allocate pseudo registers to non-simultaneously live variables
- Allocate real registers to most frequently used pseudo registers



Possible Improvements

- Use profile data to construct weights**
- Do global register allocation at compile-time**
- Track liveness information for variables at each call site**
- Track intraprocedural interference graph**
- Use real interference graph at link-time**

Performance Summary

Machine: DEC WRL Titan RISC processor (64 registers)

Basic experiment

- Local compile-time allocator uses 8 registers
 - Link-time allocator uses 52 registers
 - Simple static frequency estimates
 - Small benchmarks
- ⇒10-25% speed-up over local allocation alone

Improvements

- 0-6% with profile data
- 0-5% with compile-time global allocation

Benefit decreases with number of link-time registers

Link-time better than global register allocation

Link-Time Register Allocation: The Big Picture

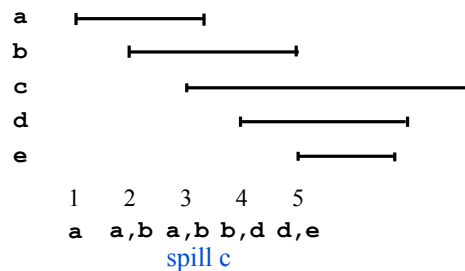
Delayed decision making

- Make decisions when more information is available, *e.g.*, link time
- Requires communication among different system components, in this case the compiler and the linker
- Leads to staged compilation
- Intuitively, more information is better, but effectively using this information can require cleverness

JIT Environment

Dynamic compilation requires fast register allocation

- Linear Scan Register Allocation [Poletto & Sarkar99]
- Not based on graph coloring
- Greedy algorithm based on live intervals
 - Spill the variable whose interval ends furthest in the future



- What if we had spilled a or b instead of c?

Linear Scan Register Allocation

Performance results

- Linear scan is linear in number of variables
- Graph coloring is $O(n^2)$
- Code quality is within 12% of graph coloring

Concepts

Register allocation and procedure calls

Calling conventions

- Caller- vs. callee-saved registers
- Precoloring
- Finding register values in stack can be hard

Interprocedural analysis

- Link-time register allocation
 - Register actions

Register allocation in a JIT

- Linear Scan Register Allocation

Course Summary

Foundations

- Control/data-flow analysis (lattice-theoretic underpinnings)
- Program representations (SSA)
- Reuse optimization (CSE, PRE, LICM)
- Alias analysis
- Reading: SSA

Interprocedural analysis

- Context (in)sensitive, flow (in)sensitive pointer analysis

Modern topics

- Program slicing
- OO languages (optimization: data/code reorganization, field analysis)
- Readings: dynamic translation, adaptive optimization

Traditional topics

- Register allocation (global, interprocedural, and fast)

Next Time

Final?

– No!

Enjoy break!