Low-Level Issues

Last lecture
– Liveness analysis
– Register allocation

Today
– More register allocation

Later
– Instruction scheduling

Register Allocation

Problem
– Assign an unbounded number of symbolic registers to a fixed number of architectural registers
– Simultaneously live data must be assigned to different architectural registers

Goal
– Minimize overhead of accessing data
  – Memory operations (loads & stores)
  – Register moves
**Improvement #1: Simplification Phase [Chaitin 81]**

**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 $\geq k$**

**Referred to as pessimistic spilling**

---

**Simplifying Graph Allocators**

**Chaitin**
- renumber
  - build
    - coalesce
      - spill costs
        - simplify
          - select

**Briggs**
- renumber
  - build
    - coalesce
      - spill costs
        - simplify
          - select

spill code
**Algorithm** [Chaitin81]

while interference graph not empty do
  while ∃ a node \( n \) with < \( k \) neighbors do
    \{ simplify \}
    Remove \( n \) from the graph
    Push \( n \) on a stack
  if any nodes remain in the graph then \{ blocked with \( \geq k \) edges \}
    Pick a node \( n \) to spill \{ lowest spill-cost or \}
    Add \( n \) to spill set \{ highest degree \}
    Remove \( n \) from the graph
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 or select \}

---

**Example**

Attempt to 3-color this graph ( , , )

Stack:
- d
dc
cb
bfa
afa
ea

Possible order:
- e
ead
abe
acf
cbd

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 [Briggs 89]

Idea
- Some neighbors might get the same color
- 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
        if any nodes remain in the graph then
            { blocked with $\geq k$ edges }
            Pick a node $n$ to spill
            { lowest spill-cost/highest degree }
            Push $n$ on stack
            Remove $n$ from the graph

    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 }
            } defer decision
        Reconstruct interference graph & start over

---

**Example**

Attempt to 2-color this graph (  ,  )

Stack:  

    d  
    c  
    b* 
    f* 
    a* 
    e* 

* blocked node

Increasing Weight:  

    e  
    a  
    f  
    b  
    c  
    d  

---
**Improvement #3: Coalescing**

**Move instructions**
- Code generation can produce unnecessary move instructions
  ```
  mov t1, t2
  ```
- If we can assign \( t_1 \) and \( t_2 \) to the same register, we can eliminate the move

**Idea**
- If \( t_1 \) and \( t_2 \) 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

---

**Coalescing Logistics**

**Rule**
- When building the interference graph, do NOT make virtual registers interfere due to copies.
- If the virtual registers \( s_1 \) and \( s_2 \) do not interfere and there is a copy statement \( s_1 = s_2 \) then \( s_1 \) and \( s_2 \) can be coalesced.
- Example

```
Before Coalescing
a = t + u
... 
b = a
c = a
...
x = b + w
z = c + y

After Coalescing
ab = t + u
... 
c = ab
...
x = ab + w
z = c + y
```
**Computing the Interference Graph (in MiniJava compiler)**

Computing interference graph to enable coalescing

Use results of live variable analysis

```plaintext
for each flow graph node n do
  for each def in def(n) do
    for each temp in liveout(n) do
      if (not stmt(n) isa MOVE or use != temp) then
        E ← E U (def, temp)
```

**Register Allocation: Spilling**

If we can’t find a k-coloring of the interference graph
- Spill variables (nodes) 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!
More on Spilling

Chaitin’s algorithm restarts the whole process on spill
– Necessary, because spill code (loads/stores) uses registers
– Okay, because it usually only happens a couple times

Alternative
– Reserve 2-3 registers for spilling
– Don’t need to start over
– But have fewer registers to work with

Weighted Interference Graph

Goal
– Weight($s$) = $\sum_{\forall r \text{ references } s} f(r)$ $f(r)$ is execution frequency of $r$

Static approximation
– Use some reasonable scheme to rank variables
– Some possibilities
  – Weight($t$) = num of times $t$ is used in program
  – Weight($t$) = 10 $\times$ (# uses in loops) + (# uses in straightline code)
  – Weight($t$) = 20 $\times$ (# uses in loops) + 2 $\times$ (# uses in straightline code) + (# uses in a branch statement)
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
– Use interprocedural allocation (not covering this)

Calling Conventions

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

Complicating Issues
– Varargs
– Passing/returning aggregates
– Exceptions, non-local returns
  – setjmp() / longjmp()
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**

```c
foo()
{
    rcaller = 4
    save rcaller
    goo()
    restore rcaller
}
```

goo()

```c
rcaller = 99
```

goo() is free to modify `rcaller`

**Callee-saved registers**

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

**Example**

```c
foo()
{
    rcallee = 4
    goo()
}
```

goo()

```c
rcallee = 99
```

goo() promises not to modify `rcallee`
Architectures with Callee and Caller Registers

**SPARC**  
- hardware-saved %i0-%i7, %o0-%o8

**Alpha**  
- 7 callee-saved out of 32 registers

**MIPS**  
- caller-saved: $t0-$t9, $a0-$a3, $v0-$v1  
- callee-saved: $s0-$s7, $ra, $fp

**PPC**  
- 18 callee-saved  
- 14 caller-saved

**StarCore EABI**  
- 4 callee-saved  
- 28 caller-saved

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

```plaintext
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
- 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

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

Caller-saved registers
- Variables live across call interfere with all caller-saved registers
Example

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

r1, r2 callee-saved
r3 callee-saved

Tradeoffs

Callee-saved registers
+ Decreases code size: one procedure body may have multiple calls
+ Small procedures tend to need fewer registers than large ones; callee-save makes sense because procedure sizes are shrinking
− May increase execution time: For long-lived variables, may save and restore registers multiple times, once for each procedure, instead of a single end-to-end save/restore

The larger “problem”
− We’re making local decisions for policies that require global information
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?

|   F4: save r3   |
| F3:            |
| F2: save r1,r2 |
| F1:            |

r1, r2 caller-saved
r3 callee-saved

Concepts

Decision tree for register allocation

Global approaches centered around an interference graph
– greedy coloring
– coloring with simplification [Chaitin]
– coloring with simplification and optimistic spilling [Briggs]
– coloring with simplification, coalescing, and optimistic spilling

Register allocation across procedure calls
– precolored nodes in the interference graph
Next Time

Lecture
  – Instruction scheduling