CS412/413

Introduction to Compilers
Radu Rugina

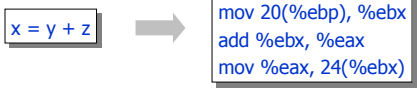
Lecture 32: Register Allocation
19 Apr 04

Variables vs. Registers/Memory

- Difference between IR and assembly code:
  - IR (and abstract assembly) manipulate data in local and temporary variables
  - Assembly code manipulates data in memory/registers
- During code generation, compiler must account for this difference
- Compiler backend must allocate variables to memory or registers in the generated assembly code

Simple Approach

- Straightforward solution:
  - Allocate each variable on stack
  - At each instruction, bring values needed into registers, perform operation, then store result to memory



- Problem: program execution very inefficient
  - moving data back and forth between memory and registers

Register Allocation

- Better approach = register allocation: keep variable values in registers as long as possible
- Best case: keep a variable's value in a register throughout the lifetime of that variable
  - In that case, we don't need to store it in memory
  - We say that the variable has been allocated in a register
  - Otherwise allocate variable on stack
  - We say that variable is spilled to memory
- Which variables can we allocate in registers?
  - Depends on the number of registers in the machine
  - Depends on how variable values are being used
- Main Idea: cannot allocate two variables to the same register if they are both live at some program point

Register Allocation Algorithm

Hence, basic algorithm for register allocation is:

1. Perform live variable analysis (over abstract assembly code!)
2. Inspect live variables at each program point
3. If two variables are in same live set, can't be allocated to the same register – they interfere with each other

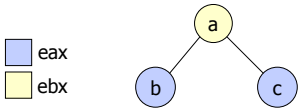
How do we determine register assignments next?

Interference Graph

- Nodes = program variables
- Edges = connect variables that interfere with each other

```
b = a + 2;      {a}
c = b*b;        {a,b}
b = c + 1;      {a,c}
return b*a;     {a,b}
```

- Register allocation = graph coloring

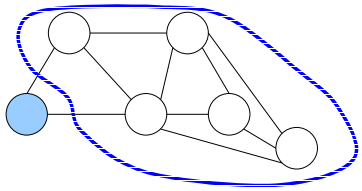


Graph Coloring

- Questions:
  - Can we efficiently find a coloring of the graph whenever possible?
  - Can we efficiently find the optimum coloring of the graph?
  - Can we assign registers to avoid move instructions?
  - What do we do when there aren't enough colors (registers) to color the graph?

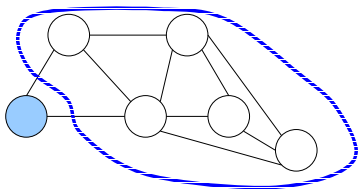
Coloring a Graph

- Assume K = number of registers (take K=3)
- Try to color graph with K colors
- Key operation = Simplify: find some node with at most K-1 edges and cut it out of the graph



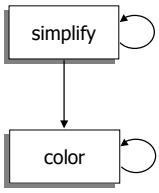
Coloring a Graph

- Idea: once coloring is found for simplified graph, removed node can be colored using free color
- Algorithm: simplify until graph contain no nodes
- unwind adding nodes back & assigning colors



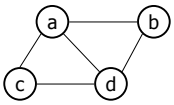
Stack Algorithm

- Phase 1: Simplification
  - Repeatedly simplify graph
  - When remove a variable (i.e., graph node), push it on a stack
- Phase 2: Coloring
  - Unwind stack and reconstruct the graph as follows:
  - Pop variable from the stack
  - Add it to the graph
  - Color the node for that variable

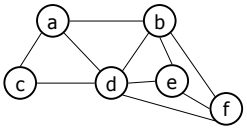


Stack Algorithm

- Example:

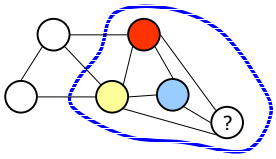


- …how about:



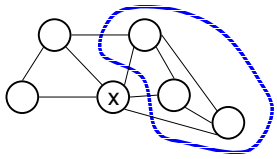
Failure of Heuristic

- If graph cannot be colored, it will reduce to a graph in which every node has at least K neighbors
- May happen even if graph is colorable in K!
- Finding K-coloring is NP-hard problem (requires search)



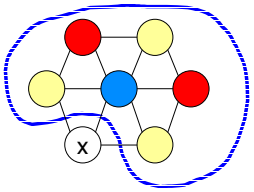
Spilling

- Once all nodes have K or more neighbors, pick a node and mark it for spilling (storage on stack).
- Remove it from graph, push it on stack
- Try to pick node not used much, not in inner loop



Optimistic Coloring

- Spilled node may be K-colorable
- Try to color it when popping the stack
- If not colorable, actual spill: assign it a stack location



Accessing Spilled Variables

- Need to generate additional instructions to get spilled variables out of stack and back in again
- Simple approach: always keep extra registers handy for shuttling data in and out
- Better approach: rewrite code introducing a new temporary, rerun liveness analysis and register allocation

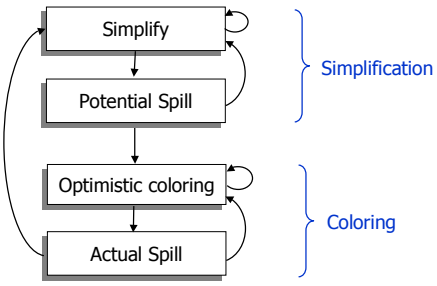
Rewriting Code

- Example: add v1, v2
- Suppose that v2 is selected for spilling and assigned to stack location [ebp-24]
- Add new variable t35 for just this instruction, rewrite:

```
mov –24(%ebp), t35
add v1, t35
```

- Advantage: t35 has short lifetime and doesn't interfere with other variables as much as v2 did.
- Now rerun algorithm; fewer or no variables will spill.

Putting Pieces Together



Precolored Nodes

- Some variables are pre-assigned to registers
- mul instruction has
  use[I] = eax, def[I] = { eax, edx }
- call instruction kills caller-save regs:
  def[I] = { eax, ecx, edx }
- To properly allocate registers, treat these register uses as special temporary variables and enter into interference graph as precolored nodes

Precolored Nodes

- Can't simplify graph by removing a pre-colored node
- Precolored nodes: starting point of coloring process
- Once simplified graph is all colored nodes, add other nodes back in and color them

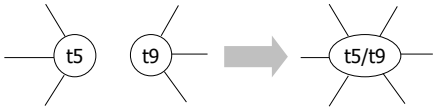
Optimizing Move Instructions

- Code generation produces a lot of extra mov instructions

  mov t5, t9
- If we can assign t5 and t9 to same register, we can get rid of the mov
- Idea: if t5 and t9 are not connected in inference graph, coalesce them into a single variable; the move will be redundant.

Coalescing

- When coalescing nodes, take union of edges
- Hence, coalescing results in high-degree nodes
- Problem: coalescing nodes can make a graph uncolorable



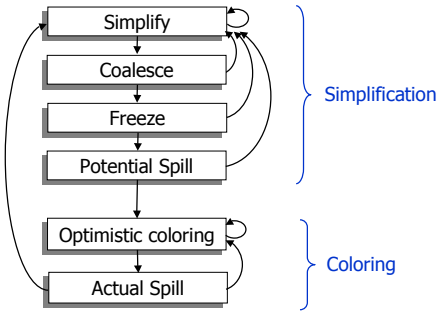
Conservative Coalescing

- Conservative = ensure that coalescing doesn't make the graph non-colorable (if it was colorable before)
- Approach 1: coalesce a and b if resulting node ab has less than K neighbors of significant degree
  - Safe because we can simplify graph by removing neighbors with insignificant degree, then remove coalesced node and get the same graph as before
- Approach 2: coalesce a and b if for every neighbor t of a: either t already interferes with b; or t has insignificant degree
  - Safe because removing insignificant neighbors with coalescing yields a subgraph of the graph obtained by removing those neighbors without coalescing

Simplification + Coalescing

- Consider M = set of move-related nodes (which appear in the source or destination of a move instruction) and N = all of the other variables
- Start by simplifying as many nodes as possible from N
- Coalesce some pairs of move-related nodes using conservative coalescing; delete corresponding mov instruction(s)
- Coalescing gives more opportunities for simplification: coalesced nodes may be simplified
- If can neither simplify nor coalesce, take a node in M and freeze all the move instruction involving that variable; go back to simplify.
- If all nodes frozen, no simplify possible, spill a variable

Full Algorithm



Overall Code Generation Process

- Start with low-level IR code
- Build DAG of the computation
  - Access global variables using static addresses
  - Access function arguments using frame pointer
  - Assume all local variables and temporaries are in registers (assume unbounded number of registers)
- Generate abstract assembly code
  - Perform tiling of DAG
- Register allocation
  - Live variable analysis over abstract assembly code
  - Assign registers and generate assembly code

Example

Program

```
array[int] a

function f:(int x) {
   int i;
   ...
   a[x+i] = a[x+i] + 1;
   ...
}
```

Low IR

```
t1 = x+i
t1 = t1*4
t1 = $a+t1
t2 = *t1
t2 = t2+1
t3 = x+i
t3 = t3*4
t3 = $a+t3
*t3 = t2
```

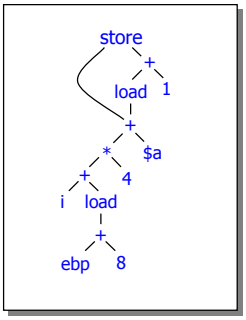

Accesses to Function Arguments

```
t1 = x+i
t1 = t1*4
t1 = $a+t1
t2 = *t1
t2 = t2+1
t3 = x+i
t3 = t3*4
t3 = $a+t3
*t3 = t2
```

```
t4 = ebp+8
t5 = *t4
t1 = t5+i
t1 = t1*4
t1 = $a+t1
t2 = *t1
t2 = t2+1
t6=ebp+8
t7 = *t6
t3 = t7+i
t3 = t3*4
t3 = $a+t3
*t3 = t2
```

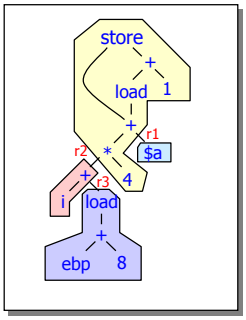

DAG Construction

```
t4 = ebp+8
t5 = *t4
t1 = t5+i
t1 = t1*4
t1 = $a+t1
t2 = *t1
t2 = t2+1
t6=ebp+8
t7 = *t6
t3 = t7+i
t3 = t3*4
t3 = $a+t3
*t3 = t2
```

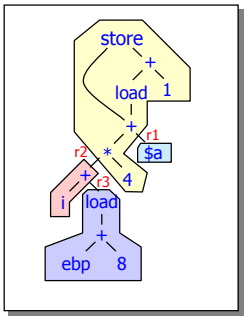


Tiling

- Find tiles
  - Maximal Munch
  - Dynamic programming
- Temporaries to transfer values between tiles
- No temporaries inside any of the tiles



Abstract Assembly Generation



Abstract Assembly

```
mov $a, r1
mov 8(%ebp), r3
mov i, r2
add r3, r2
add $1, (r1,r2,4)
```

Register Allocation

Abstract Assembly

```
mov $a, r1
mov 8(%ebp), r3
mov i, r2
add r3, r2
add $1, (r1,r2,4)
```

Live Variables

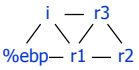
```
                    {%ebp, i}
mov $a, r1
                    {%ebp,r1,i}
mov 8(%ebp), r3
                    {r1, r3, i}
mov i, r2
                    {r1,r2,r3}
add r3, r2
                    {r1,r2}
add $1, (r1,r2,4)
                    {}
```


Register Allocation

Live Variables

```
                    {%ebp, i}
mov $a, r1
                    {%ebp,r1,i}
mov 8(%ebp), r3
                    {r1, r3, i}
mov i, r2
                    {r1,r2,r3}
add r3, r2
                    {r1,r2}
add $1, (r1,r2,4)
                    {}
```

- Build interference graph



- Allocate registers:
  eax: r1, ebx: r3
  i, r2 spilled to memory

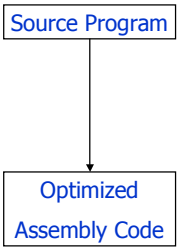
Assembly Code Generation

Abstract Assembly

```
mov $a, r1
mov 8(%ebp), r3
mov i, r2
add r3, r2
add $1, (r1,r2,4)
```

Assembly Code

```
mov $a, %eax
mov 8(%ebp), %ebx
mov –12(%ebp), %ecx
mov %ecx, -16(%ebp)
add %ebx, -16(%ebp)
mov –16(%ebp), %ecx
add $1, (%eax,%ecx,4)
```

Register allocation results:
eax: r1; ebx: r3; i, r2 spilled to memory

Where We Are