CS323: Compilers

Spring 2023

Week 10: Register allocation, Instruction Scheduling, Control Flow Graphs

Acknowledgements: Milind Kulkarni

Recall CSE

3 Address Code	Available expression(s)	Killed expression(s)	Generated Code (assembly)
			ld a r1; ld b r2;
ADD A B T1	{}		add r1 r2 r1
ADD T1 C T2	{“A + B”}	ld c r3;	add r1 r3 r2
ADD A B T3	{“A + B”, “T1 + C”}		mov r1 r3
ADD T1 T2 C	{“A + B”, “T1 + C”}	{“T1+C”}	add r1 r2 r5 st r5 c
ADD T1 C T4	{“A + B”, “T1 + T2”}		add r1 c r4
ADD T3 T2 D	{“A + B”, “T1 + T2”, “T1 + C”}		add r3 r2 r6 st r6 d
	{“A + B”, “T1 + T2”, “T1 + C”, “T3 + T2”}		

Register Allocation

- Simple code generation (in CSE example): use a register for each temporary, load from a variable on each read, store to a variable at each write

- What are the problems?
  - Real machines have a limited number of registers – one register per temporary may be too many
  - Loading from and storing to variables on each use may produce a lot of redundant loads and stores

Register Allocation

- Goal: allocate temporaries and variables to registers to:
  - Use only as many registers as machine supports
  - Minimize loading and storing variables to memory (keep variables in registers when possible)
  - Minimize putting temporaries on stack ("spilling")

Global vs. Local

- Same distinction as global vs. local CSE
  - Local register allocation is for a single basic block
  - Global register allocation is for an entire function

Does inter-procedural register allocation make sense? Why? Why not?

Hint: think about caller-save, callee-save registers

When we handle function calls, registers are pushed/popped from stack

Top-down register allocation

- For each basic block
  - Find the number of references of each variable
  - Assign registers to variables with the most references
- Details
  - Keep some registers free for operations on unassigned variables and spilling
  - Store *dirty* registers at the end of BB (i.e., registers which have variables assigned to them)
    - Do not need to do this for temporaries (why?)

Bottom-up register allocation

- Smarter approach:
  - Free registers once the data in them isn't used anymore
- Requires calculating *liveness*
  - A variable is live if it has a value that *may* be used in the future
- Easy to calculate if you have a single basic block:
  - Start at end of block, all local variables marked dead
    - If you have multiple basic blocks, all local variables defined in the block should be *live* (they may be used in the future)
  - When a variable is used, mark as live, record use
  - When a variable is defined, record def, variable dead above this
  - Creates chains linking uses of variables to where they were defined
- We will discuss how to calculate this across BBs later

Bottom-up register allocation

For each tuple op A B C in a BB, do
  R_x = ensure(A)
  R_y = ensure(B)
  if A *dead* after this tuple, free(R_x)
  if B *dead* after this tuple, free(R_y)
  R_z = allocate(C) //could use R_x or R_y
  generate code for op
  mark R_z *dirty*

At end of BB, for each dirty register
  generate code to store register into appropriate variable

- We will present this as if A, B, C are variables in memory. Can be modified to assume that A, B and C are in virtual registers, instead

Bottom-up register allocation

```
ensure(opr)
    if opr is already in register r
        return r
    else
        r = allocate(opr)
        generate load from opr into r
        return r
```

```
free(r)
    if r is marked dirty and variable is live
        generate store
    mark r as free
```

```
allocate(opr)
    if there is a free r
        choose r
    else
        choose r to free
        free(r)
    mark r associated with opr
    return r
```

Liveness Example

- What is live in this code? *Recall: a variable is live only if its value is used in future.*

Code	Live	Comments
`1: A = B + C`	{A, B}	Used B, C Killed A
`2: C = A + B`	{A, B, C}	Used A, B Killed C
`3: T1 = B + C`	{A, B, C, T1}	Used B, C Killed T1
`4: T2 = T1 + C`	{A, B, C, T2}	Used T1, C Killed T2
`5: D = T2`	{A, B, C, D}	Used T2, Killed D
`6: E = A + B`	{C, D, E}	Used A, B Killed E
`7: B = E + D`	{B, C, D}	Used E, D Killed B
`8: A = C + D`	{A, B}	Used C, D Killed A
`9: T3 = A + B`	{T3}	Used A, B Killed T3
`10: WRITE(T3)`	{}	Used T3

Bottom-up register allocation - Example

	Live	R1	R2	R3	R4	
		Registers				
1: A = 7	{A}	A*				mov 7 r1
2: B = A + 2	{A, B}	A*	B*			add r1 2 r2
3: C = A + B	{A, B, C}	A*	B*	C*		add r1 r2 r3
4: D = A + B	{B, C, D}	D*	B*	C*		add r1 r2 r1 (free r1 – dead)
5: A = C + B	{A, B, C, D}	D*	B*	C*	A*	add r3 r2 r4
6: B = C + B	{A, B, C, D}	D*	B*	C*	A*	add r3 r2 r2 (spill r2 – farthest, store if live and dirty) st r2 B;
7: E = C + D	{A, B, C, D, E}	D*	E*	C*	A*	add r3 r1 r2
8: F = C + D	{A, B, E, F}	F*	E*		A*	add r3 r1 r1 (Free dead)
9: G = A + B	{E, F, G}	F*	E*	G*		ld b r3; add r4 r3 r3
10: H = E + F	{H, G}	H*		G*		add r2 r1 r1 (Load since B not in reg. Free dead regs)
11: I = H + G	{I}	I*				add r1 r3 r1
12: WRITE(I)	{}					write r1