Code Generation

NI(E)-GEN, Spring 2021

https://courses.fit.cvut.cz/NI-GEN

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FIT

```
int foobar(int a, int b) {
    int c;
    if (a < b)
        c = a + b;
    else
        c = a - b;
    return c;
}
```

```
int foobar(int a, int b) {
    int c;
    if (a < b)
        c = a + b;
    else
        c = a - b;
    return c;
}
```

```
    0 = ARG_ADDR 0 // &a
    1 = ARG_ADDR 1 // &b
    2 = ALLOC_L int // &c
    3 = LOAD [0] // a
    4 = LOAD [1] // b
    5 = LT 3 4 // a < b ?
        COND_JUMP 5 ift iff
ift:
    6 = LOAD [0] // a
    7 = LOAD [1] // b
    8 = ADD 6 7 // a + b
        STORE [2] 8 // c =
        JMP after
iff:
    9 = LOAD [0] // a
   10 = LOAD [1] // b
   11 = SUB 9 10 // a - b
        STORE [2] 11 // c =
        JMP after
after:
   12 = LOAD [2] // c
        RET 12
```

```
int foobar(int a, int b) {
    int c;
    if (a < b)
        c = a + b;
    else
        c = a - b;
    return c;
}
```

```
    0 = ARG_ADDR 0 // &a
    1 = ARG_ADDR 1 // &b
    2 = ALLOC_L int // &c
    3 = LOAD [0] // a
    4 = LOAD [1] // b
    5 = LT 3 4 // a < b ?
        COND_JUMP 5 ift iff
ift:
    6 = LOAD [0] // a
    7 = LOAD [1] // b
    8 = ADD 6 7 // a + b
        STORE [2] 8 // c =
        JMP after
iff:
    9 = LOAD [0] // a
   10 = LOAD [1] // b
   11 = SUB 9 10 // a - b
        STORE [2] 11 // c =
        JMP after
after:
   12 = LOAD [2] // c
        RET 12
```

```
int foobar(int a, int b) {
    int c;
    if (a < b)
        c = a + b;
    else
        c = a - b;
    return c;
}
```

returns argument value

```
    0 = ARG 0 // a
    1 = ARG 1 // b
    2 = LT 0 1 // a < b ?
        COND_JUMP 5 ift iff
ift:
    3 = ADD 0 1 // a + b
        JMP after
iff:
    4 = SUB 9 10 // a - b
        JMP after
after:
    5 = PHI ift 3, iff 4
        RET 5
```

the usual phi node...

```
int foobar(int a, int b) {
    int c;
    if (a < b)
        c = a + b;
    else
        c = a - b;
    return c;
}
```

```
    0 = ARG 0 // a
    1 = ARG 1 // b
    2 = LT 0 1 // a < b ?
        COND_JUMP 5 ift iff
ift:
    3 = ADD 0 1 // a + b
        RET 3
iff:
    4 = SUB 9 10 // a - b
        RET 4
```

Registers Are Not Perfect

- not all variables can be kept in registers
- some need addresses

```
int foobar(int a, int b) {
    int c;
    if (a < b)
        c = a + b;
    else
        c = a - b;
    return c;
}
```

```
int foobar(int a, int b) {
    int c;
    if (a < b)
        c = a + b;
    else
        c = a - b;
    otherfoo(&c);
    return c;
}
```

Registers Are Not Perfect

- not all variables can be kept in registers
- some need addresses
  - references, pointers
  - shared across threads
  - …
- some start with an address
  - heap allocated
  - special memory (ports, DMA, etc.)

IR

- unlimited registers
  - SSA (not necessary)

- all registers created equal
  - i.e. any register can be used in any situation

IR

- unlimited registers
  - SSA (not necessary)
- all registers created equal
  - i.e. any register can be used in any situation

Target

- only a few registers
- not all registers equal
  - certain instructions only work with certain registers

Register Allocation

Allocation & Assignment

- allocation determines which values will stay in registers and which won’t

- assignment = for each value allocated to be in register, select the appropriate register

Value Spilling

- if there is not enough target registers, values must be stored in memory

- simple, remember we started with all values in memory in the IR
  - allocate space for the spilled values on stack
  - load / store as needed

Local Register Allocation

= single basic block

- in the beginning
  - all values in memory
- first read is load
  - all other reads just reuse the value in register
  - target registers, i.e. not SSA at this time
- last write is store
  - so that next basic blocks will have the information in memory

Local Register Allocation

- top-down:
  - loads & stores are costly, best savings when the most frequently accessed values are kept in registers
  - calculate usage frequencies, most frequent ones in registers
  - reserve enough registers for temporaries (statically known)

```
mov r0, 12
mov r1, 56
add r0, r1
add r1, 45
add r0, r1
add r0, 12
mov r3, r0
add r3, 12
```

```
mov r0, 12
mov r1, 56
add r0, r1
add r1, 45
add r0, r1
add r0, 12
mov r3, r0
add r3, 12
```

```
R0: 4 reads + 4 writes = 8
R1: 2 reads + 2 writes = 4
R3: 1 read  + 2 writes = 3
```

```
mov r0, 12
mov r1, 56
add r0, r1
add r1, 45
add r0, r1
add r0, 12
mov r3, r0
add r3, 12
```

```
R0: 4 reads + 4 writes = 8
R1: 2 reads + 2 writes = 4
R3: 1 read  + 2 writes = 3
```

```
mov r0, 12
mov r1, 56
add r0, r1
add r1, 45
add r0, r1
add r0, 12
mov r3, r0
add r3, 12
```

```
R0: 4 reads + 4 writes = 8
R1: 2 reads + 2 writes = 4
R3: 1 read  + 2 writes = 3
```

we have no registers to write this since all are occupied by values

Local Register Allocation

- top-down:
  - loads & stores are costly, best savings when the most frequently accessed values are kept in registers
  - calculate usage frequencies, most frequent ones in registers
  - reserve enough registers for temporaries (statically known)

- allocates registers for the **entire** basic block
  - this is often not necessary as values may not be used throughout the block
  - before first/after last use their register can be used for other values at no cost

Local Register Allocation

- bottom-up
  - at each instruction re-think the allocation
  - ensure operands are in registers
  - ensure result will be in register
  - if value no longer needed, forget register association
  - if there are no free registers and one is needed, use heuristic to spill
- heuristic
  - spill those that are used the furthest in the future

```
mov r0, 12
mov r1, 56
add r0, r1
add r1, 45
add r0, r1
add r0, 12
mov r3, r0
add r3, 12

free : ax bx
```

```
mov r0, 12
mov r1, 56
add r0, r1
add r1, 45
add r0, r1
add r0, 12
mov r3, r0
add r3, 12

free : bx
ax = r0
```

```
mov ax, 12
```

```
mov r0, 12
mov r1, 56
add r0, r1
add r1, 45
add r0, r1
add r0, 12
mov r3, r0
add r3, 12

free :
ax = r0
bx = r1
```

```
mov ax, 12
mov bx, 56
```

```
mov r0, 12
mov r1, 56
add r0, r1
add r1, 45
add r0, r1
add r0, 12
mov r3, r0
add r3, 12
```

```
mov ax, 12
mov bx, 56
add ax, bx
add bx, 41
add ax, bx
```

```
free :
ax = r0
bx = r1
```

```
mov r0, 12
mov r1, 56
add r0, r1
add r1, 45
add r0, r1
add r0, 12
mov r3, r0
add r3, 12
```

```
free : bx
ax = r0
```

```
mov ax, 12
mov bx, 56
add ax, bx
add bx, 41
add ax, bx
```

```
mov r0, 12
mov r1, 56
add r0, r1
add r1, 45
add r0, r1
add r0, 12
mov r3, r0
add r3, 12
```

```
mov ax, 12
mov bx, 56
add ax, bx
add bx, 41
add ax, bx
add ax, 12
mov bx, ax
```

```
free :
ax = r0
bx = r3
```

```
mov r0, 12
mov r1, 56
add r0, r1
add r1, 45
add r0, r1
add r0, 12
mov r3, r0
add r3, 12
```

```
mov ax, 12
mov bx, 56
add ax, bx
add bx, 41
add ax, bx
add ax, 12
mov bx, ax
```

```
free : ax
bx = r3
```

```
mov r0, 12
mov r1, 56
add r0, r1
add r1, 45
add r0, r1
add r0, 12
mov r3, r0
add r3, 12
```

```
mov ax, 12
mov bx, 56
add ax, bx
add bx, 41
add ax, bx
add ax, 12
mov bx, ax
add bx, 12
```

```
free : ax
bx = r3
```

Local Register Allocation

- bottom-up

- the devil lies in the details
  - don’t allocate two registers for same value
  - reurse return register if arguments no longer necessary
  - tag clean/dirty register values, prefer spilling clean values (eliminates store)

This is all nice…but:

This is all nice… but:

- code is rarely a single basic block
  - and basic blocks tend to be short (not much to optimize)
- can be faster!
  - local techniques do not optimize reuse across basic blocks
- is actually very slow:
  - passing values between basic blocks still involves memory (expensive)
  - **all** touched values are passed (more memory accesses)
- we need to consider whole functions hard:(

Global Register Allocation

- must account for control flow
- operates on IR register live ranges (NI-APR)

Live Ranges

- a range from first to last use (load/store) of a variable

```
int foo(int a) {
    int b = 1;
    int d;
    if (a == 5) {
        b = a;
    }
    d = someFunction(1,3);
    return b + d;
}
```

Live Ranges

- a range from first to last use (load/store) of a variable

```
int foo(int a) {
    int b = 1;
    int d;
    if (a == 5) {
        b = a;
    }
    d = someFunction(1,3);
    return b + d;
}
```

Live Ranges

- a range from first to last use (load/store) of a variable

```
int foo(int a) {                    a  b
    int b = 1;                      a  b
    int d;                          a  b
    if (a == 5) {                   a  b
        b = a;                      a  b
    }                                  b
    d = someFunction(1,3);             b  d
    return b + d;                      b  d
}
```

Live Ranges

- a range from first to last use (load/store) of a variable

```
int foo(int a) {                    
    int b = 1;                      a b
    int d;                          a b
    if (a == 5) {                   a b
        b = 1;                        b
    } else {                          b
        b = a + 1;                  a b
    }                                 b
    d = someFunction(1,3);            b d
    return b + d;                     b d
}
```

Global Register Allocation

- must account for control flow
- operates on IR register live ranges (NI-APR)

Linear Scan

- live ranges represented as intervals
- traversed chronologically, assigned registers in greedy manner
  - similar to bottom-up local approach
- fast (calculation) ...not so fast, actual code

Linear Scan

- live ranges represented as intervals
- traversed chronologically, assigned registers in greedy manner
  - similar to bottom-up local approach
- fast (calculation) ...not so fast, actual code
- the usual limitations of greedy algorithms
  - also, does not take holes in live ranges into account

So... Can we do better?

So... Can we do better?

If it's hard, make a graph of it...

- live ranges represented as graph nodes
- edges represent

- live ranges represented as graph nodes
- edges represent overlapping ranges

Register Allocation via Graph Coloring

- live ranges represented as graph nodes
- edges represent overlapping ranges
- colors for the graph represent target registers

is NP-complete :(

Graph Coloring

- key idea:
  - for K colors, a node with at most K-1 neighbors can always be safely colored. Just pick the color none of its neighbors have (1)

- find such nodes, color them and remove from the graph (2)
  - this might make some other nodes fall into the < K neighbors category

- if no such nodes are found, spill a range and remove it
  - this reduces the degree of its neighbors, if there is one with < K do (1), otherwise do (2)

Graph Coloring

- how to determine the color?
  - when removing nodes from graph, push to stack
  - when all on stack, pop from stack and assign lowest color none of its neighbors in the full graph has
- which ranges to spill?
  - this too complicates with control flow

Graph Coloring

- how to determine the color?
  - when removing nodes from graph, push to stack
  - when all on stack, pop from stack and assign lowest color none of its neighbors in the full graph has
- which ranges to spill?
  - this too complicates with control flow
  - assign spill costs to ranges (place/size/…)

The Other Things

- spilling can split the ranges
- ranges can be merged, if connected by a move
  - removes the move
- not all registers are created equal
  - and multiregister values
- whole program allocation
  - savings across function boundaries