CS 241 Lecture 22

Graham Cooper

July 22, 2015

Procedure Register Problem

Recall: Save local vars first then save regs

OR
Save all Regs in caller
BUT:

```
f() {
g();
g();
g();
g();
...
...
}

g() {
...
}
```

- callee-save
  - saves regs once per function
- caller-save
  - saves regs once per call!
  - may cost you space as shown above

Boilerplate code for Function calls

```
factor -> ID (expr1, exprn)
code(factor) =
push($29)
push($31)
code(expr1)
```

```
push($3)
...
code(exprn)
push($3)
lis $5
.word FID
jalr $5
pop all args
pop($31)
pop($29)

code(procedure) =
sub $29, $30, $4
push(dcls)
push(reg)
code(stmts)
code(expr)
pop(regs)
add $30, $29, $4
jr $31
```

Optimization

Computationally unsolvable. Can only approximate.

Eg 1+2

```
lis $3
.word 1
sw $3, -4($30)
sub $30, $30, $4
lis $3
.word 2
lw $5, 0($30)
add $30, $30, $4
add $3, $5, $3
```

We could have it in 2 instructions:

```
lis $3
.word 3
```

- called constant folding

Constant propagation

int x = 1;
x + x;

```
lis $3
.word 1
sub $3, -4($30)
sub $30m $30, $4
lw $3, -12($29) (Assuming x is at -12($29))
sw $3, -4($30)
sub $30, $30, $4
lw $3, -12($29)
lw $5, 0($30)
add $30, $30, $4
add $3, $5, $3
```

Could recognize that x = 1 and do:

```
lis $3
.word 1
sw $3, -4($30)
sub $30, $30, $4
lis $3
.word 2
```

IF that's the only place x is used, it doesn't need a stack entry.

```
list $3
.word 2
```

Watch out for aliasing!!

```
int x = 1;
int y = NULL;
y = &x;
*y = 2;
x + x;
```

Even if x's value is not known, could recognize that $3 already contains x:

```
lw $3, -12($29)
add $3, $3, $3
```

This is known as common subexpression elimination
- use a register to hold at b, then multiply it by itself, rather than compute a + b twice.

Dead Code Elimination

- if you are certain that some branch of a program will never run, do not output code for it. ie. if statements that are always false

Register Allocation

- cheaper to use regs for vasr instead of the stack
- (saves lw's and sw's)
- eg, regs $12-28 unused by our code gen
- we could use these registers to hold 17 variables and use the stack for the rest

Which 17 variables should we store on the stack?
- probably the most used ones
Issue: & address of operator - if you take the address of a variable, it can't be in a register, it must be in ram

Strength Reduction

- add usually runs faster than mult (prefer add over mult if we have a choice)
- could add to itself rather than multiply by 2

Procedure-specific Optimizations

0.0.1 Inlining

```
int f(int x){
return x + x;
}
int wain(int a, int b){
return f(a);
}

Could we turn wain into
int wain(int a, int b){
return a + a;
}
```

- replace the function call wiht its body, right in the caller.
- saves the overhead of a function call
- IS it always a win?
- if F is called many times, we get many copies of its body
- some functions are harder to inline, ie recursive functions
- if all calls to f are inline, then we do not need to generate code for f

0.0.2 Tail Recursion

```
int fact(int n, int a){
if (n == 0) return a;
else return fact(n-1, n*a);
}
```

tail recursive because the recursive call is the last thigns that hte function does.
No pending work to do when a recursive call returns
Re-use the content of the current frame, so we do not need to pop the frame off the stack!
Can this be done in WLP4?
- no because return is at the end and no returns are inside of ifs
but can we transform the program?
Basic Transformation: When return immediately follows an if-statement push inside both branches

```
if(){
-- if(){
-- }
-- else {
-- }
}
else {
}
return x;
```

SAME AS

```
if(){
-- if(){
-- -- return x;
-- }
-- else {
-- -- return x;
-- }
}
```

```
else {
-- return x;
}
```

Whenever rturn x follows an assgn to x, merge: x = f(...)
return x;

into
return f(...)

WLP4?