

CIS 341: Compilers

Lecture 24

The Plan

- Today:
 - Optimization
- Project 5: Full OAT compiler: objects and extended typechecking
 - Due: This Friday, November 7th at 11:59 pm

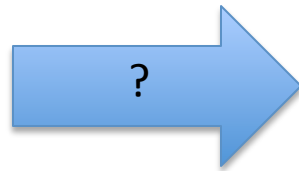
Optimizations

- The code generated by our OAT compiler so far is pretty inefficient.
 - Lots of redundant moves.
 - Lots of unnecessary arithmetic instructions.
- Consider this OAT program:

```
int foo(int w) (  
    int x = 3 + 5;  
    int y = x * w;  
    int z = y - 0;  
    z * 4  
)
```

Unoptimized vs. Optimized Output

```
__fun__foo:
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
__3:
    movl $3, -16(%ebp)
    addl $5, -16(%ebp)
    movl -16(%ebp), %ecx
    movl %ecx, -12(%ebp)
    movl -12(%ebp), %ecx
    movl %ecx, -16(%ebp)
    movl 8(%ebp), %ecx
    movl -16(%ebp), %eax
    imull %ecx, %eax
    movl %eax, -16(%ebp)
    movl -16(%ebp), %ecx
    movl %ecx, -8(%ebp)
    movl -8(%ebp), %ecx
    movl %ecx, -16(%ebp)
    subl $0, -16(%ebp)
    movl -16(%ebp), %ecx
    movl %ecx, -4(%ebp)
    movl -4(%ebp), %ecx
    movl %ecx, -16(%ebp)
    movl -16(%ebp), %eax
    imull $4, %eax
    movl %eax, -16(%ebp)
    movl -16(%ebp), %eax
    movl %ebp, %esp
    popl %ebp
    ret
```



Hand optimized code:

```
__fun__foo:
    movl -4(%esp), %eax
    shl $5, %eax
    ret
```

Equivalent OAT source:

```
int foo(int w) ( w << 5 )
```

- Function foo may be inlined by the compiler, so it can be implemented by just one instruction!

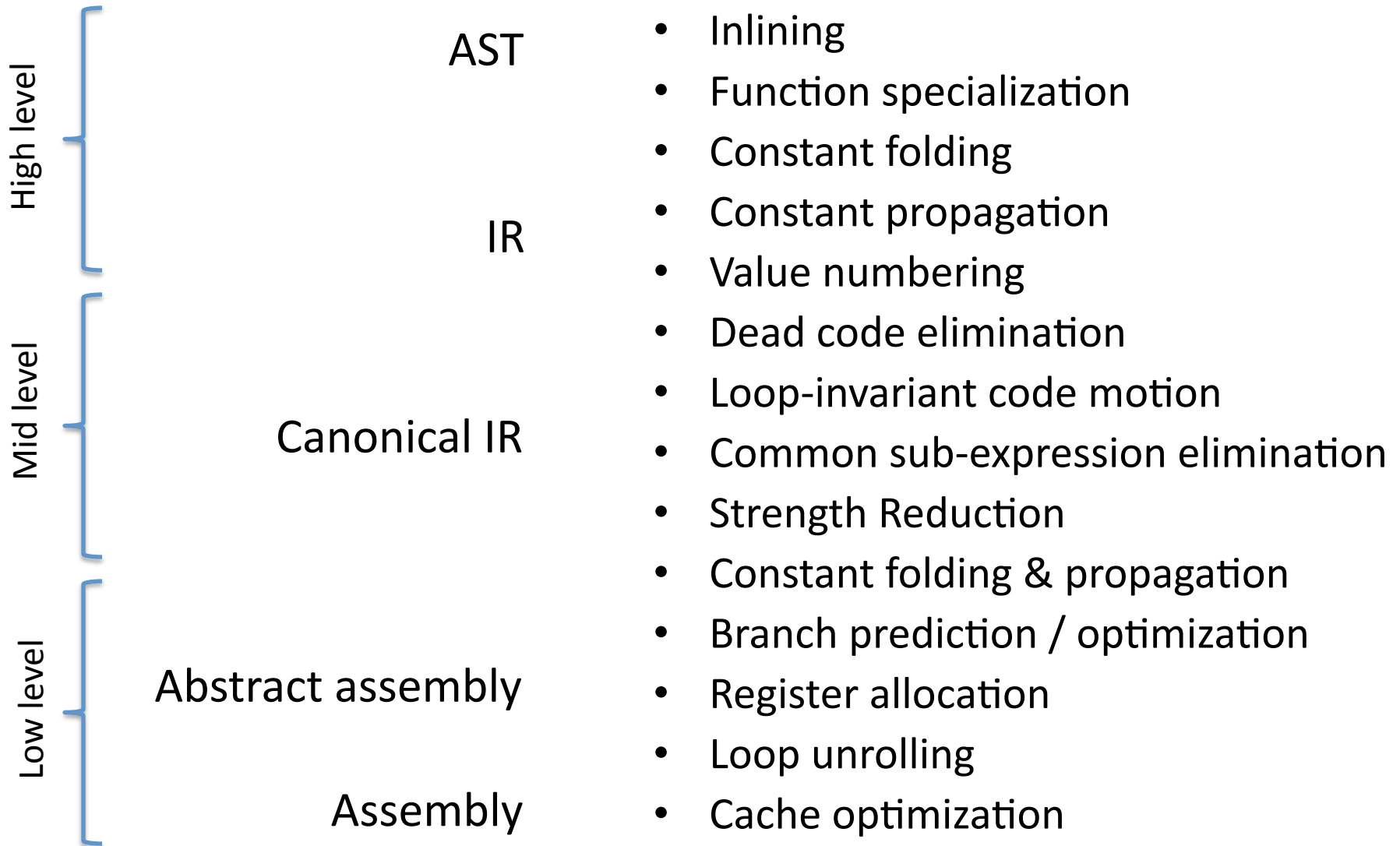
Why do we need optimizations?

- To help programmers...
 - They write modular, clean, high-level programs
 - Compiler generates efficient, high-performance assembly
- Programmers don't write optimal code
- High-level languages make avoiding redundant computation inconvenient or impossible
 - e.g. $A[i][j] = A[i][j] + 1$
- Architectural independence
 - Optimal code depends on features not expressed to the programmer
 - Modern architectures *assume* optimization
- Different kinds of optimizations:
 - Time: improve execution speed
 - Space: reduce amount of memory needed
 - Power: lower power consumption (e.g. to extend battery life)

Some caveats

- Optimization are code transformations:
 - They can be applied at any stage of the compiler
 - They must be *safe* – they can't change the meaning of the program.
- In general, optimizations require some program analysis:
 - To determine if the transformation really is safe
 - To determine whether the transformation is cost effective
- This course: most common and valuable performance optimizations
 - See Muchnick (optional text) for ~10 chapters about optimization

When to apply optimization



Where to Optimize?

- Usual goal: improve time performance
- Problem: many optimizations trade space for time
- Example: *Loop unrolling*
 - Idea: rewrite a loop like:

```
for(int i=0; i<100; i=i+1) {  
    s = s + a[i];  
}
```
 - Into a loop like:

```
for(int i=0; i<99; i=i+2){  
    s = s + a[i];  
    s = s + a[i+1];  
}
```
- Tradeoffs:
 - Increasing codes space slows down whole program a tiny bit but speeds up the loop
 - Frequently executed code with long loops, generally a win
 - Interacts with instruction cache and branch prediction hardware
- Complex optimizations may never pay off!

Writing Fast Programs In Practice

- Pick the right algorithms and data structures.
 - These have a much bigger impact on performance than compiler optimizations.
 - Reduce # of operations
 - Reduce memory accesses
 - Minimize indirection – it breaks working-set coherence
- *Then* turn on compiler optimizations
- Profile to determine program hot spots
- Evaluate whether the algorithm/data structure design works
- ...if so: “tweak” the source code until the optimizer does “the right thing” to the machine code

Safety

- Whether an optimization is *safe* depends on the programming language semantics.
 - Languages that provide weaker guarantees to the programmer permit more optimizations, but have more ambiguity in their behavior.
 - e.g. In Java tail-call optimization (that turns recursive function calls into loops) is not valid.
- Example: *loop-invariant code motion*
 - Idea: hoist invariant code out of a loop

```
while (b) {  
  z = y/x;  
  ...           // y, x not updated  
}
```



```
z = y/x;  
while (b) {  
  ...           // y, x not updated  
}
```

- Is this more efficient?
- Is this safe?

Constant Folding

- Idea: If operands are known at compile time, perform the operation statically.

int x = (2 + 3) * y → int x = 5 * y
b & false → false

- Performed at every stage of optimization...
- Why?
 - Constant expressions can be created by translation or earlier optimizations
- Example: **A[2]** might be compiled to:
MEM[MEM[A] + 2 * 4] → MEM[MEM[A] + 8]

Constant Folding Conditionals

if (true) S → S

if (false) S → ;

if (true) S else S' → S

if (false) S else S' → S'

while (false) S → ;

if (2 > 3) S → ;

Algebraic Simplification

- More general form of constant folding
 - Take advantage of mathematically sound simplification rules
- Identities:
 - $a * 1 \rightarrow a$ $a * 0 \rightarrow 0$
 - $a + 0 \rightarrow a$ $a - 0 \rightarrow a$
 - $b | \text{false} \rightarrow b$ $b \& \text{true} \rightarrow b$
- Reassociation & commutativity:
 - $(a + 1) + 2 \rightarrow a + (1 + 2) \rightarrow a + 3$
 - $(2 + a) + 4 \rightarrow (a + 2) + 4 \rightarrow a + (2 + 4) \rightarrow a + 6$
- Strength reduction: (replace expensive op with cheaper op)
 - $a * 4 \rightarrow a \ll 2$
 - $a * 7 \rightarrow (a \ll 3) - a$
 - $a / 32767 \rightarrow (a \gg 15) + (a \gg 30)$
- Note 1: must be careful with floating point (due to rounding)
- Note 2: iteration of these optimizations is useful... how much?

Constant Propagation

- If the value is known to be a constant, replace the use of the variable by the constant
- Value of the variable must be propagated forward from the point of assignment
 - This is a substitution operation

- Example:

```
int x = 5;
```

```
int y = x * 2; → int y = 5 * 2; → int y = 10; →
```

```
int z = a[y];      int z = a[y];      int z = a[y];  int z = a[10];
```

- To be most effective, constant propagation should be interleaved with constant folding

Copy Propagation

- If one variable is assigned to another, replace uses of the assigned variable with the copied variable.
- Need to know where copies of the variable propagate.
- Interacts with the scoping rules of the language.

- Example:

```
x = y;  
if (x > 1) {  
    x = x * f(x - 1);  
}  
→  
x = y;  
if (y > 1) {  
    x = y * f(y - 1);  
}
```

- Can make the first assignment to *x* *dead* code (that can be eliminated).

Dead Code Elimination

- If a side-effect free statement can never be observed, it is safe to eliminate the statement.

```
x = y * y // x is dead!  
...      // x never used → ...  
x = z * z      x = z * z
```

- A variable is *dead* if it is never used after it is defined.
 - Computing such *definition* and *use* information is an important component of compiler
- Dead variables can be created by other optimizations...