Lecture 21

CIS 4521/5521: COMPILERS

Announcements

- HW5: Oat v. 2.0
  - records, function pointers, type checking, array-bounds checks, etc.
  - typechecker & safety
  - Due: Tomorrow, April 9th
  - Test cases due ***tonight!***

- HW6: Optimizations
  - Program analysis and register allocation
  - Available: end of this week
  - Due: Wednesday, April 30th

MULTIPLE INHERITANCE

General Approaches

- Can't directly identify methods by position anymore.

- Option 1: Use a level of indirection:
  - Map method identifiers to code pointers (e.g. index by method name)
  - Use a hash table
  - May need to do search up the class hierarchy

- Option 2: Give up separate compilation
  - Use "sparse" dispatch vectors, or binary decision trees
  - Must know then entire class hierarchy

- Option 3: Allow multiple D.V. tables (C++)
  - Choose which D.V. to use based on static type
  - Casting from/to a class may require run-time operations

- Note: many variations on these themes
  - Different Java compilers pick different approaches to options1 and 2…

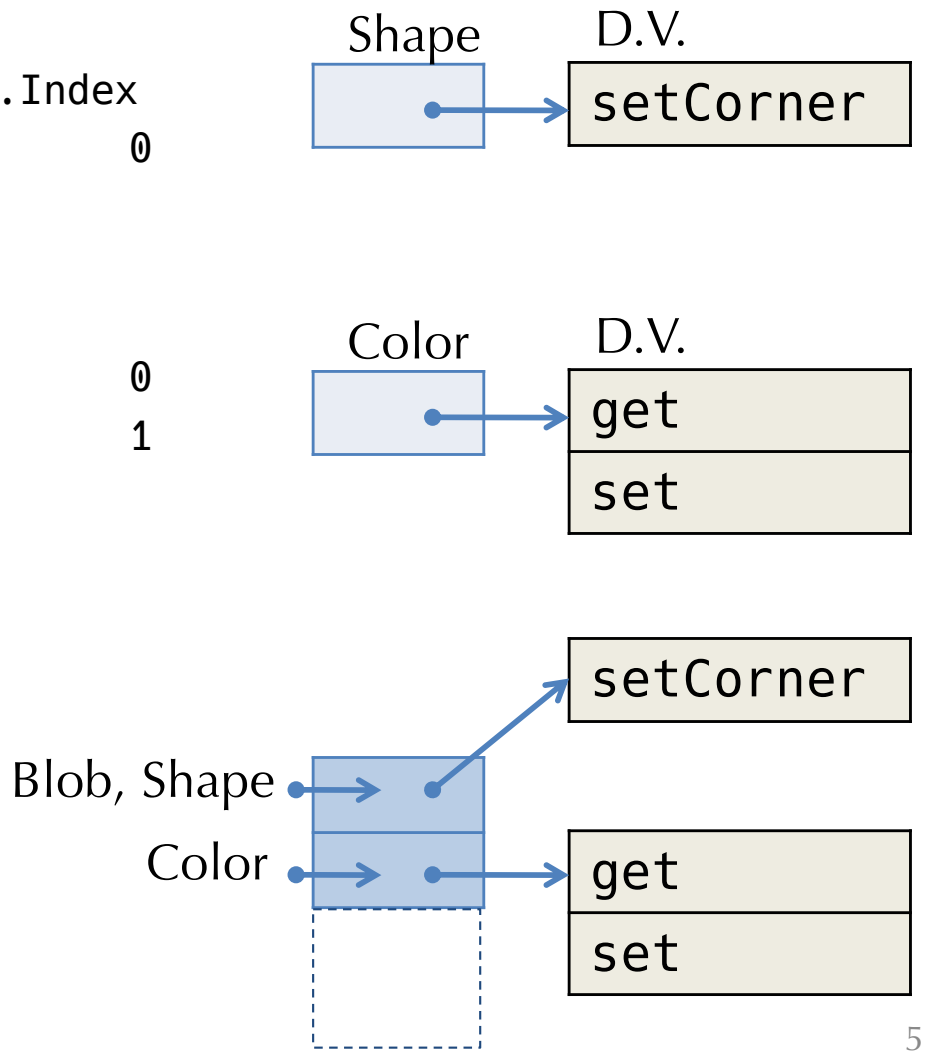
Option 3: Multiple Dispatch Vectors

- Duplicate the D.V. pointers in the object representation.
- Static type of the object determines which D.V. is used.

```
interface Shape {                          D.V.Index
  void setCorner(int w, Point p);                0
}

interface Color {
  float get(int rgb);                            0
  void set(int rgb, float value);                1
}

class Blob implements Shape, Color {
  void setCorner(int w, Point p) {…}
  float get(int rgb) {…}
  void set(int rgb, float value) {…}
}
```



Multiple Dispatch Vectors

- A reference to an object might have multiple "entry points"
  - Each entry point corresponds to a dispatch vector
  - Which one is used depends on the statically known type of the program.

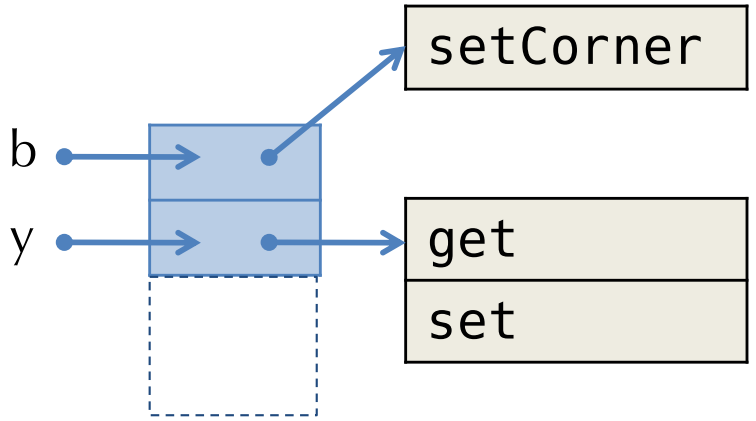
```
Blob b = new Blob( );
Color y = b;     // implicit cast!
```

- Compile

```
Color y = b;
```

As

```
Movq ⟦b⟧ + 8 , y
```



Multiple D.V. Summary

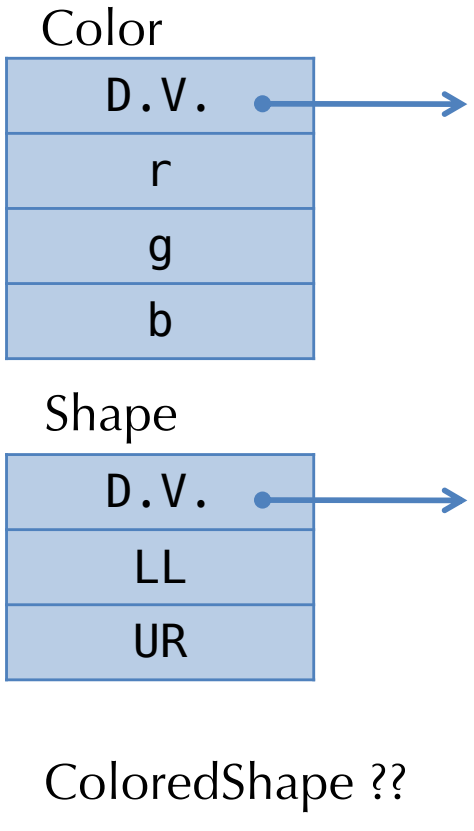
- Benefit: Efficient dispatch, same cost as for multiple inheritance
- Drawbacks:
  - Cast has a runtime cost
  - More complicated programming model… hard to understand/debug?

- What about multiple inheritance and fields?

Multiple Inheritance: Fields

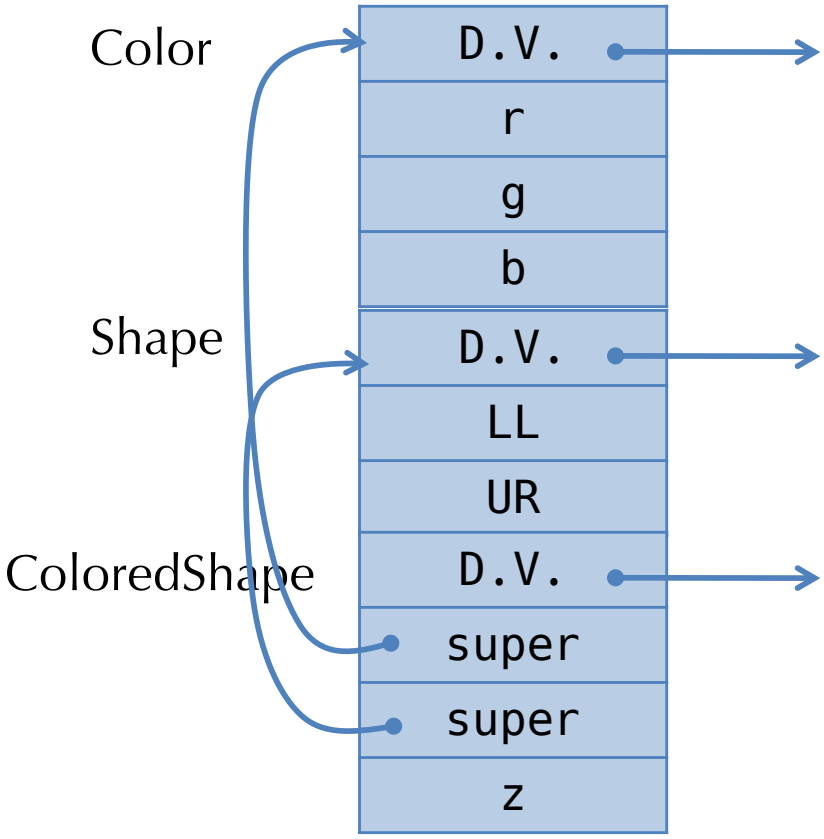
- Multiple supertypes (Java): methods conflict (as we saw)
- Multiple inheritance (C++): fields can also conflict
- Location of the object's fields can no longer be a constant offset from the start of the object.

```
class Color {
  float r, g, b; /* offsets: 4,8,12 */
}
class Shape {
  Point LL, UR; /* offsets: 4, 8 */
}
class ColoredShape extends
Color, Shape {
  int z;
}
```



C++ approach:

- Add pointers to the superclass fields
  - Need to have multiple dispatch vectors anyway (to deal with methods)
- Extra indirection needed to access superclass fields
- Used even if there is a single superclass
  - Uniformity



Compiling lambda calculus to straight-line code.

Representing evaluation environments at runtime.

CLOSURE CONVERSION REVISITED

Compiling First-class Functions

- To implement first-class functions on a processor, there are two problems:
  - First: we must implement substitution of free variables
  - Second: we must separate 'code' from 'data'

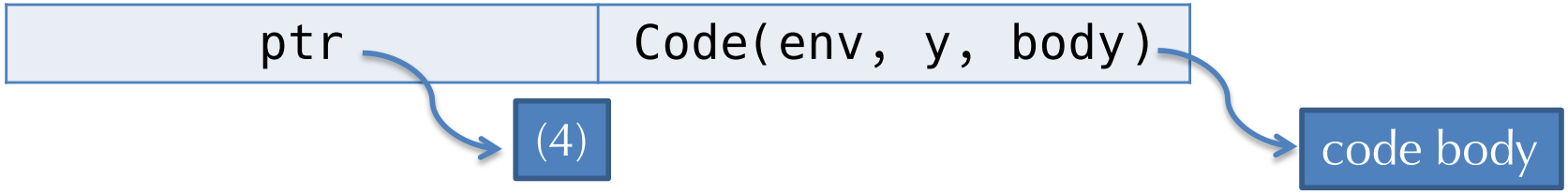
- Reify the substitution:
  - Move substitution from the meta language to the object language by making the data structure & lookup operation explicit
  - The environment-based interpreter is one step in this direction
- Closure Conversion:
  - Eliminates free variables by packaging up the needed environment in the data structure.
- Hoisting:
  - Separates code from data, pulling closed code to the top level.

Example of closure creation

- Recall the "add" function:
  `let add = fun x -> fun y -> x + y`

- Consider the inner function: `fun y -> x + y`

- When run the function application: `add 4`
  the program builds a closure and returns it.
  - The closure is a pair of the environment and a code pointer.



- The code pointer takes a pair of parameters: env and y
  - The function code is (essentially):
    `fun (env, y) -> let x = nth env 0 in x + y`

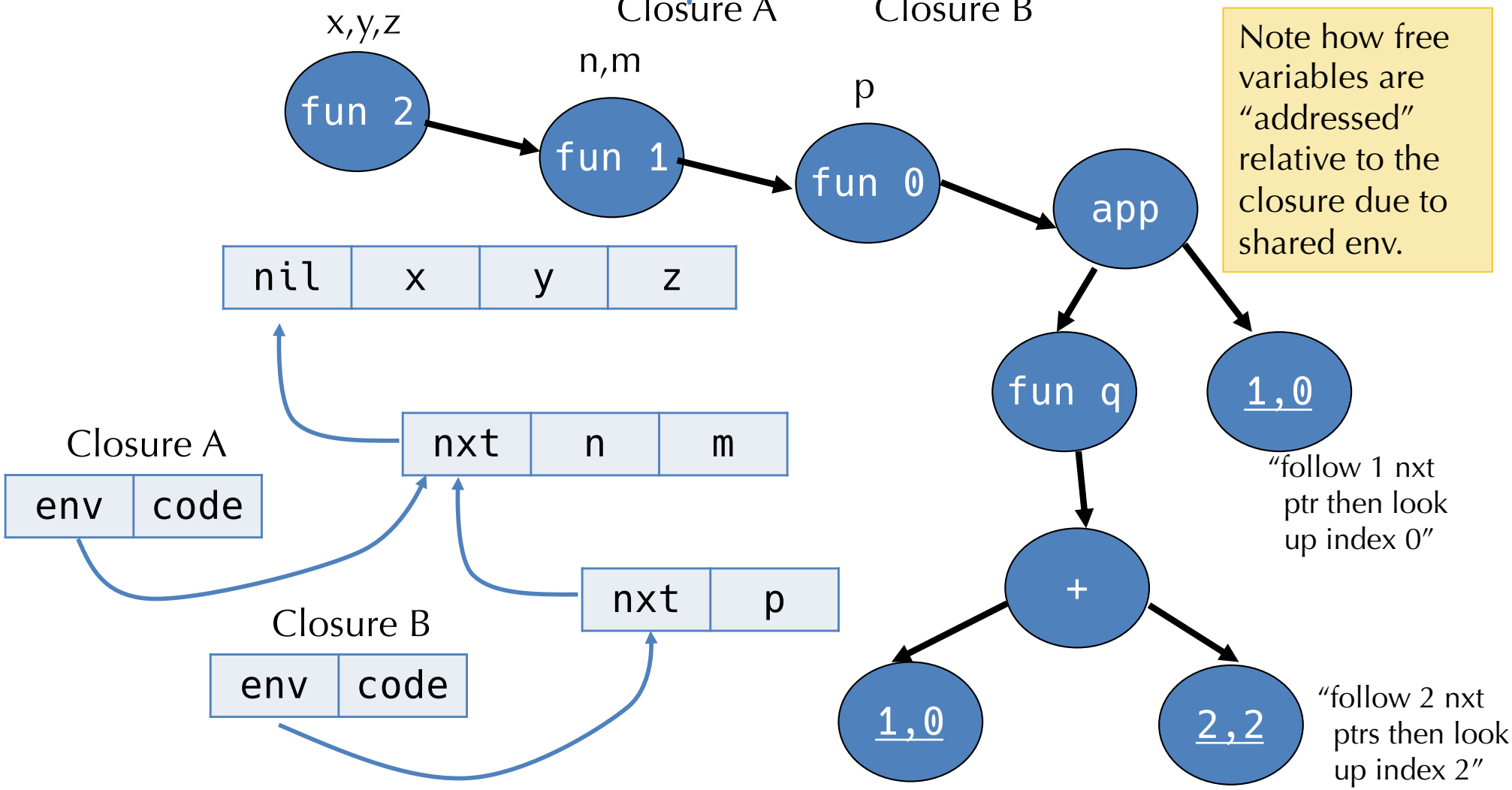
Representing Closures

- As we saw, the simple closure conversion algorithm doesn't generate very efficient code.
  - It stores all the values for variables in the environment, even if they aren't needed by the function body.
  - It copies the environment values each time a nested closure is created.
  - It uses a linked-list datastructure for tuples.

- There are many options:
  - Store only the values for free variables in the body of the closure.
  - Share subcomponents of the environment to avoid copying
  - Use vectors or arrays rather than linked structures

Array-based Closures with N-ary Functions

```
(fun (x y z) ->
   (fun (n m) -> (fun p -> (fun q -> n + z) x)
```

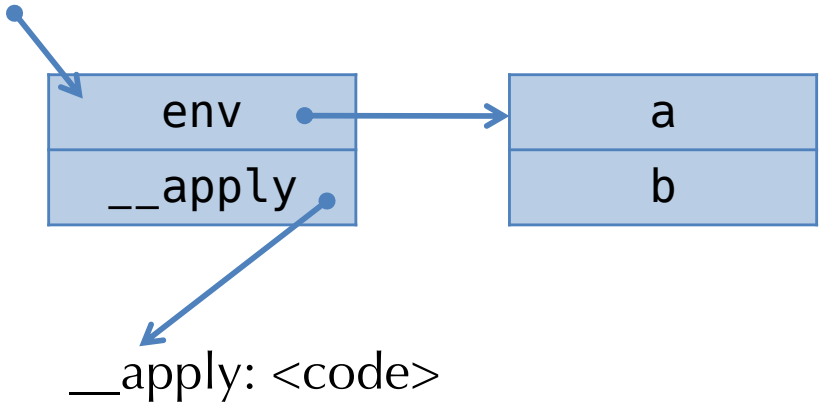


Note how free variables are "addressed" relative to the closure due to shared env.

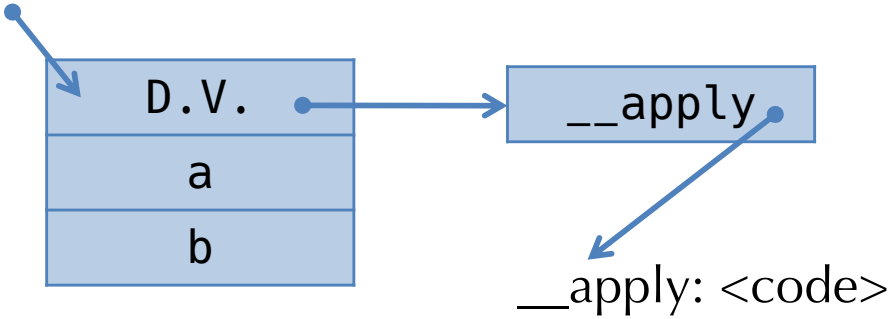
Observe: Closure ≈ Single-method Object

- Free variables ≈ Fields
- Environment pointer ≈ "this" parameter
- Closure for function: ≈ Instance of this class:

```
fun (x,y) ->
   x + y + a + b
```




```
class C {
   int a, b;
   int apply(x,y) {
      x + y + a + b
   }
}
```



Optimizations

Source Code
(Character stream)

```
if (b == 0) { a = 1; }
```

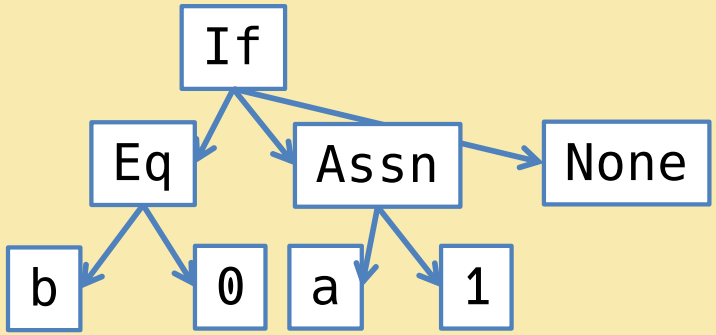
Lexical Analysis

Token stream:

if	(	b	==	0	)	{	a	=	0	;	}

Parsing

Abstract Syntax Tree:



Intermediate code:

```
l1:
  %cnd = icmp eq i64 %b,
0
  br i1 %cnd, label %l2,
label %l3
l2:
  store i64* %a, 1
  br label %l3
l3:
```

Analysis &
Transformation

Backend

Assembly Code

```
l1:
  cmpq %eax, $0
  jeq l2
  jmp l3
l2:
  …
```

Why optimize?

OPTIMIZATIONS, GENERALLY

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) {
  var x = 3 + 5;
  var y = x * w;
  var z = y - 0;
  return z * 4;
}
```


```
define i64 @foo(i64 %_w1) {
  %_w2 = alloca i64
  %_x5 = alloca i64
  %_y10 = alloca i64
  %_z14 = alloca i64
  store i64 %_w1, i64* %_w2
  %_bop4 = add i64 3, 5
  store i64 %_bop4, i64* %_x5
  %_x7 = load i64, i64* %_x5
  %_w8 = load i64, i64* %_w2
  %_bop9 = mul i64 %_x7, %_w8
  store i64 %_bop9, i64* %_y10
  %_y12 = load i64, i64* %_y10
  %_bop13 = sub i64 %_y12, 0
  store i64 %_bop13, i64* %_z14
  %_z16 = load i64, i64* %_z14
  %_bop17 = mul i64 %_z16, 4
  ret i64 %_bop17
}
```

- opt-example.c, opt-example.oat

Unoptimized vs. Optimized Output

```
define i64 @foo(i64 %_w1) {
  %_w2 = alloca i64
  %_x5 = alloca i64
  %_y10 = alloca i64
  %_z14 = alloca i64
  store i64 %_w1, i64* %_w2
  %_bop4 = add i64 3, 5
  store i64 %_bop4, i64* %_x5
  %_x7 = load i64, i64* %_x5
  %_w8 = load i64, i64* %_w2
  %_bop9 = mul i64 %_x7, %_w8
  store i64 %_bop9, i64* %_y10
  %_y12 = load i64, i64* %_y10
  %_bop13 = sub i64 %_y12, 0
  store i64 %_bop13, i64* %_z14
  %_z16 = load i64, i64* %_z14
  %_bop17 = mul i64 %_z16, 4
  ret i64 %_bop17
}
```

backend.ml

```
        .text
        .globl  _foo
_foo:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $136, %rsp
        movq    %rdi, %rax
        movq    %rax, -8(%rbp)
        pushq   $0
        movq    %rsp, -16(%rbp)
        pushq   $0
        movq    %rsp, -24(%rbp)
        pushq   $0
        movq    %rsp, -32(%rbp)
        pushq   $0
        movq    %rsp, -40(%rbp)
        movq    -8(%rbp), %rcx
        movq    -16(%rbp), %rax
        movq    %rcx, (%rax)
        movq    $3, %rax
        movq    $5, %rcx
        addq    %rcx, %rax
        movq    %rax, -56(%rbp)
        movq    -56(%rbp), %rcx
        movq    -24(%rbp), %rax
        movq    %rcx, (%rax)
        movq    -24(%rbp), %rax
        movq    (%rax), %rcx
        movq    %rcx, -72(%rbp)
        movq    -16(%rbp), %rax
        movq    (%rax), %rcx
        movq    %rcx, -80(%rbp)
        movq    -72(%rbp), %rax
        movq    -80(%rbp), %rcx
        imulq   %rcx, %rax
        movq    %rax, -88(%rbp)
        movq    -88(%rbp), %rcx
        movq    -32(%rbp), %rax
        movq    %rcx, (%rax)
        movq    -32(%rbp), %rax
        movq    (%rax), %rcx
        movq    %rcx, -104(%rbp)
        movq    -104(%rbp), %rax
        movq    $0, %rcx
        subq    %rcx, %rax
        movq    %rax, -112(%rbp)
        movq    -112(%rbp), %rcx
        movq    -40(%rbp), %rax
        movq    %rcx, (%rax)
        movq    -40(%rbp), %rax
        movq    (%rax), %rcx
        movq    %rcx, -128(%rbp)
        movq    -128(%rbp), %rax
        movq    $4, %rcx
        imulq   %rcx, %rax
        movq    %rax, -136(%rbp)
        movq    -136(%rbp), %rax
        movq    %rbp, %rsp
        popq    %rbp
        retq
```

???

Optimized code:

```
_foo:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    %rdi, %rax
        shlq    $5, %rax
        popq    %rbp
        retq
```

- Code above generated by `clang -O3`
- 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`

In Oat/ Java it's not possible for the programmer to manually express the sharing of the two computations of A[i][j] because there is no concept of "interior pointer".

- 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 *sound* – they shouldn'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

> ***(static) program analysis:*** the process of (soundly) approximating the dynamic behavior of a program at compile time, usually by representing some facts about the state of the computation at each program point.

- This course: most common and valuable performance optimizations
  - See Muchnick (optional text) for ~10 chapters about optimization

When to apply optimization

AST

IR

Canonical IR

Abstract assembly

Assembly

- Inlining
- Function specialization
- Constant folding
- Constant propagation
- Value numbering
- Dead code elimination
- Loop-invariant code motion
- Common sub-expression elimination
- Strength Reduction
- Constant folding & propagation
- Branch prediction / optimization
- Register allocation
- Loop unrolling
- Instruction Selection
- Cache 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 code space slows down whole program a tiny bit (extra instructions to manage) but speeds up the loop a lot
  - For 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 that 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

Soundness

- Whether an optimization is ***sound*** (i.e., correct) 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 C, writing to unallocated memory is undefined behavior, so the compiler can do anything if a program writes to an array out of bounds.
  - *e.g.*, In Java, tail-call optimization (which turns recursive function calls into loops) is not valid because of "stack inspection".
- 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?

Comparing Behaviors*

- Consider two programs P1 and P2 possibly in different languages.
  - e.g. P1 is an Oat program, P2 is its compilation to LL

- The semantics of the languages associate to each program a set of observable behaviors:

$$\mathfrak{B}(P) \text{ and } \mathfrak{B}(P')$$

- Note: $|\mathfrak{B}(P)| = 1$ if P is deterministic, > 1 otherwise

*Note: this series of slides is adapted from some by Xavier Leroy from a summer school about compiler verification.

What is Observable?

- For C-like languages:

  observable behavior ::=
  | terminates(st) (i.e. observe the final state)
  | diverges
  | goeswrong

- For pure functional languages:

  observable behavior ::=
  | terminates(v) (i.e. observe the final value)
  | diverges
  | goeswrong

What about I/O?

- Add a *trace* of input-output events performed:

```
          t     ::=  []  |  e :: t          (finite traces)
coind.    T     ::=  []  |  e :: T          (finite and infinite traces)
```

```
observable behavior ::=
          | terminates(t, st)     (end in state st after trace t)
          | diverges(T)           (loop, producing trace T)
          | goeswrong(t)
```

Examples

- P1:
  `print(1);` / st ⇒ terminates(out(1)::[],st)

- P2:
  `print(1); print(2);` / st
  ⇒ terminates(out(1)::out(2)::[],st)

- P3:
  `WHILE true DO print(1) END` / st
  ⇒ diverges(out(1)::out(1)::…)

- So $\mathfrak{B}(P1) \neq \mathfrak{B}(P2) \neq \mathfrak{B}(P3)$

Bisimulation

- Two programs P1 and P2 are bisimilar whenever:

$$\mathfrak{B}(P1) = \mathfrak{B}(P2)$$

- The two programs are completely indistinguishable.

- But… this is often too strong in practice.

Compilation Reduces Nondeterminism

- Some languages (like C) have underspecified behaviors:
  - Example: order of evaluation of expressions f() + g()

- Concurrent programs often permit nondeterminism
  - Classic optimizations can reduce this nondeterminism
  - Example:

```
a := x + 1; b := x + 1      ||      x := x+1
```

vs.

```
a := x + 1; b := a          ||      x := x+1
```

- LLVM explicitly allows nondeterminism:
  - undef values (not part of LLVM lite)
  - see the discussion later

Backward Simulation / Refinement

- Program P2 can exhibit fewer behaviors than P1:

$$\mathfrak{B}(P1) \supseteq \mathfrak{B}(P2)$$

- All the behaviors of P2 are permitted by P1, though some of them may have been eliminated.
- Also called *refinement*.

What about goeswrong?

- Compilers often translate away bad behaviors.

`x := 1/y ; x := 42` vs. `x := 42`

(divide by 0 error) (always terminates)

- Justifications:
  - Compiled program does not "go wrong" because the program type checks or is otherwise formally verified
  - Or just "garbage in/garbage out"

Safe Backwards Simulation

- Only require the compiled program's behaviors to agree if the source program could not go wrong:

$$\text{goeswrong}(t) \notin \mathfrak{B}(P1) \quad \Rightarrow \quad \mathfrak{B}(P1) \supseteq \mathfrak{B}(P2)$$

This definition of "safe backwards simulation" is typically what we mean by "correctness" of a program transformation.

A high-level tour of a variety of optimizations.

BASIC OPTIMIZATIONS

Constant Folding

- Idea: If operands are known at compile type, 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             ➔
  if (false) S           ➔ ;
```

Algebraic Simplification

- More general form of constant folding
  - Take advantage of mathematically sound simplification rules

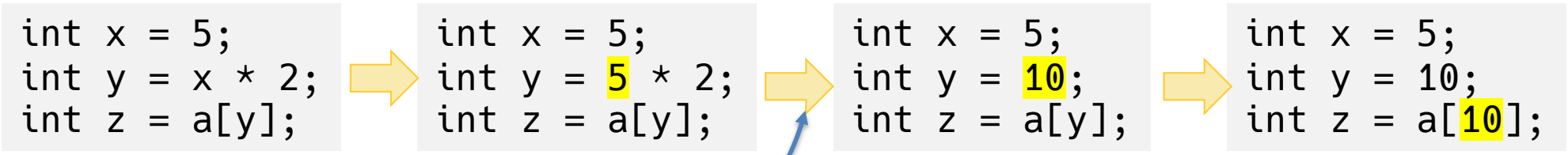
- ***Mathematical identities***:
  - `a * 1 ➔ a` `a * 0 ➔ 0`
  - `a + 0 ➔ a` `a - 0 ➔ a`
  - `b | false ➔ b` `b & true ➔ b`
- ***Reassociation & commutativity***:
  - `(a + 1) + 2 ➔ a + (1 + 2) ➔ a + 3`
  - `(2 + a) + 4 ➔ (a + 2) + 4 ➔ a + (2 + 4) ➔ a + 6`
- ***Strength reduction***: (replace expensive op with cheaper op)
  - `a * 4 ➔ a << 2`
  - `a * 7 ➔ (a << 3) - a`
  - `a / 32767 ➔ (a >> 15) + (a >> 30)`

- *Note 1:* must be careful with floating point (due to rounding) and integer arithmetic (due to overflow/underflow)
- *Note 2:* iteration of these optimizations is useful… how much?
- *Note 3:* must be sure that rewrites terminate:
  - commutativity apply like: `(x + y) ➔ (y + x) ➔ (x + y) ➔ (y + x) ➔ …`

Constant Propagation

- If a variable 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:



- 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 program analysis

- Dead variables can be created by other optimizations…

```
int x = 5;
int y = x * 2;
int z = a[y];
```



```
~~int x = 5;~~
~~int y = 10;~~
int z = a[10];
```

Unreachable/Dead Code

- Basic blocks not reachable by any trace leading from the starting basic block are ***unreachable*** and can be deleted.
  - Performed at the IR or assembly level
  - Improves cache, TLB performance

- Dead code: similar to unreachable blocks.
  - A value might be computed but never subsequently used.
- Code for computing the value can be dropped
- But only if it's ***pure***, *i.e.*, it has *no externally visible side effects*
  - Externally visible effects: raising an exception, modifying a global variable, going into an infinite loop, printing to standard output, sending a network packet, launching a rocket
  - Note: Pure functional languages (*e.g.*, Haskell) make reasoning about the safety of optimizations (and code transformations in general) easier!

Inlining

- Replace a call to a function with the body of the function itself with arguments rewritten to be local variables:
- Example in OAT: inline `pow` into `g`

```
int g(int x) { return x + pow(x); }
int pow(int a) {
    var b = 1; var x = 0;
    while (x < a) {b = 2 * b; x = x + 1}
    return b;
}
```

note: renaming

```
int g(int x) {
    int a = x;
    int b = 1; int x2 = 0;
    while (x2 < a) {b = 2 * b; x2 = x2 + 1};
    tmp = b;
    return x + tmp;
}
```

- May need to rename variables to avoid *capture*
  - See lecture about ***capture avoiding substitution*** for lambda calculus
- Best done at the AST or relatively high-level IR.
- When is it profitable?
  - Eliminates the stack manipulation, jump, etc.
  - Can increase code size.
  - Enables further optimizations

Code Specialization

- Idea: create specialized versions of a function that is called from different places with different arguments.
- Example: specialize function `f` in:

```
class A implements I { int m() {…} }
class B implements I { int m() {…} }
int f(I x) { x.m(); }          // don't know which m
A a = new A(); f(a);           // know it's A.m
B b = new B(); f(b);           // know it's B.m
```

- `f_A` would have code specialized to dispatch to `A.m`
- `f_B` would have code specialized to dispatch to `B.m`
- You can also inline methods when the run-time type is known statically
  - Often just one class implements a method.

Common Subexpression Elimination

- ***fold redundant computations together***
  - in some sense, it's the opposite of inlining
- Example:

```
a[i] = a[i] + 1
```

compiles to:

```
MEM[a + i*8] := MEM[a + i*8] + 1
```

Common subexpression elimination removes the redundant add and multiply:

```
t = a + i*8; MEM[t] := MEM[t] + 1
```

- For safety, you must be sure that the shared expression always has the same value in both places!

Unsafe Common Subexpression Elimination

- Example: consider this OAT function:

```
unit f(int[] a, int[] b, int[] c) {
  var j = …; var i = …; var k = …;
  b[j] = a[i] + 1;
  c[k] = a[i];
  return;
}
```

- The optimization that shares the expression `a[i]` is unsafe… why?

```
unit f(int[] a, int[] b, int[] c) {
  var j = …; var i = …; var k = …;
  t = a[i];
  b[j] = t + 1;
  c[k] = t;
  return;
}
```

LOOP OPTIMIZATIONS

Loop Optimizations

- Program hot spots often occur in loops.
  - Especially inner loops
  - Not always: consider operating systems code or compilers vs. a computer game or word processor

- Most program execution time occurs in loops.
  - The 90/10 rule of thumb holds here too.
    (90% of the execution time is spent in 10% of the code)

- Loop optimizations are very important, effective, and numerous
  - Also, concentrating effort to improve loop body code is usually a win

Loop Invariant Code Motion (revisited)

- Another form of redundancy elimination.
- If the result of a statement or expression does not change during the loop *and* it's pure, it can be hoisted outside the loop body.
- Often useful for array element addressing code
  - Invariant code not visible at the source level

```
for (i = 0; i < a.length; i++) {
    /* a not modified in the body */
}
```

```
t = a.length;
for (i =0; i < t; i++) {
 /* same body as above */
}
```

Hoisted loop-invariant expression

Strength Reduction (revisited)

- Strength reduction can work for loops too
- Idea: replace expensive operations (multiplies, divides) by cheap ones (adds and subtracts)
- For loops, create a *dependent induction variable*:

- Example:

```
for (int i = 0; i<n; i++) { a[i*3] = 1; }  // stride by 3
```

```
int j = 0;
for (int i = 0; i<n; i++) {
  a[j] = 1;
  j = j + 3;     // replace multiply by add
}
```

Loop Unrolling (revisited)

- Branches can be expensive, unroll loops to avoid them.

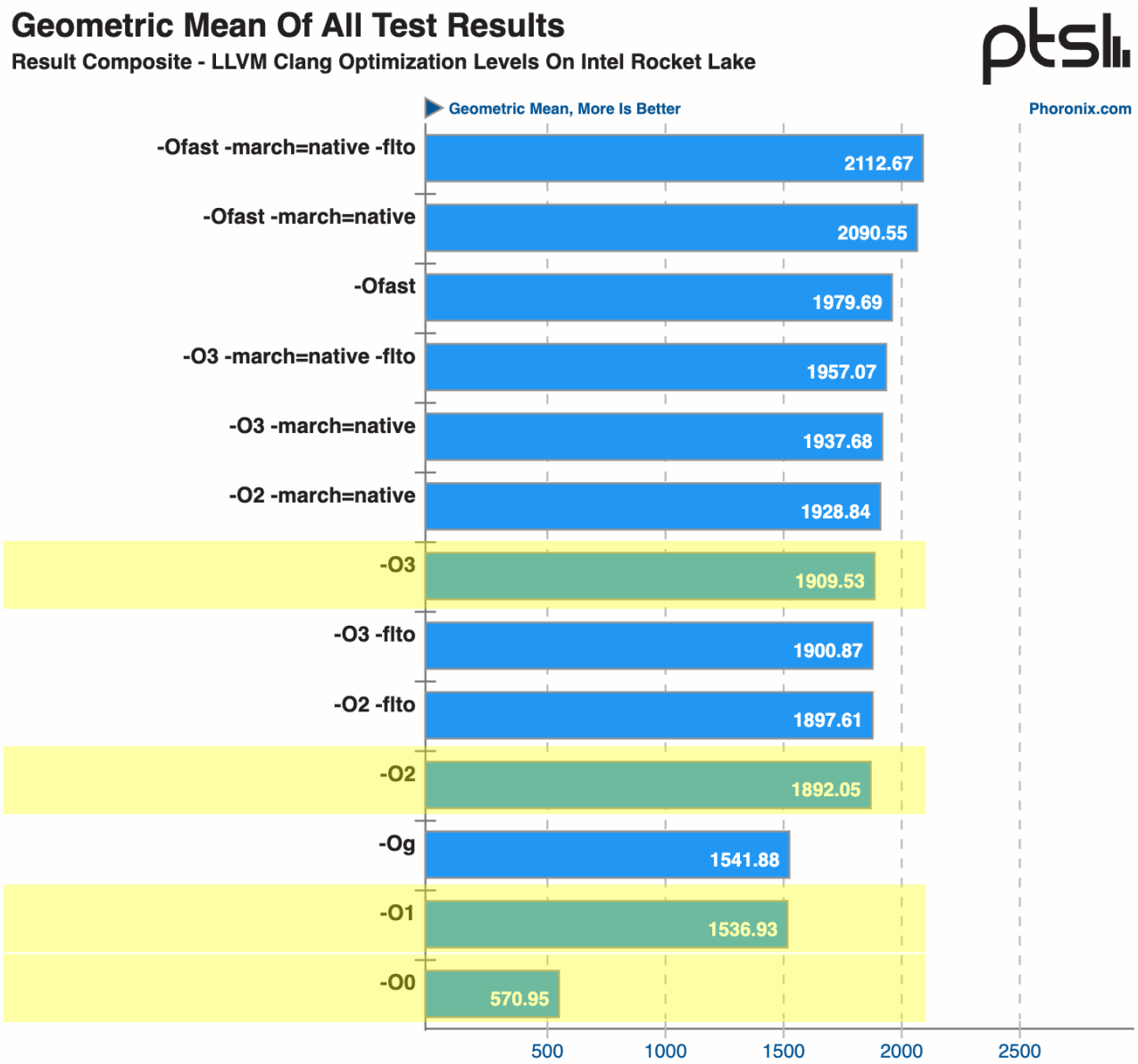
```
for (int i=0; i<n; i++) { S }
```

```
for (int i=0; i<n-3; i+=4) {S;S;S;S};
for (       ; i<n; i++) { S } // left over iterations
```

- With k unrollings, eliminates (k-1)/k conditional branches
  - So for the above program, it eliminates ¾ of the branches
- Space-time tradeoff:
  - Not a good idea for large S or small n
- Interacts with instruction caching, branch prediction

EFFECTIVENESS?

Optimization Effectiveness?



Geom. mean over 44 benchmark programs at various –O levels.

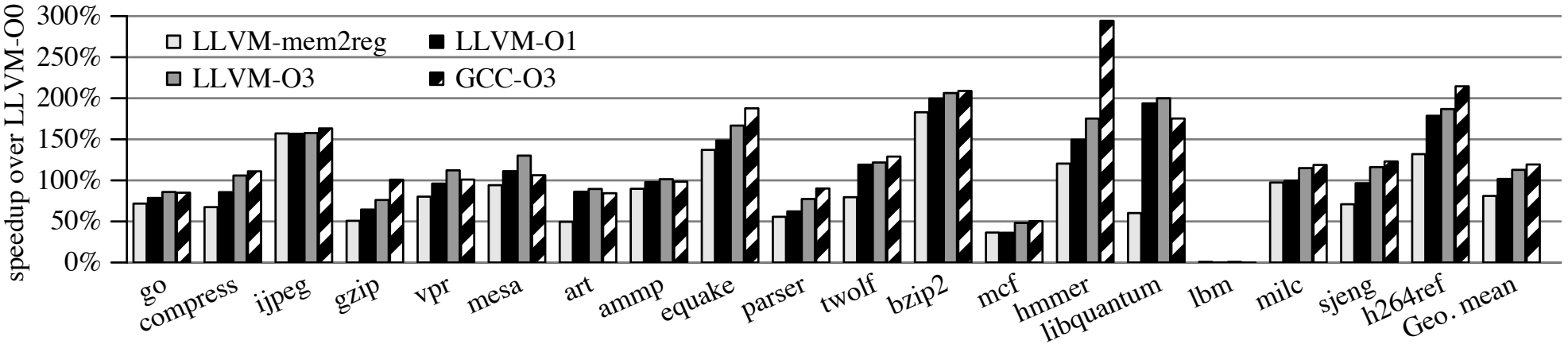
Clang 12

https://www.phoronix.com/review/clang-12-opt

LLVM Clang 12 Benchmarks At Varying Optimization Levels, LTO

25 June 2021

Optimization Effectiveness?



$$\%\text{speedup} = \left[\frac{\text{base time}}{\text{optimized time}} - 1 \right] \times 100\%$$

Example:

base time = 2s

optimized time = 1s ⇒ 100% speedup

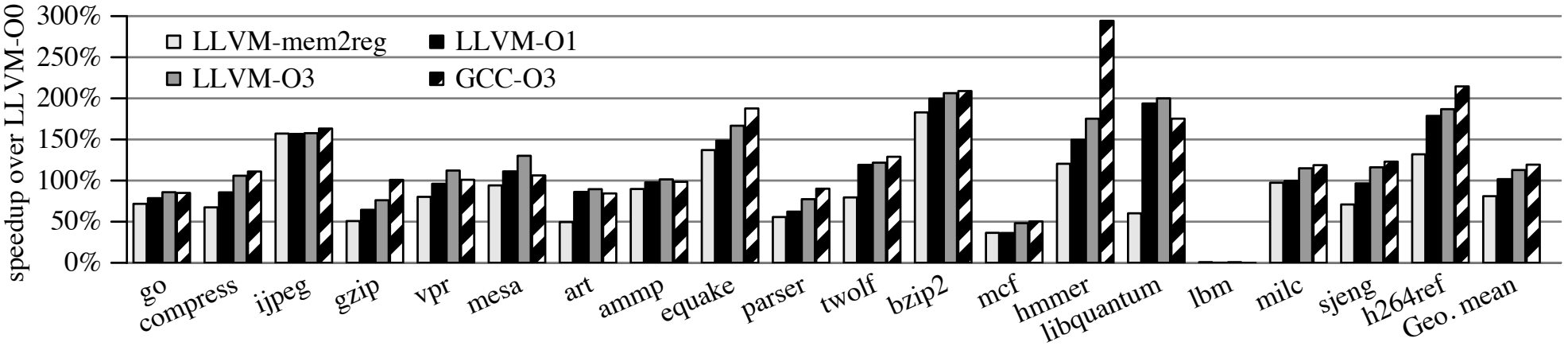
Example:

base time = 1.2s

optimized time = 0.87s ⇒ 38% speedup

Graph taken from:
Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic.
Formal Verification of SSA-Based Optimizations for LLVM.
In Proc. 2013 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI), 2013

Optimization Effectiveness?



- mem2reg: promotes alloca'ed stack slots to temporaries to enable register allocation
- Analysis:
  - mem2reg alone (+ back-end optimizations like register allocation) yields ~78% speedup on average
  - -O1 yields ~100% speedup
    (so all the rest of the optimizations combined account for ~22%)
  - -O3 yields ~120% speedup
- Hypothetical program that takes 10 sec. (base time):
  - Mem2reg alone: expect ~5.6 sec
  - -O1: expect ~5 sec
  - -O3: expect ~4.5 sec