YSC4230: Programming Language Design and Implementation

Week 11: Code Optimizations

Ilya Sergey

ilya.sergey@yale-nus.edu.sg

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;
}
```

frontend.ml

```
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
}
```

Optimized vs Non-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

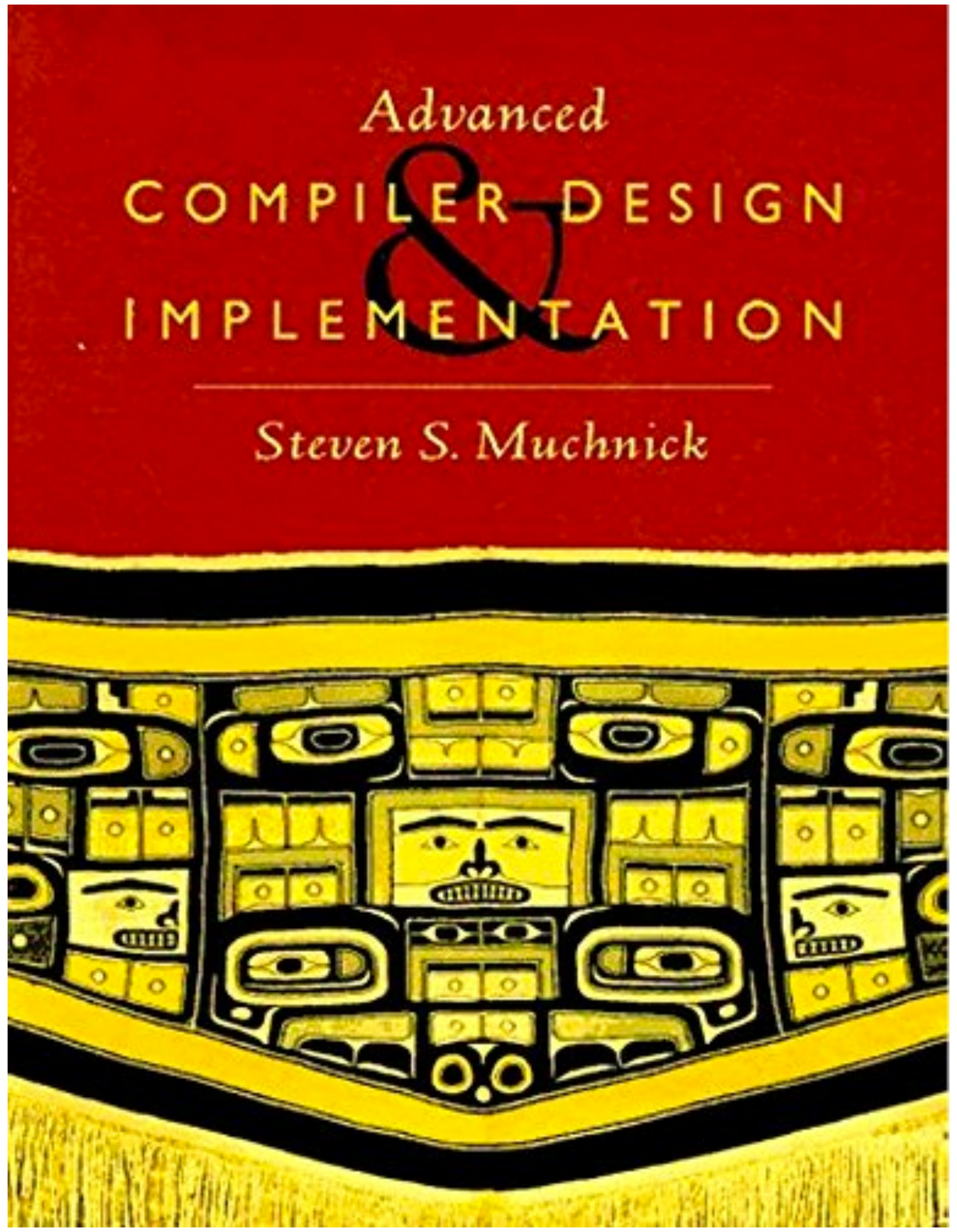
- High-level languages make avoiding redundant computation inconvenient or impossible
  - e.g. `A[i][j] = A[i][j] + 1`

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



When to apply optimization

High level
- AST
- IR

Mid level
- Canonical IR

Low level
- 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
- Cache optimization

A good place to have a break

Where to Optimize?

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

```
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

- *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 behaviour.
  - e.g. In C, loading from initialized memory is undefined, so the compiler can do anything.

- 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?

The Zoo of 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	➔ ;

Algebraic Simplification

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

- 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: must be careful with floating point (due to rounding) and integer arithmetic (due to overflow/underflow)

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;                        ~~x = y;~~
if (x > 1) {        ➔        if (y > 1) {
  x = x * f(x – 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…

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
- 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 with the body of the function itself with arguments rewritten to be local variables:
- Example in Oat code:

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

- May need to rename variable names to avoid *name capture*
  - Example of what can go wrong?
- 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

```
int g(int x) ( 1 + f(x) )
Int f(int a) (a + x)
➔
const int x = 3;
int g(int x) ( 1 + (int a = x; a + x) )
```

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 (CSE)

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

  a[i] = a[i] + 1 compiles to:

  [a + i*4] = [a + i*4] + 1

Common subexpression elimination removes the redundant add and multiply:

  t = a + i*4; [t] = [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) {
    int j = …; int i = …; int 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) {
    int j = …; int i = …; int k = …;
    t = a[i];
    b[j] = t + 1;
    c[k] = t;
    return;
}
```

Loop Optimizations

Loop Optimizations

- 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
  - so-called invariant code

```
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.

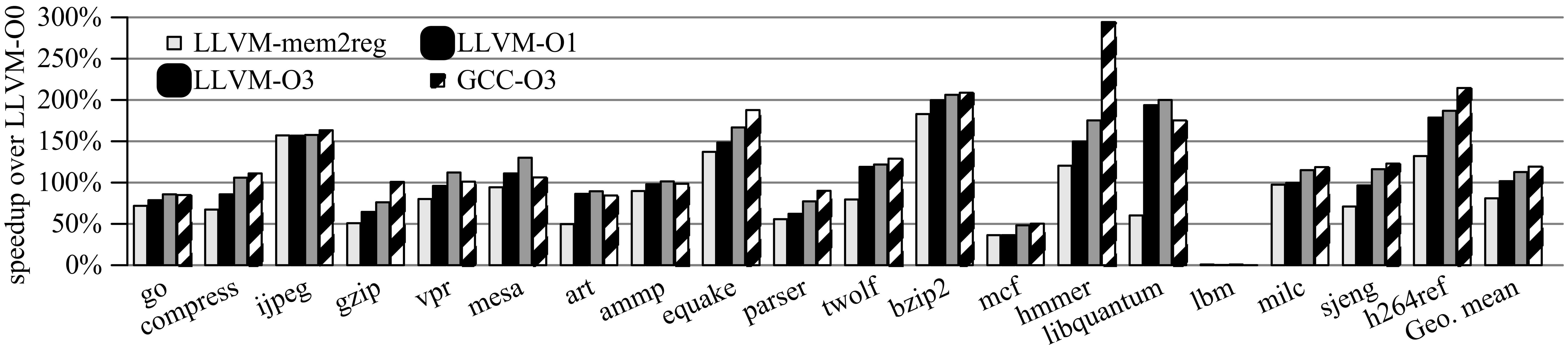
```
for (int i=0; i < n; i++) { S }
```

```
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

Optimization Effectiveness

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

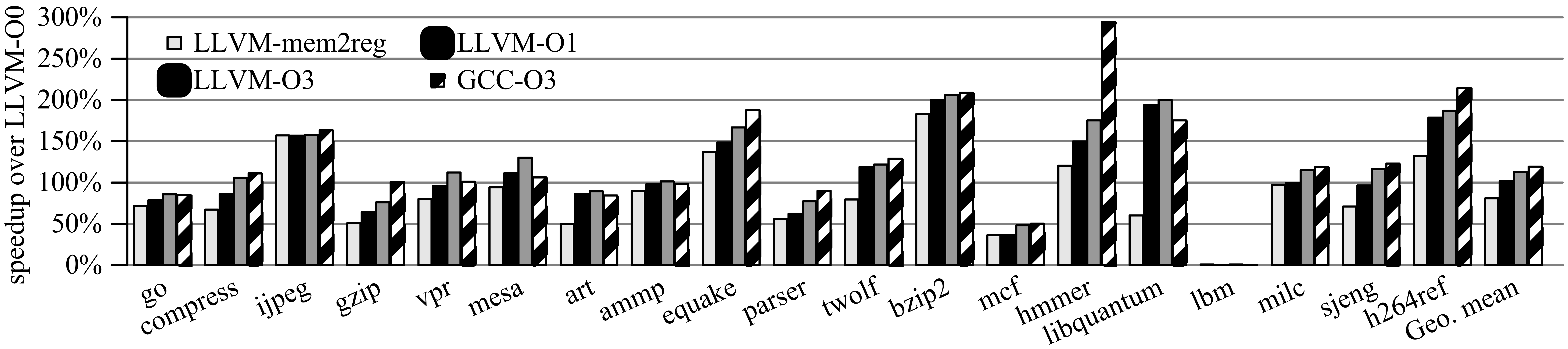
Example:

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

Code Analysis

Motivating Code Analyses

- There are lots of things that might influence the safety/applicability of an optimization
  - What algorithms and data structures can help?

- How do you know what is a loop?
- How do you know an expression is invariant (constant)?
- How do you know if an expression has no side effects?
- How do you keep track of where a variable is defined?
- How do you know where a variable is used?
- How do you know if two reference values may be aliases of one another?

Moving Towards Register Allocation

- The Oat compiler currently generates as *many* temporary variables as it needs
  - These are the %uids you should be very familiar with by now.
- Current compilation strategy:
  - Each %uid maps to a stack location.
  - This yields programs with many loads/stores to memory.
  - Very inefficient.
- Ideally, we'd like to map as many %uid's as possible into registers.
  - Eliminate the use of the alloca instruction?
  - Only 16 max registers available on 64-bit X86
  - %rsp and %rbp are reserved and some have special semantics, so really only 10 or 12 available
  - This means that a register must hold more than one slot
- When is this safe?

Liveness

- Observation: %uid1 and %uid2 can be assigned to the same register if their values will *not be needed* at the same time.
  - What does it mean for an %uid to be "needed"?
  - Ans: its contents will be used as a source operand in a later instruction.

- Such a variable is called "*live*"

- Two variables can share the same register if they are *not* live at the same time.

Scope vs. Liveness

- We can already get some coarse liveness information from variable scoping.
- Consider the following OAT program:

```
int f(int x) {
    var a = 0;
    if (x > 0) {
        var b = x * x;
        a = b + b;
    }
    var c = a * x;
    return c;
}
```

- Note that due to Oat's scoping rules, variables b and c can never be live at the same time.
  - c's scope is disjoint from b's scope
- So, we could assign b and c to the same alloca'ed slot and potentially to the same register.

But Scope is too Coarse

- Consider this program:

```
int f(int x) {
  int a = x + 2;
  int b = a * a;
  int c = b + x;
  return c;
}
```

int a = x + 2;	x is live
int b = a * a;	a and x are live
int c = b + x;	b and x are live
return c;	c is live

- The scopes of a, b, c, x all overlap – they're all in scope at the end of the block.
- But, a, b, c are *never live at the same time*.
  - So they can share the same stack slot / register

Live Variable Analysis

- A variable v is *live* at a program point if v is defined before the program point and used after it.
- Liveness is defined in terms of where variables are *defined* and where variables are *used*
- Liveness analysis: Compute the live variables between each statement.
  - May be *conservative* (i.e. it may claim a variable is live when it isn't) so because that's a safe approximation
  - To be useful, it should be more *precise* than simple scoping rules.
- Liveness analysis is one example of *dataflow analysis*
  - Other examples: Available Expressions, Reaching Definitions, Constant-Propagation Analysis, …

Control-flow Graphs Revisited

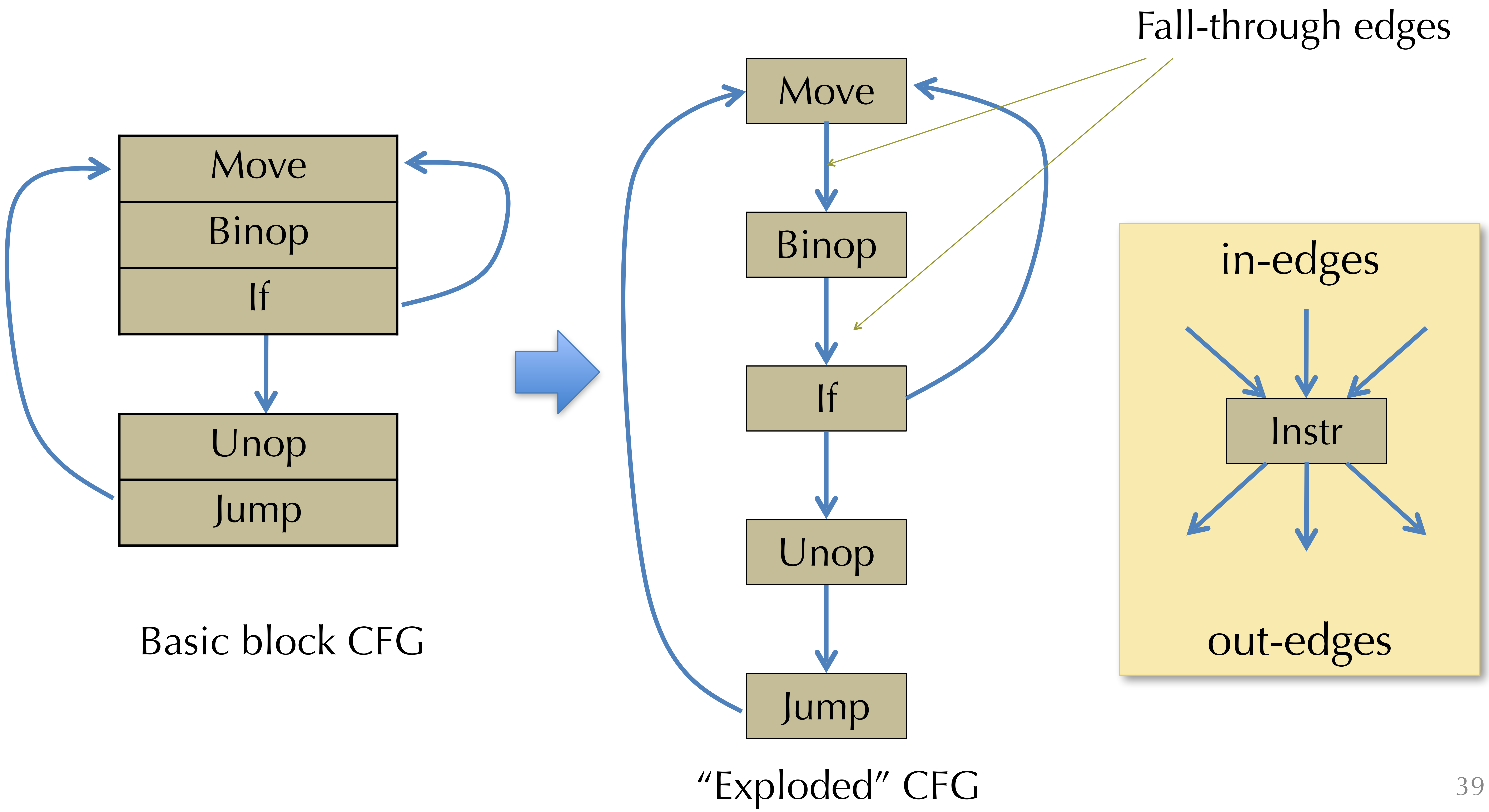
- For the purposes of dataflow analysis, we use the *control-flow graph* (CFG) intermediate form.

- Recall that a basic block is a sequence of instructions such that:
  - There is a distinguished, labeled entry point (no jumps into the middle of a basic block)
  - There is a (possibly empty) sequence of non-control-flow instructions
  - The block ends with a single control-flow instruction (jump, conditional branch, return, etc.)

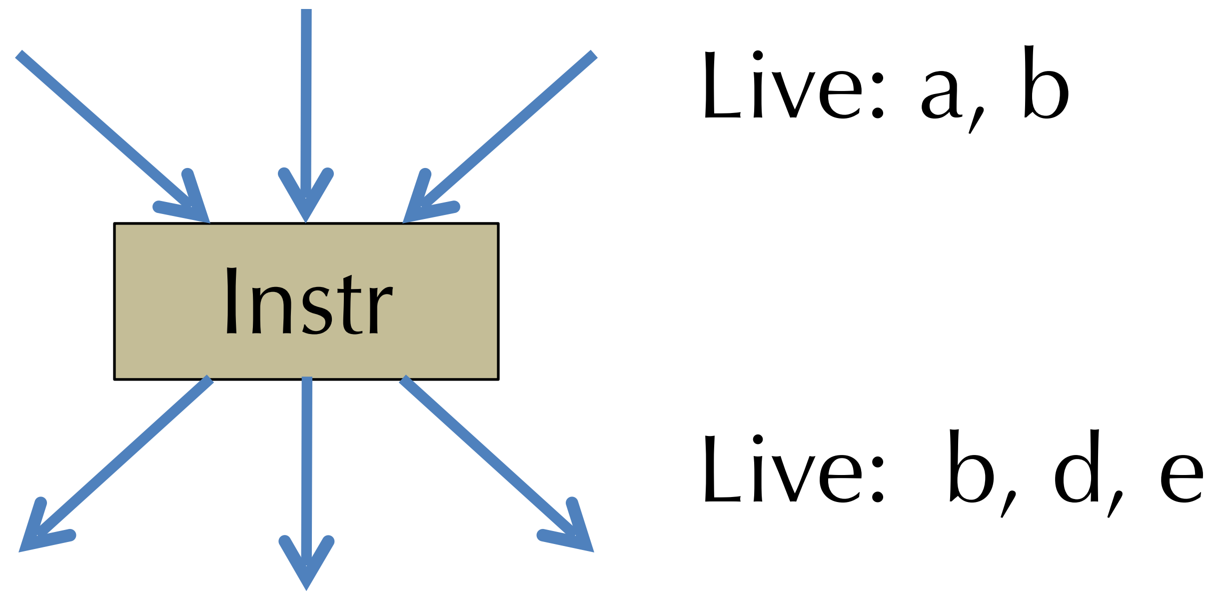
- A *control flow graph*
  - Nodes are blocks
  - There is an edge from B1 to B2 if the control-flow instruction of B1 might jump to the entry label of B2
  - There are no "dangling" edges – there is a block for every jump target.

Dataflow over CFGs

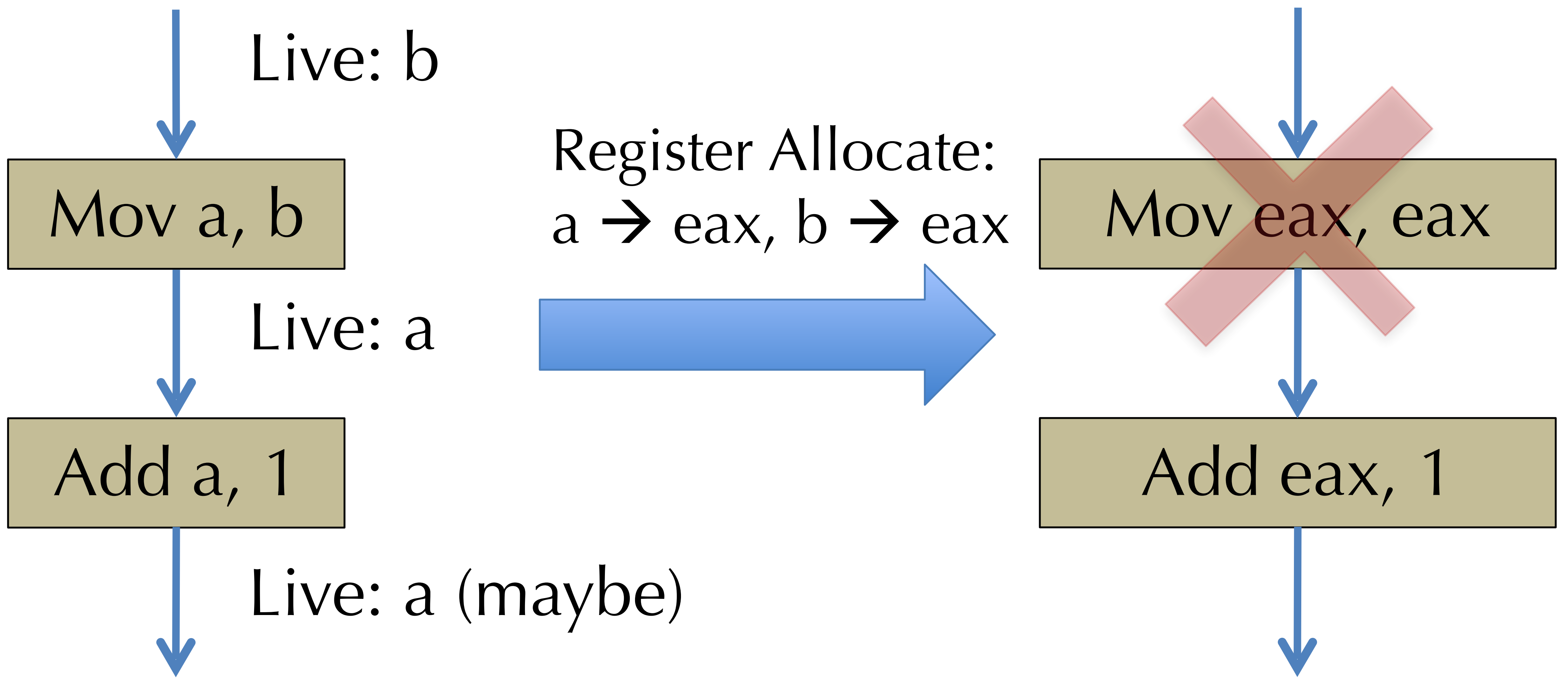
- For precision, it is helpful to think of the "fall through" between sequential instructions as an edge of the control-flow graph too.
  - Different implementation tradeoffs in practice…



Liveness is Associated with *Edges*



- This is useful so that the same register can be used for different temporaries in the same statement.
- Example: a = b + 1
- Compiles to:



Next Lecture

- Liveness analysis, formally
- Other Dataflow Analyses
- A general algebraic framework for defining DFAs