中山大學
SUN YAT-SEN UNIVERSITY

国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Compilation Principle
编 译 原 理

第20讲：代码优化(2)

张献伟

xianweiz.github.io

DCS290, 5/26/2022

Quiz Questions

- Q1: why have the phase of Intermediate Code Gen?

  AST → IR, both language- and machine independent. Easy to apply common optimizations and to extend.

- Q2: TAC of A[i][j], type(A) = array(20, array(10, double))?

  Addr(A[i][j]) = base + i * 80 + j * 8

  $t_1 = i * 80$; $t_2 = j * 8$; $t_3 = t_1 + t_2$; $t_4 = A[t_3]$

- Q3: is the code SSA? If not, convert it.

  No. x is assigned more than once.

  $x_1 = a + b$; if $x_1 > 5$: $x_2 = c$; $y_1 = PHI(x_1, x_2)*2$;

```
x = a + b;
if x > 5: x = c;
y = x * 2;
```

- Q4: for the IR of S -> if (B) S_1 else S_2, where to place 'goto S.next'?

  S_1.code {goto S.next} else S_2.code: skip S_2 after executing S_1.

- Q5: usage of backpatching?

  One-pass code gen, backpatching 'goto _' when target is known.

Control-Flow Analysis[控制流分析]

- The compiling process has done lots of analysis
  - Lexical
  - Syntax
  - Semantic
  - IR
- But, it still doesn't really know how the program does what it does
- **Control-flow analysis** helps compiler to figure out more info about how the program does its work
  - First construct a **control-flow graph**, which is a graph of the different possible paths program flow could take through a function
    - To build the graph, we first divide the code into basic blocks

Basic Block[基本块]

- A **basic block** is a maximal sequence of instructions that
  - Except the first instruction, there are no other labels[只第一条入]
  - Except the last instruction, there are no jumps[只末一条出]
- Therefore, [进/出口唯一]
  - Can only jump into the beginning of a block
  - Can only jump out at the end of a block
- Are units of control flow that cannot be divided further
  - All instructions in basic block execute or none at all[all or nothing]

- Local optimizations are limited to scope of a basic block
- Global optimizations are across basic blocks

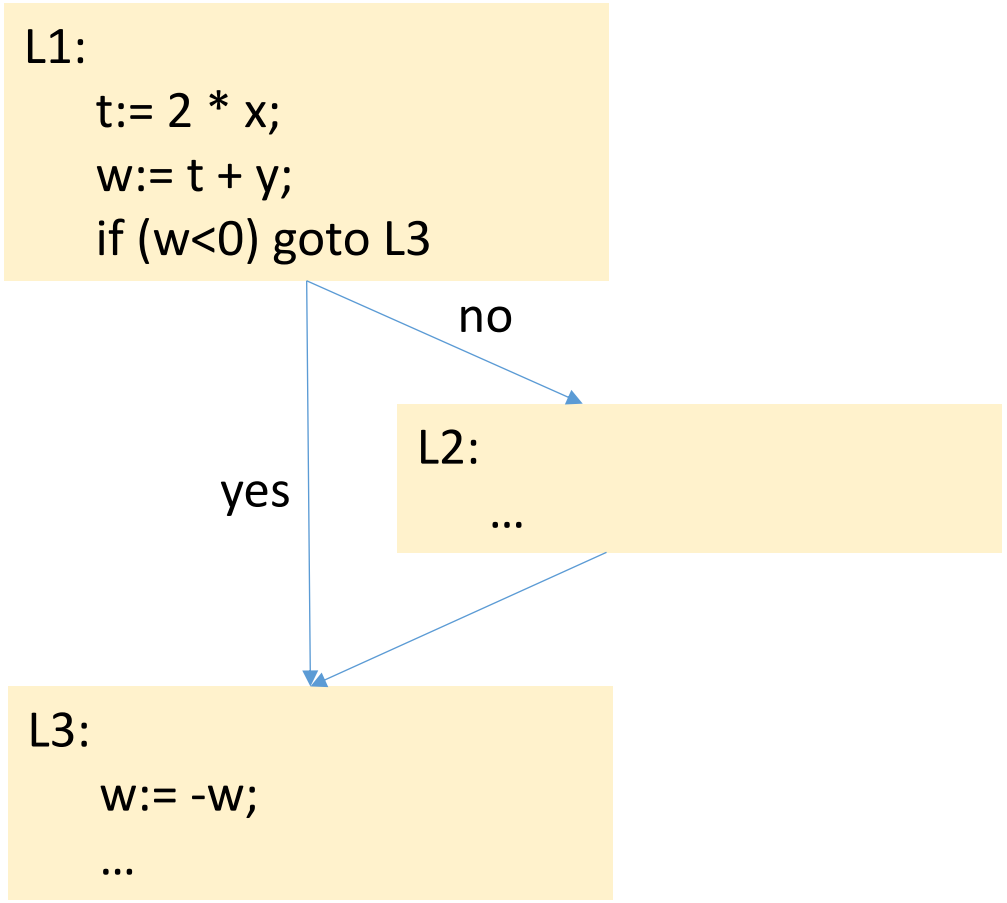
Control Flow Graph[控制流图]

- A **control flow graph** is a directed graph in which
  - Nodes are basic blocks
  - Edges represent flow of execution between basic blocks
    - Flow from end of one basic block to beginning of another
    - Flow can be result of a control flow divergence
    - Flow can be result of a control flow merge
  - Control statements introduce control flow edges
    - e.g. if-then-else, for-loop, while-loop, ...

- CFG is widely used to represent a function
- CFG is widely used for program analysis, especially for global analysis/optimization

Example

```
L1:
    t:= 2 * x;
    w:= t + y;
    if (w<0) goto L3
L2:
    ...
L3:
    w:= -w
     ...
```



LLVM CFG

- $clang -emit-llvm -S ../tester/functional/027_if2.sysu.c

```
@a = dso_local global i32 0, align 4

define dso_local i32 @main() {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 10, i32* @a, align 4
  %2 = load i32, i32* @a, align 4
  %3 = icmp sgt i32 %2, 0
  br i1 %3, label %4, label %5

4:
  store i32 1, i32* %1, align 4
  br label %6

5:
  store i32 0, i32* %1, align 4
  br label %6

6:
  %7 = load i32, i32* %1, align 4
  ret i32 %7
}
```

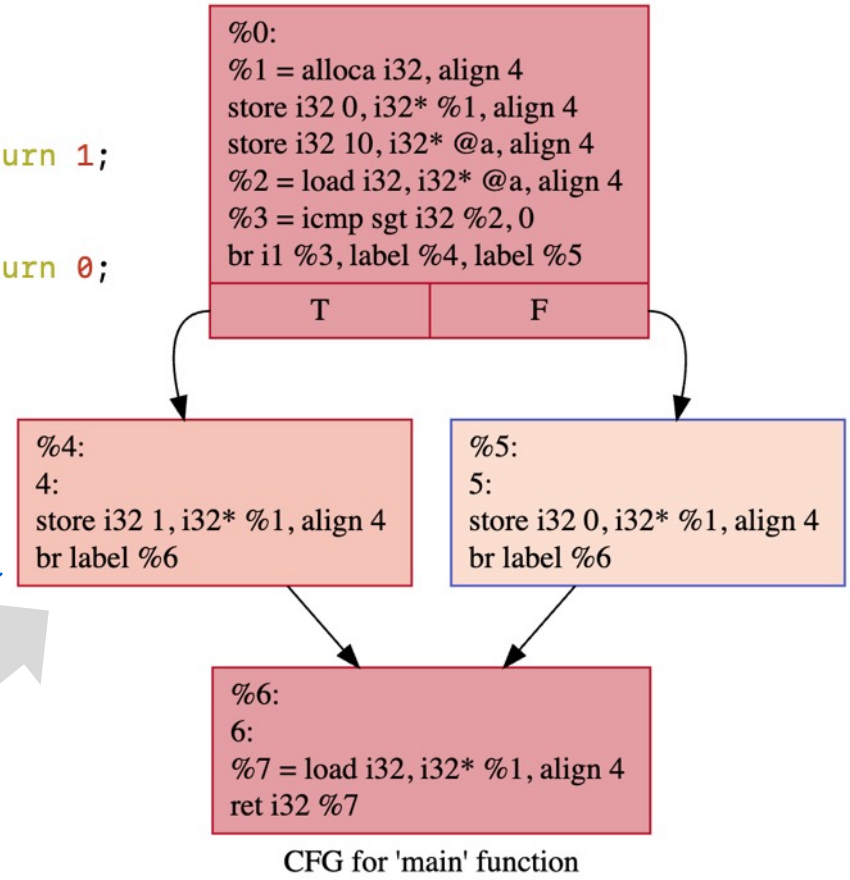
```
1  int a;
2  int main(){
3          a = 10;
4          if( a>0 ){
5                  return 1;
6          }
7          else{
8                  return 0;
9          }
10 }
```

$opt -dot-cfg 027_if2.sysu.ll [→ .main.dot]

```
digraph "CFG for 'main' function" {
        label="CFG for 'main' function";

        Node0x2a784a90 [shape=record,color="#b70d28ff", style=filled, fillcolor="#b
70d2870",label="{%0:\l  %1 = alloca i32, align 4\l  store i32 0, i32* %1, align 4\l
  store i32 10, i32* @a, align 4\l  %2 = load i32, i32* @a, align 4\l  %3 = icmp sg
t i32 %2, 0\l  br i1 %3, label %4, label %5\l|{<s0>T|<s1>F}}"];
        Node0x2a784a90:s0 -> Node0x2a784c70;
        Node0x2a784a90:s1 -> Node0x2a784cc0;
        Node0x2a784c70 [shape=record,color="#b70d28ff", style=filled, fillcolor="#e
8765c70",label="{%4:\l4:                                                \l  store i
32 1, i32* %1, align 4\l  br label %6\l}"];
        Node0x2a784c70 -> Node0x2a784e50;
        Node0x2a784cc0 [shape=record,color="#3d50c3ff", style=filled, fillcolor="#f
7b39670",label="{%5:\l5:                                                \l  store i
32 0, i32* %1, align 4\l  br label %6\l}"];
        Node0x2a784cc0 -> Node0x2a784e50;
        Node0x2a784e50 [shape=record,color="#b70d28ff", style=filled, fillcolor="#b
70d2870",label="{%6:\l6:                                                \l  %7 = lo
ad i32, i32* %1, align 4\l  ret i32 %7\l}"];
}
```

http://viz-js.com/



CFG for 'main' function

Construct CFG

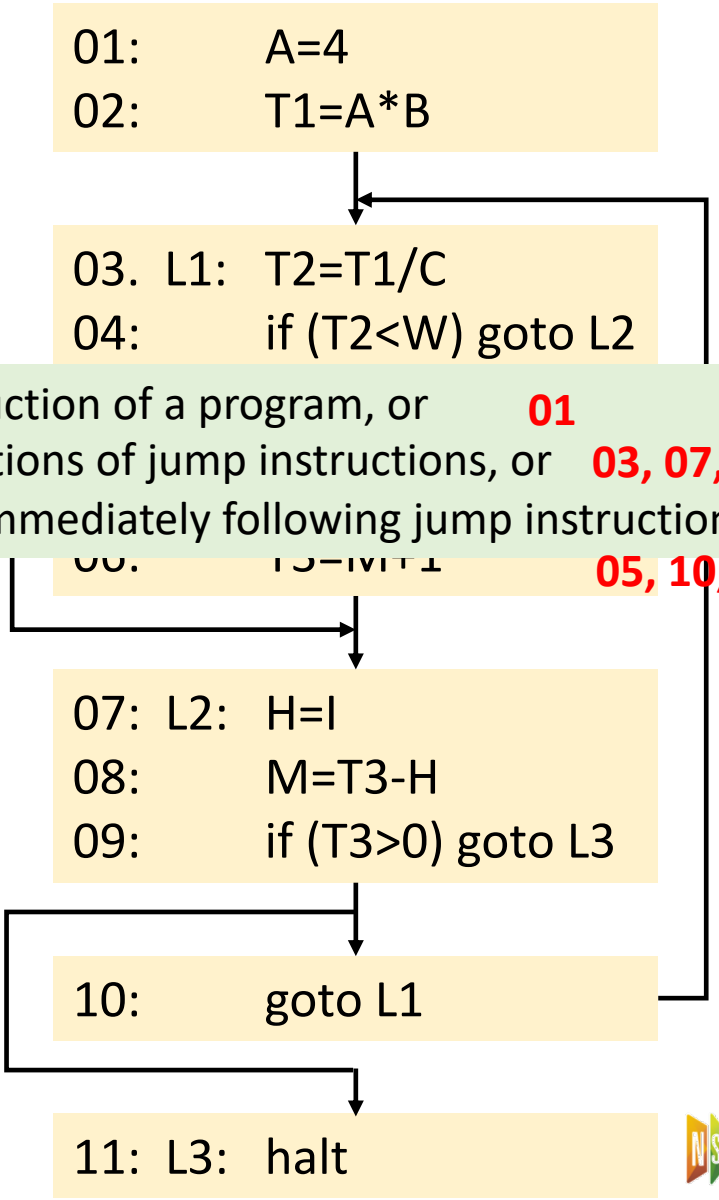
- Step 1: partition code into basic blocks[分解为基本块]
  - Identify **leader** instructions that are
    - the first instruction of a program, or[首条指令]
    - target instructions of jump instructions, or[跳转目标]
    - instructions immediately following jump instructions[紧跟跳转]
  - A basic block consists of a leader instruction and subsequent instructions before the next leader
- Step 2: add an edge between basic blocks B1 and B2 if[连接基本块]
  - B2 follows B1, and B1 may "fall through" to B2[相邻]
    - B1 ends with a conditional jump to another basic block[若条件假，到达B2]
    - B1 ends with a non-jump instruction (B2 is a target of a jump)[无跳转，B1顺序执行到达B2]
    - Note: if B1 ends in an unconditional jump, cannot fall through[B1无条件跳转，会绕开B2]
  - B2 doesn't follow B1, but B1 ends with a jump to B2[不相邻，但B2是B1的跳转目标]

Example

- Partition code into basic blocks
  - Identify leader instructions
- Add edges between basic blocks

```
01:        A=4
02:        T1=A*B
03.  L1:   T2=T1/C
04:        if (T2<W) goto L2
05:        M=T1*K
06:        T3=M+1
07:  L2:   H=I
08:        M=T3-H
09:        if (T3>0) goto L3
10:        goto L1
11:  L3:   halt
```

- the first instruction of a program, or **01**
- target instructions of jump instructions, or **03, 07, 11**
- instructions immediately following jump instructions **05, 10, 11**



Local and Global Optimizations

- Local optimizations[局部优化]
  - Optimizations performed exclusively within a basic block
  - Typically the easiest, never consider any control flow info
    - All instructions in scope executed exactly once
  - Examples:
    - constant folding[常量折叠]
    - common subexpression elimination[删除公共子表达式]

- Global optimizations[全局优化]
  - Optimizations performed across basic blocks
    - Scope can contain if / while / for statements
    - Some insts may not execute, or even execute multiple times
  - Note: global here doesn't mean across the entire program
    - We usually optimize one function at a time

Local Optimization: Examples

- Common subexpression elimination[删除公共子表达式]
  - Two operations are common if they produce the same result
    - It is likely more efficient to compute the result once and reference it the second time rather than re-evaluate it[避免重复计算]
- Dead code elimination[删除无用代码]
  - If an instruction's result is never used, the instruction is considered "dead" and can be removed from the instruction stream[结果从不使用]

```
y = x + z;
y = x * x  + (x/3)
z = x * x + y;
```

```
y = x + z;
t1 = x * x
t2 = x / 3
y = t1  + t2
t3 = x * x
z = t3 + y;
```

```
y = x + z;
t1 = x * x
t2 = x / 3
y = t1  + t2
t3 = x * x
z = t1 + y;
```

DAG of Basic Blocks

- Many important techniques for local optimization begin by transforming a BB into a DAG (directed acyclic graph)[无环有向图]

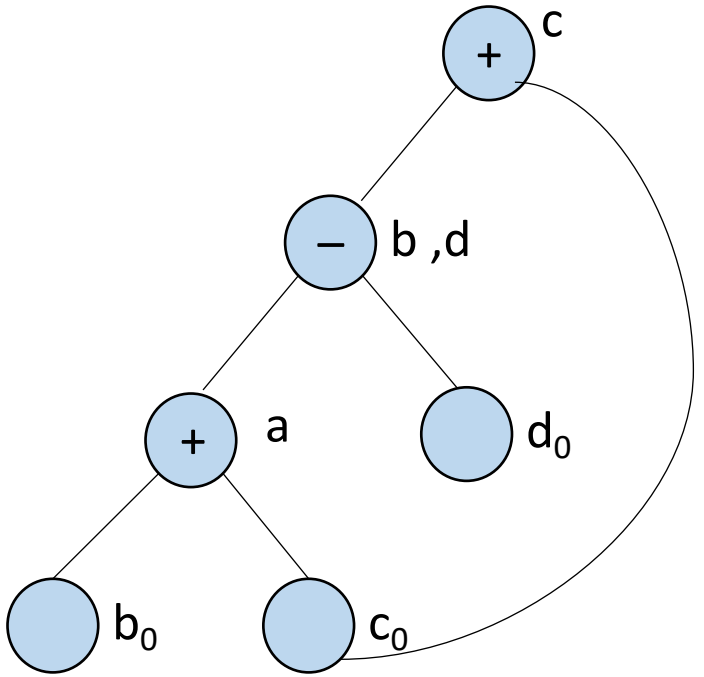
- To construct a DAG for a BB as follows
  - Create a node for each of the initial values of the variables appearing in the BB[为变量初始值创建节点，叶子]
  - Create a node *N* associated with each statement *s* within the block[为声明语句创建节点，中间]
    - The children of *N* are those nodes corresponding to statements that are the last definitions, prior to *s*, of the operands used by *s*
    - Label *N* by the operator applied at *s*[用运算符标注节点]
  - Certain nodes are designated output nodes[某些为输出节点]
    - These are the nodes whose variables are *live on exit* from the block (i.e., their values may be used later, in another block of the flow graph)

Example: DAG

- (3) c = b + c
  - *b* refers to the node labelled '-'
    - Most recent definition of *b*

(1) a = b + c
(2) b = a – d
(3) c = b + c
(4) d = a – d

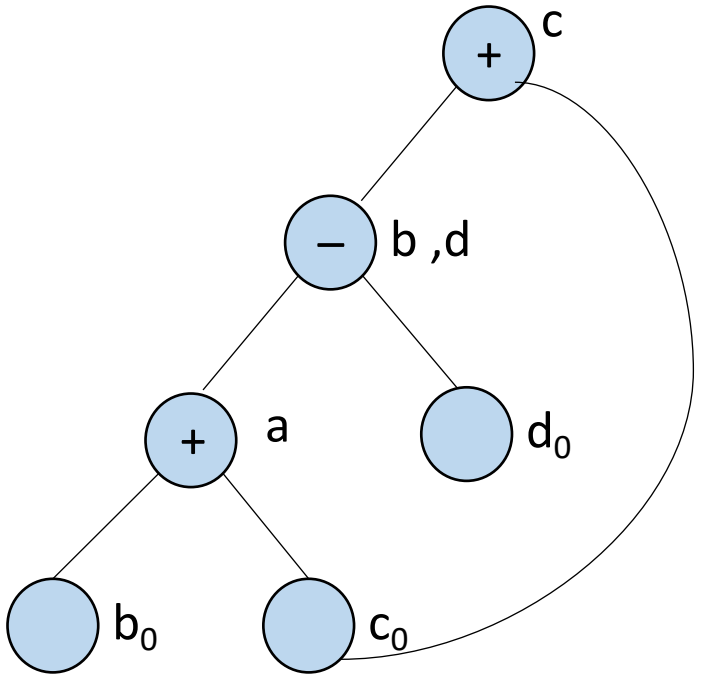
- (4) d = a – d
  - Operator and children are the same as the 2nd statement
    - Reuse the node



Local Opt.: Elimination

- If *b* is not live on exit from the block
  - No need to keep *b = a – d*
- If both *b* and *d* are live
  - Remove either (2) or (4) : **common subexpr elimination**
  - Add a 4th statement to copy one to the other
- If only *a* is live on exit
  - Then remove nodes from the DAG correspond to dead code
    - c -> b,d -> d_0
  - This is actually **dead code elimination**

(1) a = b + c
(2) b = a – d
(3) c = b + c
(4) d = a – d

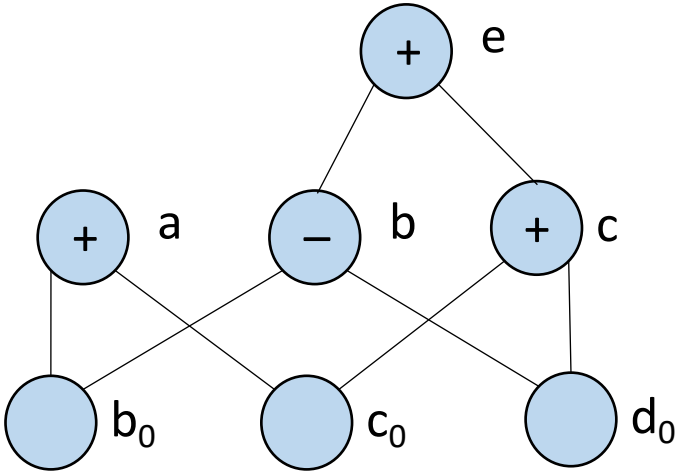


Local Opt.: Elimination (cont.)

- When finding common subexprs, we really are finding exprs that are guaranteed to compute the same value, no matter how that value is computed[过于严苛]
  - Thus miss the fact that (1) and (4) are the same
    - $b + c = (b - d) + (c + d) = b_0 + c_0$

(1) a = b + c
(2) b = b – d
(3) c = c + d
(4) e = b + c

- **Solution**: algebraic identities[代数恒等式]



Local Opt.: Algebraic Identities[代数恒等式]

- Eliminate computations by applying mathematical rules[使用数学规则]
  - Identities: a * 1 ≡ a, a * 0 ≡ 0, b & true ≡ b
  - Reassociation and commutativity[重组合、交换]
    - (a + b) + c ≡ a + (b + c) , a + b ≡ b + a
- **Strength Reduction**[强度削减]
  - Replacing expensive operations (*multiplication*, *division*) by less expensive operations (*add*, *sub*, *shift*)
  - Some ops can be replaced with cheaper ops
  - Examples
    - x=y/8 --> x=y»3
    - y=y*8 --> x=y«3
    - x^2 --> x * x
    - 2 * x --> x + x

Local Opt.: Constant Folding[常量折叠]

- **Constant Folding**
  - Computing operations on constants at compile time
  - Example:

```
#define LEN 100
x = 2 * LEN;
if (LEN < 0) print(”error”);
```

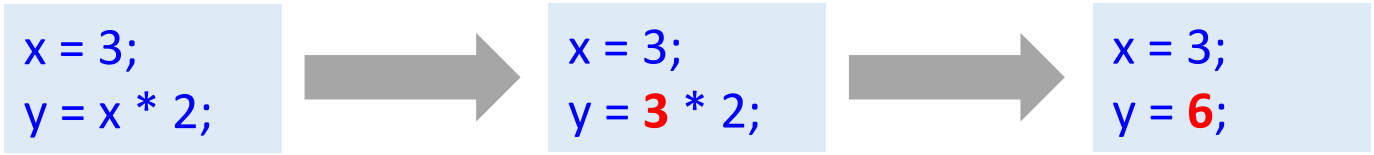
  - After constant folding

```
x = 200;
if (false) print(”error”);
```

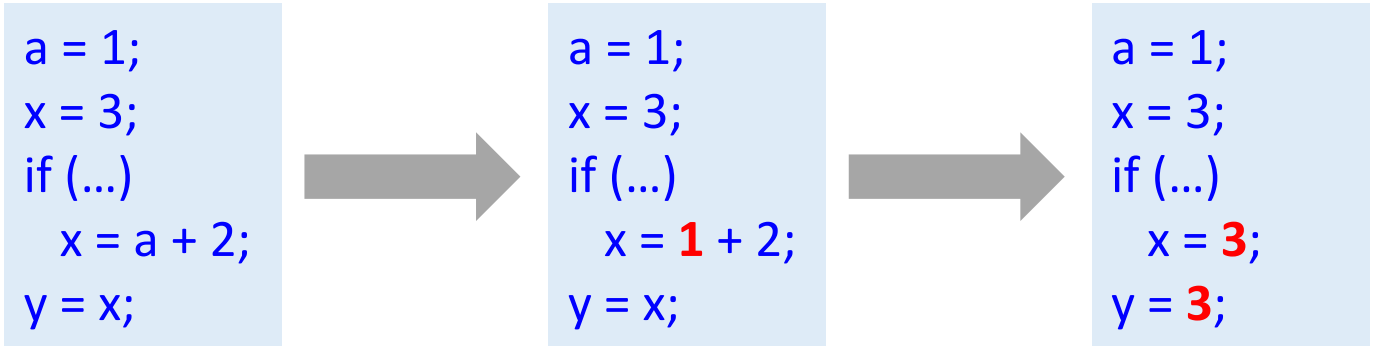
  - Dead code elimination can further remove the above *if* statement
  - Inherently local since scope limited to statement

Local Opt.: Constant Propagation[常量传播]

- **Constant Propagation**
  - Substituting values of known constants at compile time
  - Local Constant Propagation (LCP)



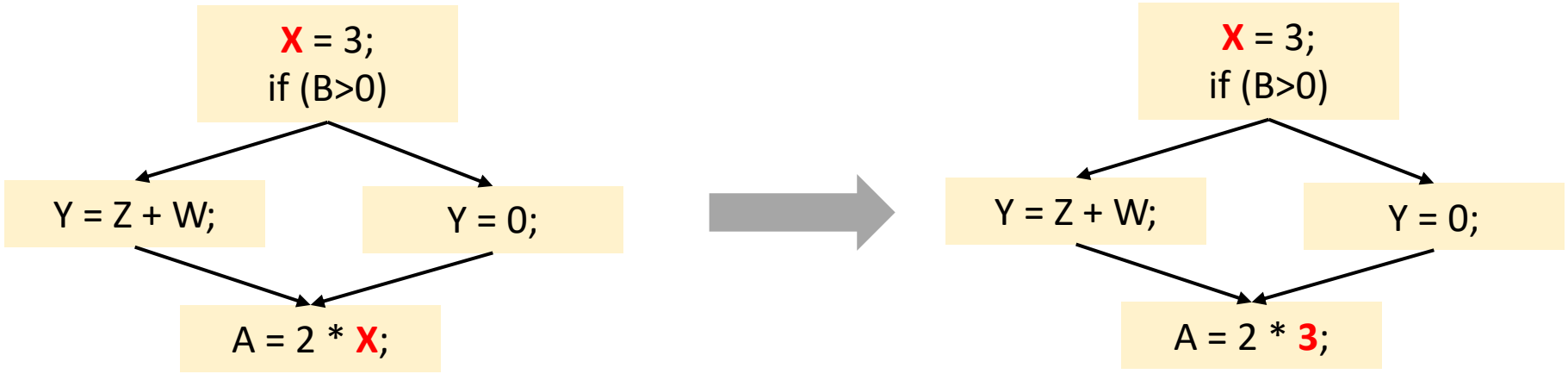
- Some optimizations have both local and global versions
  - Global Constant Propagation (GCP)

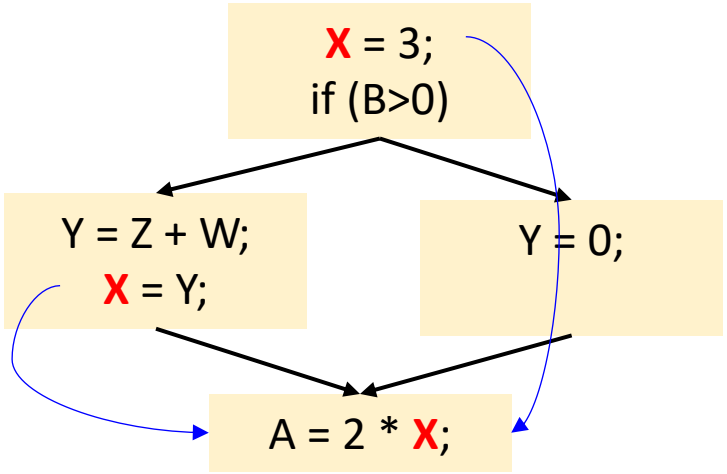


  - GCP more powerful than LCP but also more complicated
    - Must determine *x* is constant across all paths reaching *x*

Global Optimizations

- Extend optimizations to flow of control, i.e. CFG
  - Along **all paths**, the last assignment to X is "X=C"
  - Optimization must be stopped if incorrect in even one path





Global Opt.: Conservative[需保守]

- Compiler must prove some property X at a particular point
  - Need to prove at that point property X holds along all paths
  - Need to be **conservative** to ensure correctness
    - An optimization is enabled only when X is definitely true
    - If not sure if it is true or not, it is safe to say **don't know**
    - If analysis result is **don't know**, no optimization done
    - May lose opt. opportunities but guarantees correctness
- Property X often involves data flow of program
  - E.g. Global Constant Propagation (GCP):

```
X = 7;
...
Y = X + 3; // Does value of 7 flow into this use of X?
```

  - Needs knowledge of **data flow**, as well as control flow
    - Whether data flow is interrupted between points A and B

Global Opt.: Data Flow[数据流]

- Most optimizations rely on a property at given point
  - For Global Constant Propagation (GCP):

    A = B + C; // Property: {A=?, B=10, C=?}
  - After optimization:

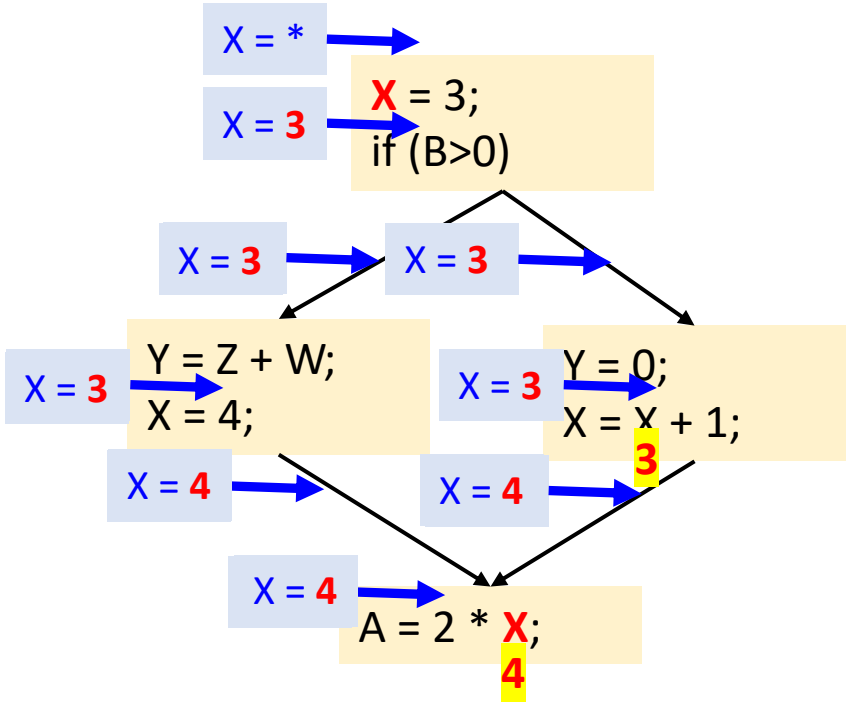
    A = 10 + C;
- For this discussion, let's call these properties ***values***
- **Dataflow analysis**: compiler analysis that calculates values for each point in a program
  - Values get propagated from one statement to the next
  - Statements can modify values (for GCP, assigning to vars)
  - Requires CFG since values flow through control flow edges
- **Dataflow analysis framework**: a framework for dataflow analysis that guarantees correctness for **all paths**
  - Does *not* traverse all possible paths (could be infinite)
  - To be feasible, makes **conservative** approximations

Global Constant Propagation (GCP)

- Let's apply dataflow analysis to compute values for GCP
  - Emulates what human does when tracing through code
- Let's use following notation to express the state of a var:
  - x=*: not assigned (default)
  - x=1, x=2, ...: assigned to a constant value
  - x=#: assigned to multiple values
- All values start as x=* and are iteratively refined
  - Until they stabilize and reach a fixed point
- Once fixed point is reached, can replace with constants:
  - x=*: replace with any constant (typically 0)
  - x=1, x=2, ...: replace with given constant value
  - x=#: cannot do anything

Example

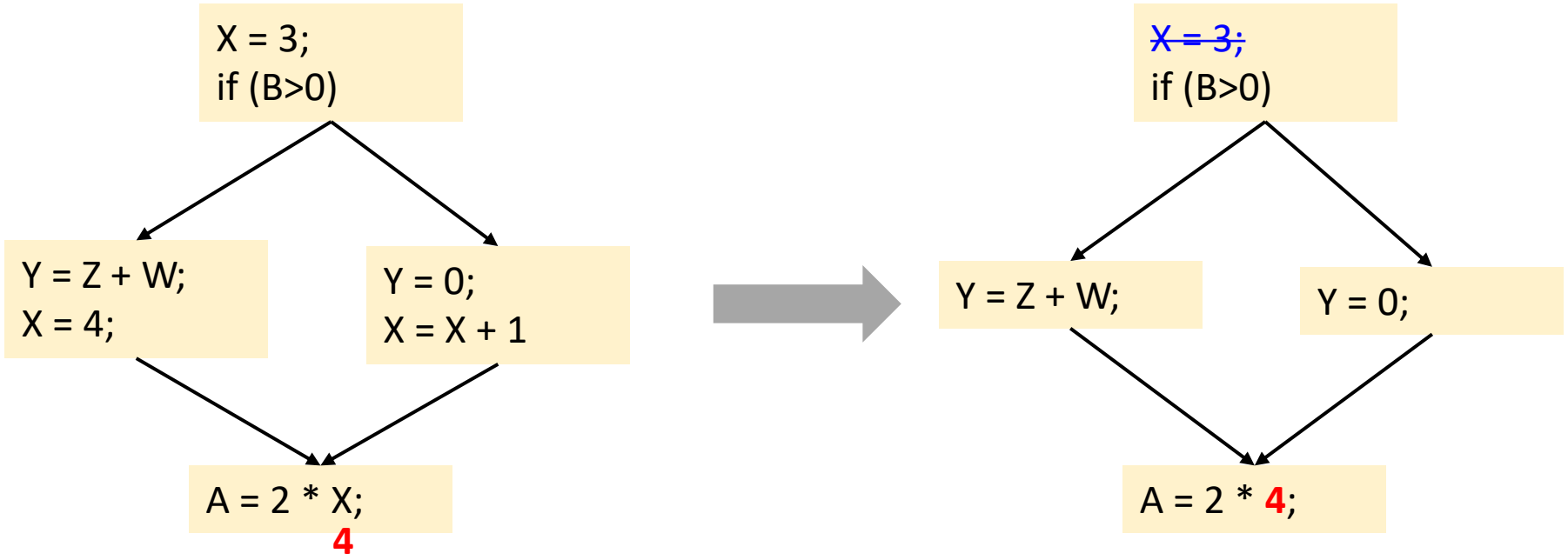
- In this example, constants can be propagated to **X+1**, **2*X**
- Statements visited in reverse postorder (predecessor first)



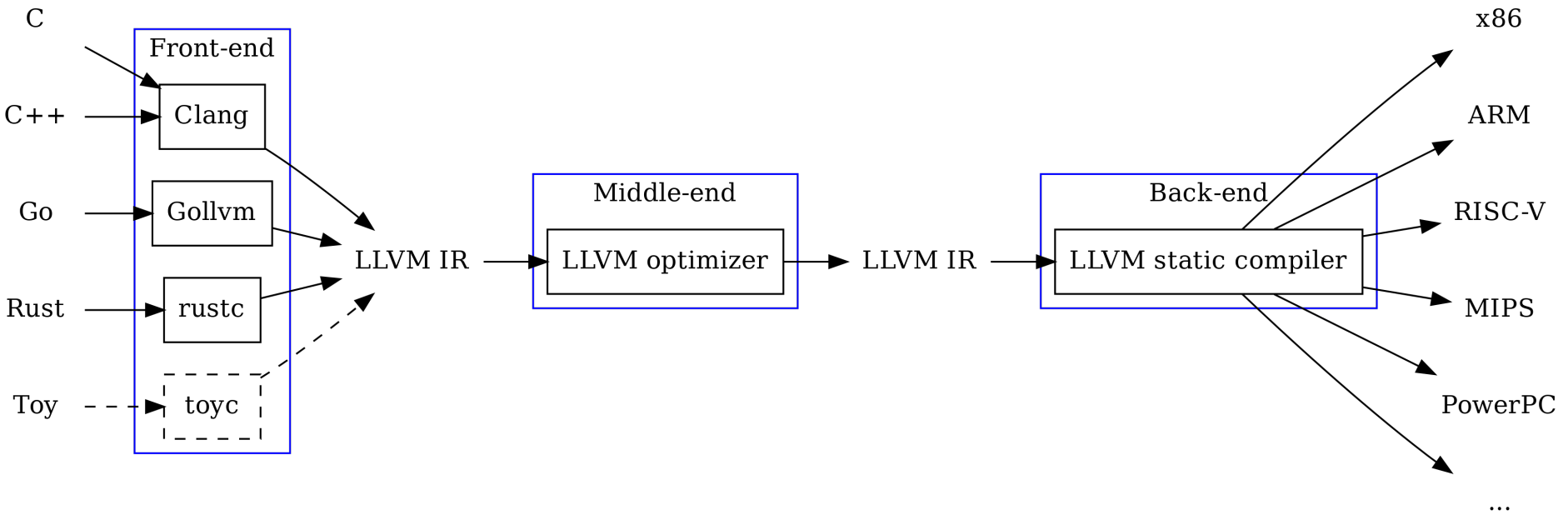
x=*: not assigned (default)
x=1, x=2, ...: assigned to a constant value
x=#: assigned to multiple values

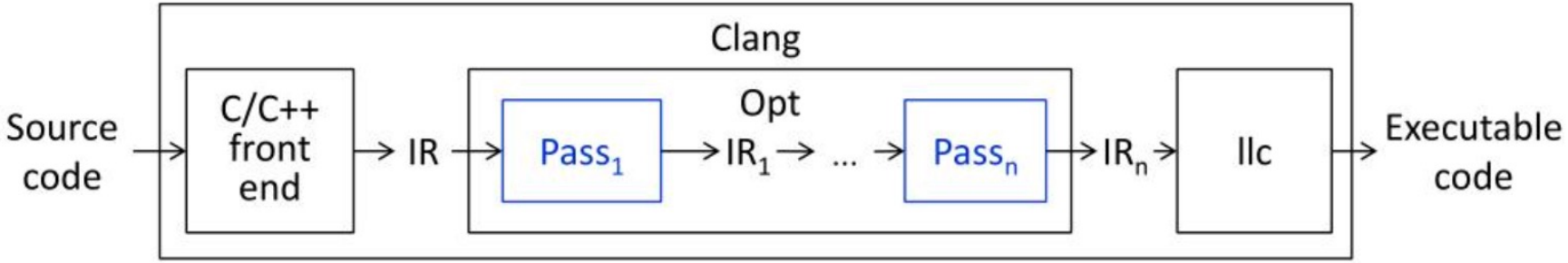
Example (cont.)

- Once constants have been globally propagated, we would like to eliminate the dead code



IR Optimization of LLVM





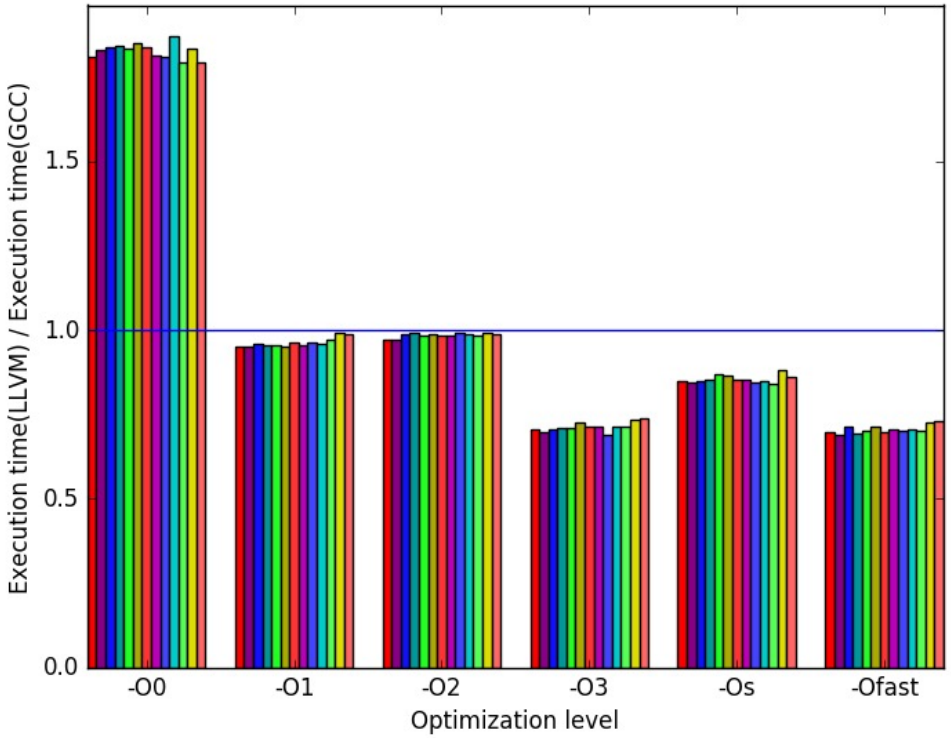
https://www.slideserve.com/quinlan-dominguez/llvm-pass-and-code-instrumentation

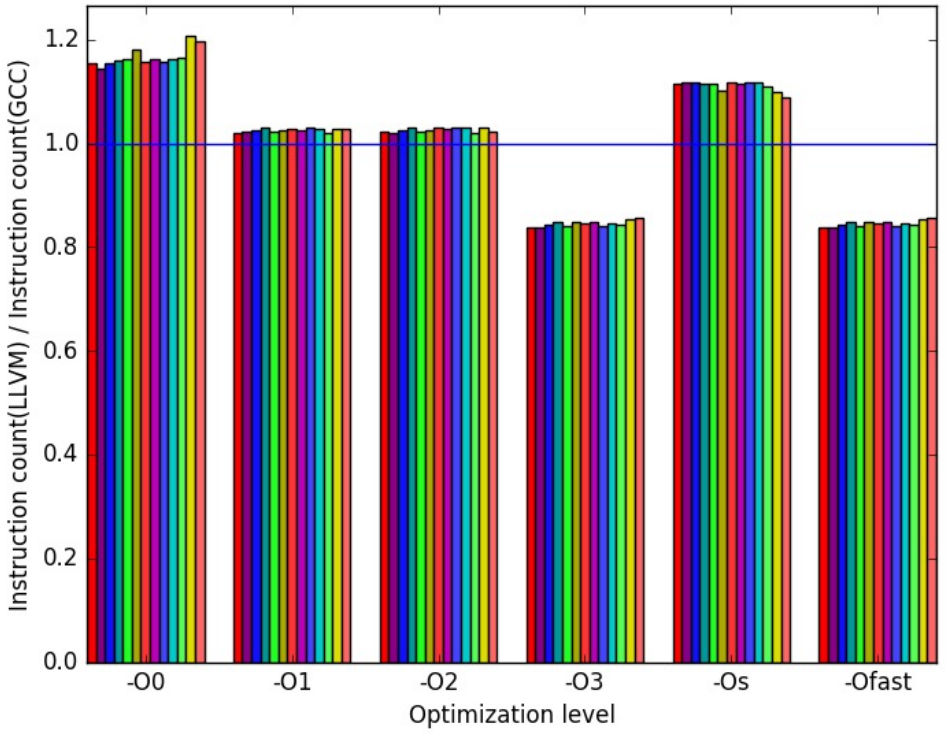
LLVM Optimization Flags

- O0: no optimization
  - Compiles the fastest and generates the most debuggable code
- O1: somewhere between O0 and O2
- O2: moderate level of optimization enabling most optimizations
- O3: like O2,
  - except that it enables opts that take longer to perform or that may generate larger code (in an attempt to make the program run faster)
- Os: like O2 with exta opts to reduce code size
- Oz: like Os, but reduce code size further
- O4: enables link-time opt Clang has support for O4, but not opt

Performance at Varying Flags

- Compare the performance of the benchmark when compiled with either GCC or LLVM
  - Compile benchmark at six optimization levels
  - Each workload was run 3 times with each executable on the Intel Core i7-2600 machines

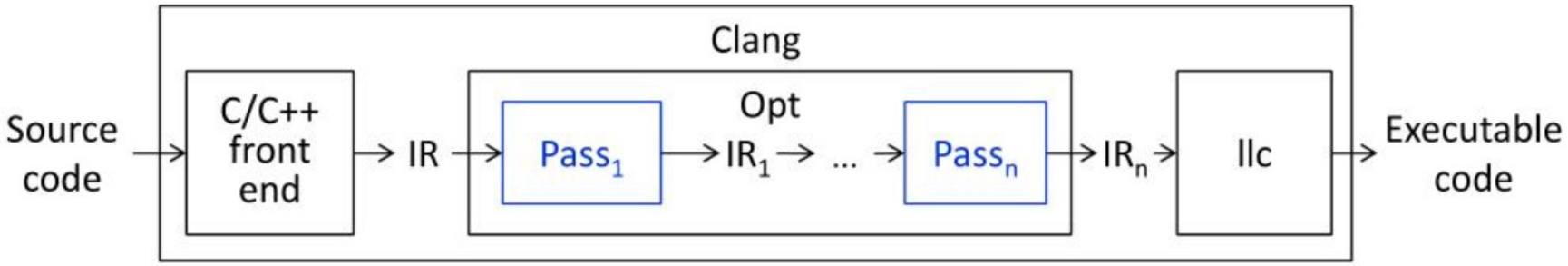




https://webdocs.cs.ualberta.ca/~amaral/AlbertaWorkloadsForSPECCPU2017/reports/exchange2_report.html#x1-12003r1

LLVM Passes

- Optimizations are implemented as **Passes** that traverse some portion of a program to either collect information or transform the program
- A Pass receives an LLVM IR and performs analyses and/or transformations
  - Using opt, it is possible to run each Pass
- A Pass can be executed in a middle of compiling process from source code to binary code
  - The pipeline of Passes is arranged by Pass Manager



LLVM Passes (cont.)

- **Analysis** passes: compute info that other passes can use or for debugging or program visualization purposes
  - -memdep: Memory Dependence Analysis
  - -print-function: Print function to stderr
  - …
- **Transform** passes: can use (or invalidate) the analysis passes, all mutating the program in some way
  - -dce: Dead Code Elimination
  - -loop-unroll: Unroll loops
  - …
- **Utility** passes: provides some utility but don't otherwise fit categorization
  - -view-cfg: View CFG of function
  - …

Example

- $clang -emit-llvm -S sum.c

```
int sum(int a, int b) {
  return a + b;
}
```

- $opt sum.ll -debug-pass=Structure -mem2reg -S -o sum-O1.ll

```
Pass Arguments:  -targetlibinfo -tti -targetpassconfig -assumption-cache-tracker -domtree -mem2reg -verify -print-module
Target Library Information
Target Transform Information
Target Pass Configuration
Assumption Cache Tracker
  ModulePass Manager
    FunctionPass Manager
      Dominator Tree Construction
      Promote Memory to Register
      Module Verifier
    Print Module IR
```

$opt sum.ll -debug-pass=Structure -O1 -S -o sum-O1.ll
$opt sum.ll -time-passes -O1 -o sum-tim.ll

- $opt sum.ll -time-passes -mem2reg -o sum-tim.ll

```
===-------------------------------------------------------------------------===
                      ... Pass execution timing report ...
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0003 seconds (0.0003 wall clock)

   ---User Time---   --System Time--   --User+System--   ---Wall Time---  --- Name ---
   0.0002 ( 91.1%)   0.0001 ( 90.2%)   0.0003 ( 90.8%)   0.0003 ( 90.6%)  Bitcode Writer
   0.0000 (  3.7%)   0.0000 (  4.5%)   0.0000 (  4.0%)   0.0000 (  3.7%)  Module Verifier
   0.0000 (  2.3%)   0.0000 (  2.3%)   0.0000 (  2.3%)   0.0000 (  2.8%)  Dominator Tree Construction
   0.0000 (  2.3%)   0.0000 (  2.3%)   0.0000 (  2.3%)   0.0000 (  2.4%)  Promote Memory to Register
   0.0000 (  0.5%)   0.0000 (  0.8%)   0.0000 (  0.6%)   0.0000 (  0.6%)  Assumption Cache Tracker
   0.0002 (100.0%)   0.0001 (100.0%)   0.0003 (100.0%)   0.0003 (100.0%)  Total

===-------------------------------------------------------------------------===
                                LLVM IR Parsing
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0006 seconds (0.0006 wall clock)
```