

- Motivation
- Overview of optimizing compiler internals
- Control flow analysis
- Scalar optimizations on SSA Form
- Register allocation
- Instruction scheduling
- Vectorization

# The Compiler is the Programmer's Most Important Tool

- The programmer with knowledge about optimizing compilers knows
  - what the compiler can optimize faster and better than himself/herself, and
  - compilers' limitations and how to write code that helps them to do better automatic optimization.
- The competent programmer focuses on writing code which is
  - correct,
  - efficient, and
  - easy to maintain.
- Using optimizing compilers improves programmer productivity.
- If your engineers spend 1000 programmer hours to improve the performance by one percent, management should consider getting better compilers! (the example is from a real product and optimization is so time-consuming because it has been optimized for many years)

# Use Different Compilers!

*Using different compilers is more likely to expose:*

- bugs which happened to go undetected with one compiler
- non-portable code — which depends on
  - unspecified behavior (e.g. evaluation order of parameters)
  - implementation-defined behavior (e.g. sizes of integer types)
- nonstandard code — e.g. both IBM XL and GCC have errors in how they implement `extern inline`:
  - "pure macro" by GCC
  - "weak symbol" by IBM XL
  - "external definition" according to the C99 Standard

# Overview of the Internals of an Optimizing Compiler

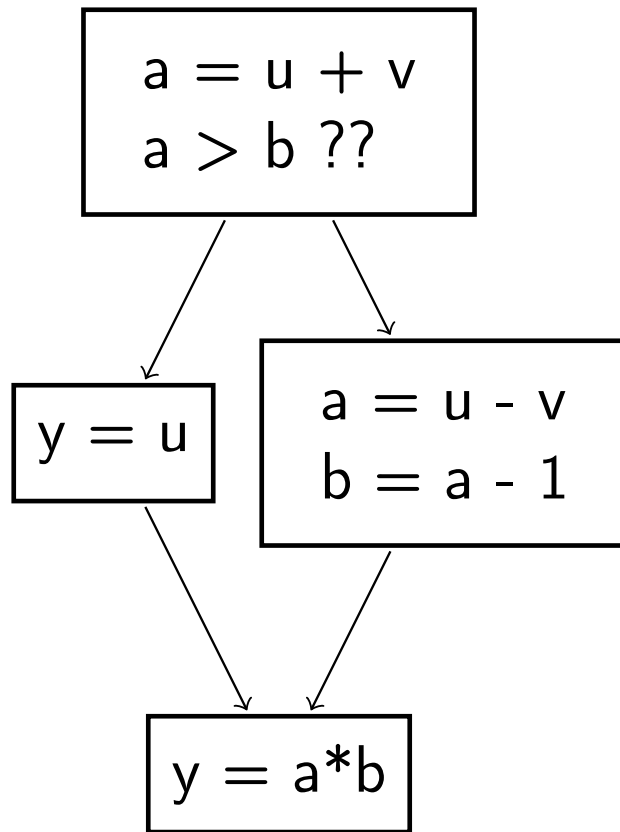
- Lexical, syntactic and semantic analysis: output is an abstract syntax tree (AST).
- Code generation: rewrites the AST to three-address code — similar to assembler for a generic RISC architecture
- Control flow analysis: represents a function as a directed graph of straight line code
- Initial optimizations such as constant propagation
- High-order transformations: vectorization, parallelization, locality optimization
- Scalar optimizations
- Instruction scheduling and register allocation

- Lexical analysis is often implemented using tools such as flex or lex, or without any tool as normal C functions (also very easy).
- Parsing is often implemented using tools such as bison or yacc.
- Semantic analysis is easily implemented as a normal module of C functions.

# Control-Flow Graph: Example C Code

```
a = u + v;  
if (a > b) {  
    y = u;  
} else {  
    a = u - v;  
    b = a - 1;  
}  
y = a * b;
```

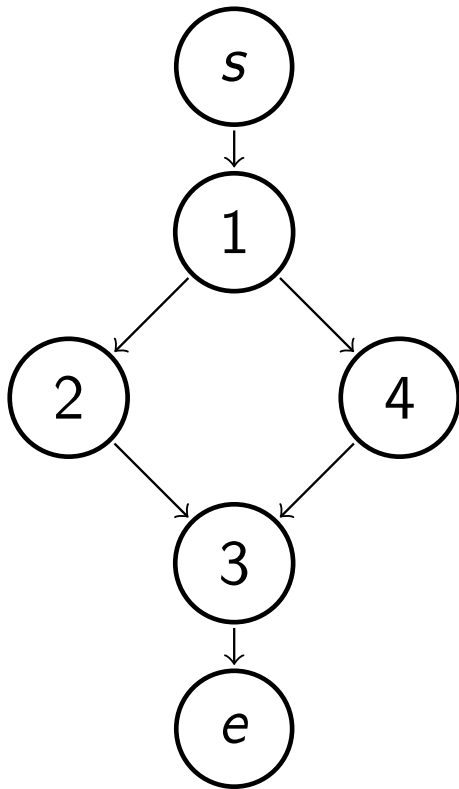
# Control-flow graph: Basic Blocks and Branches



Basic block: sequence of instructions with no label or branch

CFG: directed graph with basic blocks as nodes and branches as edges

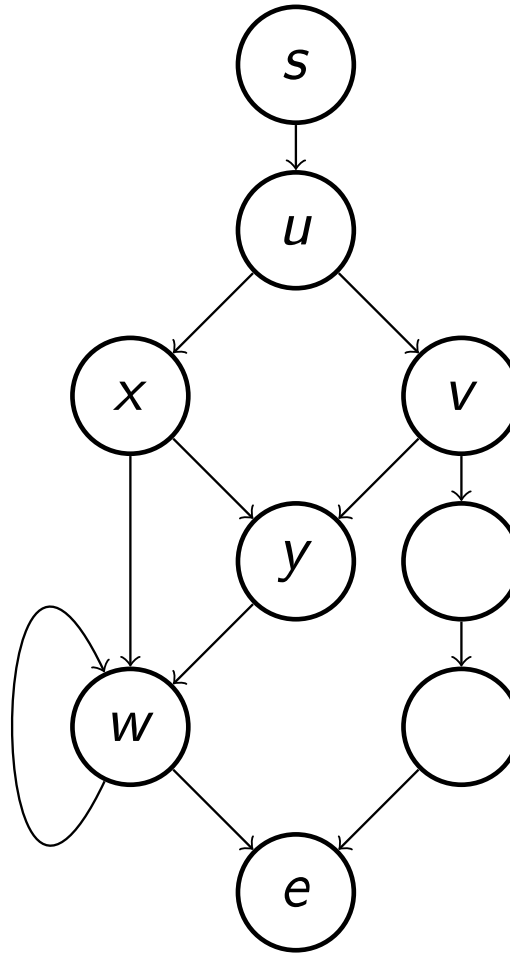
# Control-Flow Graph: the CFG View



Special nodes:

- the first node is called *s* — start
- the last node is called *e* — exit

# Dominance in the CFG



$u$  dominates  $v$  if all paths from  $s$  to  $v$  include  $u$

# Dominance Analysis: Finding who Dominates who

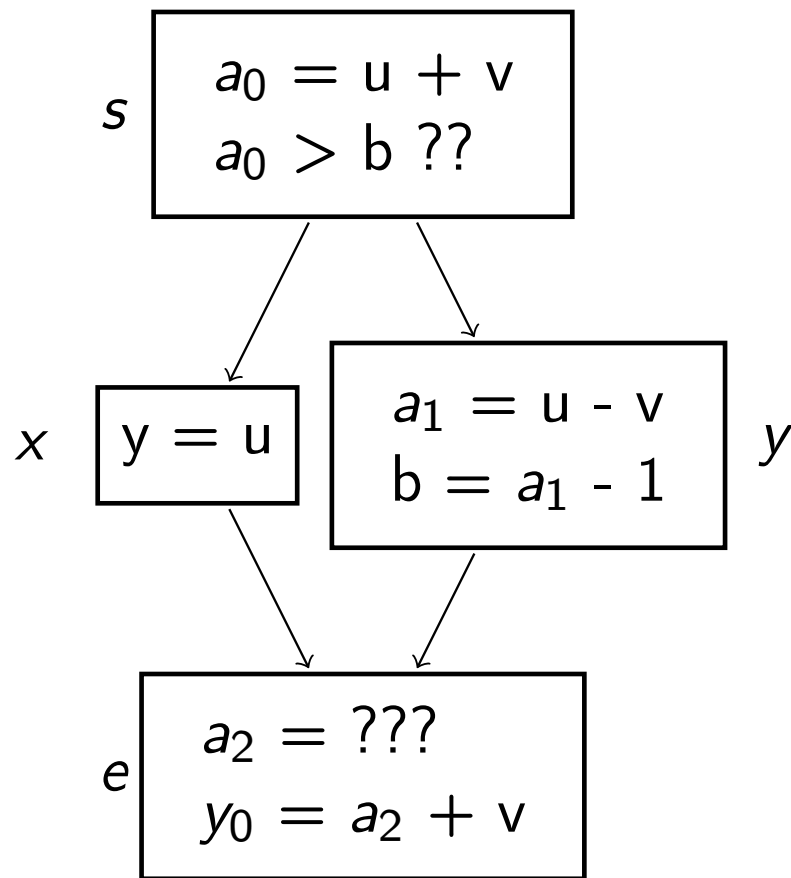
- The fastest algorithm for finding dominators was discovered by Robert Tarjan 1979.

# Static Single Assignment: SSA Form

- A variable is only assigned to by one unique instruction
- That instruction dominates all the uses of the assigned value
- We introduce a new variable name at each assignment
- SSA form is the key to elegant and efficient scalar optimization algorithms
- Invented by IBM Research Yorktown Heights in New York

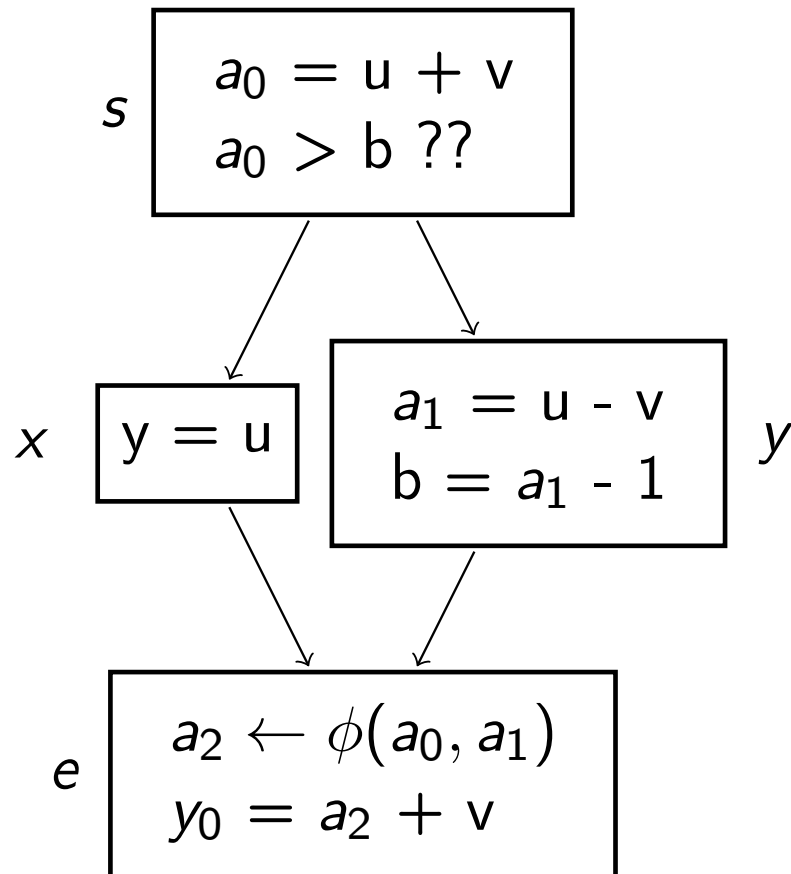
**But what to do when paths from different assignments join???**

# Partial Translation to SSA Form



In node *e*: if we came from node *x* we let  $a_2 \leftarrow a_0$  and if we came from node *y* we let  $a_2 \leftarrow a_1$ . This operation is called the  $\phi$ -function.

# Our Example Translated to SSA Form



# A Function Translated to SSA Form

- We insert a  $\phi$ -function where the paths from two different assignments of the same variable join
- With the  $\phi$ -function, each definition dominates its uses

# Copy Propagation

```
x0 = a0 + b0;
if (...) {
    ...;
}
y0 = x0;      /* COPY */
if (...) {
    ...;
}
c0 = y0 + 1; /* USE */
```

```
x0 = a0 + b0;
if (...) {
    ...;
}
if (...) {
    ...;
}
c0 = x0 + 1;
```

- With SSA form we can know that it is correct to replace  $y_0$  with  $x_0$
- The values of  $x_0$  and  $y_0$  do not change after the definition (in a static sense)

# Hash-Based Value Numbering

## *Useful rules if A is an integer*

$2 * a \Rightarrow a \ll 1$

$a / 2 \Rightarrow a \gg 1$  OK if unsigned integer

$a - a \Rightarrow 0$

$1 * a \Rightarrow a$

$0 * a \Rightarrow 0$

- Shift right is defined in Java to be arithmetic but may be logic in C/C++
- What is the value  $\infty \times 0$  according to IEEE 754 (ie IEC 60559) ?
- Hash-based value numbering is typically implemented as part of the translation to SSA form

# Global Value Numbering (GVN)

```
int h(int a, int b)
{
    int x, y;

    x = 1;
    y = 1;
    do {
        a = a + b;
        x = x + a;
        y = y + a;
    } while (a > 0);
    return x + y;
}
```

```
int h(int a, int b)
{
    int x;

    x = 1;
    do {
        a = a + b;
        x = x + a;
    } while (a > 0);
    return x + x;
}
```

# Common Subexpression Elimination (CSE)

```
int h(int a, int b)
{
    int    c = 1, d = 2;

    if (a > b)
        c = a * b;
    else
        d = a * b;
    return c + a * b;
}
```

```
int h(int a, int b)
{
    int    c = 1, d = 2;
    int    t;
    if (a > b) {
        t = a * b;
        c = t;
    } else {
        t = a * b;
        d = t;
    }
    return c + t;
}
```

# Loop-Invariant Code Motion

```
while (x != y)           ==>   t = a[i];
    x = x + a[i];         while (x != y)
                           x = x + t;
```

```
do                        t = a[i];
    x = x + a[i];         do
while (x != y);           x = x + t;
                           while (x != y);
```

Which transformation above is valid?

# Partial Redundancy Elimination (PRE)

```
a = u + v;  
if (...) {  
    ...;
```

====>

```
} else {  
    a = u - v;  
    x = a * b;  
  
}  
y = a * b;
```

```
a = u + v;  
if (...) {  
    ...;  
    t = a * b;  
} else {  
    a = u - v;  
    t = a * b;  
    x = t;  
  
}  
y = t;
```

# More Partial Redundancy Elimination (PRE)

```
do
    x = x + a / b;
while (x != y);

====>

t = a / b;
do
    x = x + t;
while (x != y);
```

**a/b is partially redundant!**

**PRE can move code out of loops without knowledge about loops**

# Induction Variable Elimination

Also known as Operator strength reduction

```
do {  
    x = x + a[i];  
    i = i + 1;  
} while (i < N);
```

```
do {  
    s = i * 4;  
    t = load a+s;  
    x = x + t;  
    i = i + 1;  
} while (i < N);
```

The primary goal is to get rid of the multiplication

# Basic and Dependent IV

```
do {  
    s = i * 4;  
    t = load a+s;  
    x = x + t;  
    i = i + 1;  
} while (i < N);
```

- $i$  is a *basic* induction variable
- Classes of *dependent* induction variables:  $j \leftarrow b \times i + c$ ,  $i$  är basic IV
- $s \leftarrow 4 \times i + 0$

# Strength Reduction

```
do {  
    s = i * 4;  
    t = load a+s;  
    x = x + t;  
    i = i + 1;  
} while (i < N);
```

```
s = 4 * i;  
do {  
    t = load a+s;  
    x = x + t;  
    i = i + 1;  
    s = s + 4;  
} while (i < N);
```

- Initialize the dependent IV before the loop
- Increment the dependent IV just after the basic IV is incremented
- Maybe we can get rid of the basic IV now?

# Linear Function Test Replacement

```
s = 4 * i;  
do {  
    t = load a+s;  
    x = x + t;  
    i = i + 1;  
    s = s + 4;  
} while (i < N);
```

```
m = 4 * N;  
s = 4 * i;  
do {  
    t = load a+s;  
    x = x + t;  
    s = s + 4;  
} while (s < m);
```

- $s = i \times b + c$  (we have  $b = 4$  and  $c = 0$ )
- $i = \frac{s-c}{b}$
- $i < N \Rightarrow \frac{s-c}{b} < N \Rightarrow s < N \times b + c$ , if  $b > 0$

# Translation Back from SSA Form

- A copy is inserted for each operand of the  $\phi$ -function
- A clever register allocator will put  $a_0$ ,  $a_1$  and  $a_2$  in the same register and remove the COPY

# Register Allocation

- Reading a variable from a register is much faster than reading from memory.
- Usually register allocation is the most important optimization.
- If two variables are used at the same time, they cannot be put the same register.
- Construct an undirected graph with variables as nodes and an edge between two nodes/variables if they are used simultaneously.
- Try to colour the graph with  $K$  colors ( $K$  = number of machine registers).
- NP-complete problem, but there is a good practical solution invented by IBM mathematician Greg Chaitin in 1980 (for the IBM 801 project).
- Essentially all good compilers now use his algorithm.

# Basics of Chaitin's Algorithm

- Remove any node from the graph with degree less than  $K$  and push it on a stack.
- Such a node is guaranteed to be given a colour if the rest of the graph can be colored.
- Suppose you have three colors and a variable with two neighbors. Then surely there will be an unused/available colour when the neighbors have selected their colors.
- So remove nodes with degree less than  $K$  and push them on a stack until the graph is empty, and then pop the nodes one at a time and re-insert the nodes and edges while selecting a colour which no neighbor has already selected.

# Some Questions about Chaitin's Algorithm

- Q1: What should we do if all nodes have degree  $\geq K$  ?
- A1: Then select a not-so-frequently-used variable to live only in memory.
- Q2: What about MOV A,B that we insert when translating from SSA form?
- A2: Try to assign A and B the same colour so that the MOV can be removed.
- Q3: What about function call arguments passed in pre-specified registers?
- A3: Assume variable X should be an argument in R3. Add a MOV X,R3 statement and then apply A2.

# List Scheduling: within one Basic Block

- Create a data dependence graph between the instructions.
- An edge from a producer to a consumer of a value. TRUE
- An edge from a producer to a later producer of the same variable. OUTPUT
- An edge from a consumer to a later producer of the same variable. ANTI
- Perform a topological sort of the graph, ie schedule any instruction with no predecessor in the graph.
- The goal is to reduce the total time to execute the basic block.

# Software Pipelining: Modulo Scheduling

- Normally, one loop iteration is executed to completion before the next is started.
- In software pipelining the next iteration is started  $\Pi$  ( $\Pi =$  initiation interval) cycles after the current, without (1) violating data dependencies or (2) using more hardware resources than are available (eg issue slots, functional units).
- One iteration is scheduled using list scheduling, and hardware resources are checked modulo  $\Pi$ , and data dependencies are also checked with respect to  $\Pi$ .
- If a valid schedule with  $\Pi$  is not found,  $\Pi$  is incremented and a new schedule is tried.
- Modulo scheduling can often give a speedup of 2-3 in numerical codes, but it does increase the register pressure, since each concurrent iteration needs its registers.

# Modulo Scheduling Example

```
for (i = 0; i < N; i++) { | A0
    A | B0 A1
    B | for (i = 2; i < N; i += 3) {
    C |     C0 B1 A2
} |     A0 C1 B2
    |     B0 A1 C2
    | }
    | C0 B1
    | C1
```

- Think that three threads (0, 1, and 2) are running, sharing PC and registers.
- While waiting for one producer two other threads are running.

# Representing Array References as Matrices

```
for (i = 0; i < 4; i++)  
  for (j = 0; j < 4; j++)  
    x[ 2 i - 1 ][ i + j ] = x[ 3 j ][ i + 2 ]
```

- An array reference is written as  $x(lA + a_0)$  where  $l = (i, j)$
- The two references become  $x(lA + a_0)$  and  $x(lB + b_0)$  with

$$A = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} \text{ and } a_0 = \begin{pmatrix} -1 & 0 \end{pmatrix}, \text{ and}$$

$$B = \begin{pmatrix} 0 & 1 \\ 3 & 0 \end{pmatrix} \text{ and } b_0 = \begin{pmatrix} 0 & 2 \end{pmatrix}$$

# The Dependence Matrix

- There is a data dependence between two references  $S(i_1, j_1)$  and  $T(i_2, j_2)$  if they access the same memory location and at least one of the accesses is a write
- If there is an *integer* solution to  $l_1A + a_0 = l_2B + b_0$  there is a dependence between the iterations  $l_1$  and  $l_2$
- Data dependence analysis tests for a possible solution between all references to the same array in a loop nest
- The *dependence distance* is  $l_2 - l_1$  (or  $l_1 - l_2$ , if  $l_2$  comes first)
- The dependence matrix  $D$  consists of all dependence distances in the loop

# An Example Dependence Matrix

```
for (i = 0; i < 4; i++)
  for (j = 0; j < 4; j++)
    x[ i ][ j ] = x[ i - 1 ][ j ] + x[ i ][ j - 1 ];
    /* ref A           ref B           ref C           */
```

$$A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, a_0 = ( 0 \ 0 ) \quad B = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, b_0 = ( -1 \ 0 )$$

$$C = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, c_0 = ( 0 \ -1 )$$

The dependence matrix for the loop nest becomes  $D = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

This  $D$  tells us that neither loop can execute concurrently

# Unimodular Transformations

- We would like to transform our dependence matrix into eg

$D_T = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$  which has no dependencies at level 2 so that the inner loop can execute in parallel

- By the finding unimodular matrix  $U$  such that  $D_T = DU$  we can rewrite the loop and execute the inner loop in parallel

- For our example  $U = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$  and the new loop variables  
 $(k_1, k_2) = (i, j) \cdot U$

```
for (k1 = 0; k1 <= 6; k1++)  
  for (k2 = max(0, k1 - 3); k2 <= min(3, k1); k2++)  
    x[k1 - k2][k2] = x[k1 - k2 - 1][k2] + x[k1 - k2][k2 - 1];
```

# Loop Parallelization

- Parallel inner loops can be exploited for:
  - Modulo-scheduling
  - Vectorization, eg using modern SIMD instructions
- Parallel outer loops can be exploited for:
  - Parallel computers, eg shared-memory multiprocessors
  - PS3, ie its Cell Processor with one Power and eight SPU processors.

# Optimizing Compilers Hall of Fame at LTH

Year	Group	Programme	Cycles
2010	Joakim Andersson/Jon Steen	D	126616
2009	Manfred Dellkrantz/Jesper Öqvist	D	950
2008	Jonas Paulsson	D	18977
2007	Björn Carlin/Hans Gylling	$\pi$ /D	1047
2006	Fredrik Nilsson	D	3388
2005	Mats Mattsson	$\pi$	5738
2004	Jonas Åström	$\pi$	21538
2003	Alexander Malmberg	D	3744
2002	Richard Johansson	D	7930
2001	Bo Do/Per Fransson	CS	65776
2000	Per Cederberg	PhD/Robotics	10768

- Three of these persons now work full time as optimizing compiler engineers. Jonas Paulsson at Ericsson in Stockholm, and Jonas Åström and Mats Mattsson at Nema Labs in Göteborg.
- The benchmark has sometimes been modified over the years and for 2011 there is a new one.