Contents

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

EDAN75 Optimizing compilers

- Course book at amazon.se: ISBN 9781725930483
- Two compulsory labs: dominance analysis and SSA form
- We will use E:Alfa and discord for the labs
- Twelve lectures
- Three projects:
  - Two with vcc: constant propagation and dead code elimination
  - One with llvm: a new SSA-optimization
- Oral exam: sign up at calendly.com/forsete

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 itself, 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
- Suppose you are a product manager and your engineers spend 1000 programmer hours to improve the performance of the product by 1%, then it may be useful to check out improving the compiler as well (real world example and exactly what management decided)

Use different compilers and optimization levels!

This can help with:

- detect bugs in your code which happened to go undetected with one compiler
- detect non-portable code — which depends on
  - unspecified behavior (e.g. evaluation order of parameters)
  - implementation-defined behavior (e.g. sizes of integer types)
- detect compiler bugs
- better insights into which compilers are best on different kinds of source code (or is one best for all on your favorite machine?)

Overview of the Internals of an Optimizing Compiler

- Lexical, syntactic and semantic analysis: output is an abstract syntax tree (AST)
- Translate 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 selection, instruction scheduling and register allocation

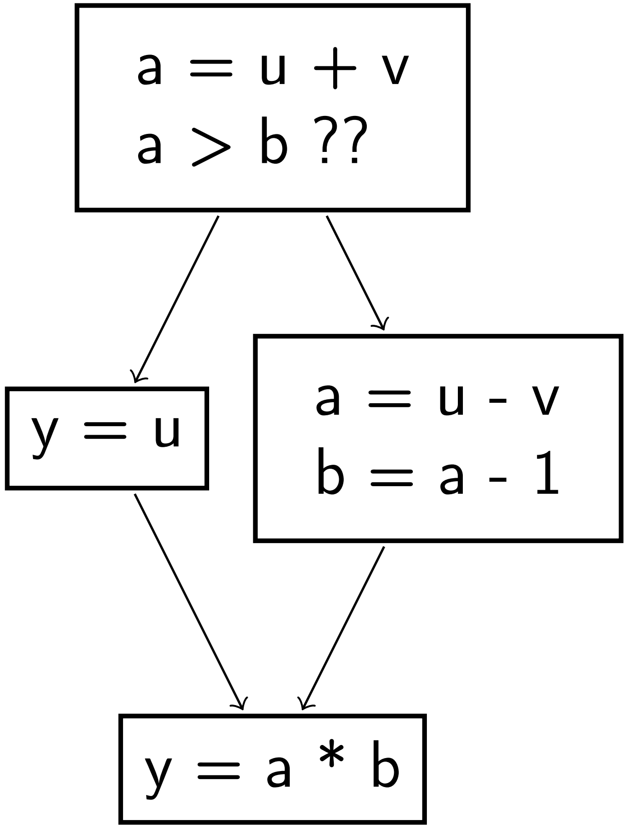
Lexical Analysis and Parsing

- 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

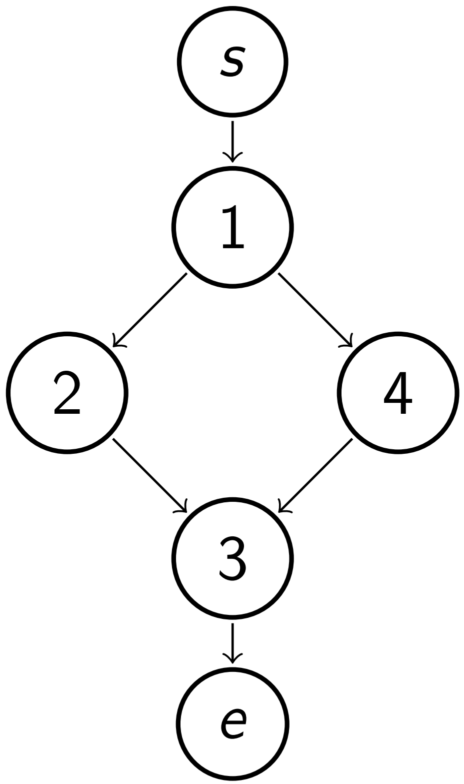
```
a = u + v;
if (a > b) {
           y = u;
} else {
           a = u - v;
           b = a - 1;
}
y = a * b;
```

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

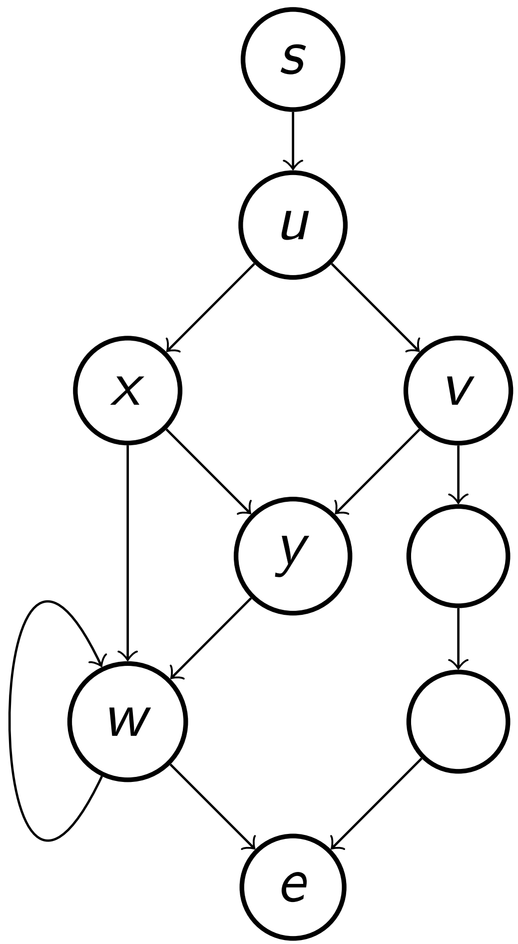
Control-Flow Graph: the CFG View



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

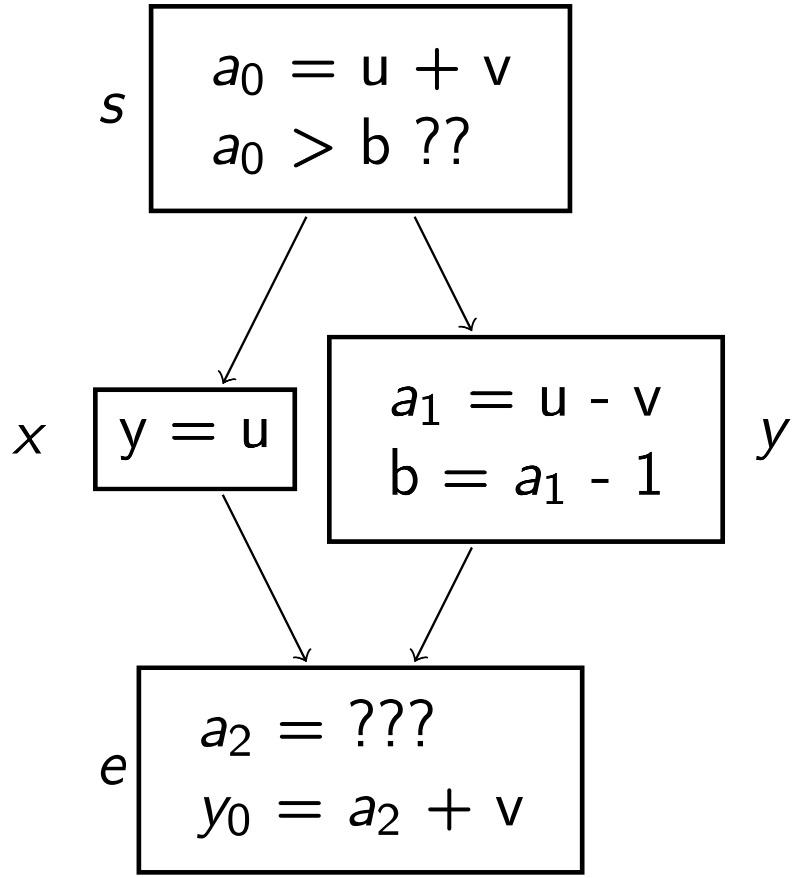
- The fastest algorithm for finding dominators was discovered by Robert Tarjan in 1979.

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

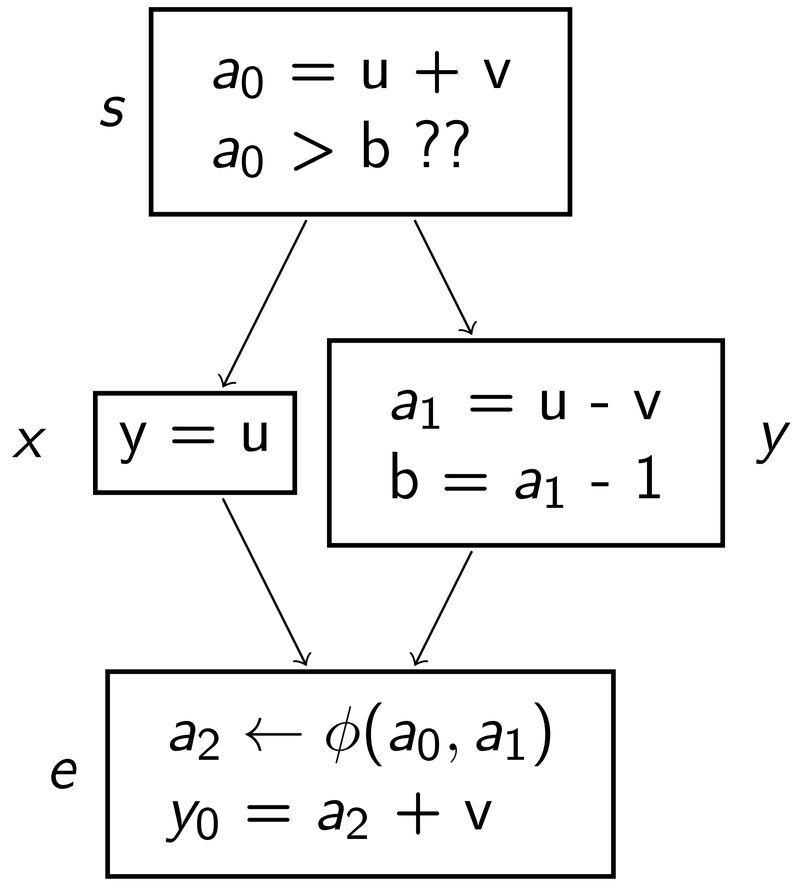
But what to do when paths from different assignments join???

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 ϕ-function.

Our Example Translated to SSA Form



A Function Translated to SSA Form

- We insert a ϕ-function where the paths from two different assignments of the same variable join
- With the ϕ-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 `y0` with `x0`
- The values of `x0` and `y0` do not change after the definition (in a static sense)

Hash-Based Value Numbering

Useful rules if A is an integer

```
2 * a   =>  a << 1
a / 2   =>  a >> 1  OK if unsigned integer
a - a   =>  0
1 * a   =>  a
0 * a   =>  0
```

- Shift right is defined in Java to be arithmetic but may be logic in C/C++
- What is the value of $\infty \times 0$ 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;                          a = u + v;
if (...) {                          if (...) {
    ...;                                ...;
                 ===>                   t = a * b;
} else {                            } else {
    a = u - v;                          a = u - v;
    x = a * b;                          t = a * b;
                                        x = t;
}                                   }
y = a * b;                          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 is a 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

- Essentially, a copy is inserted for each operand of the ϕ-function
- One copy instruction for each predecessor node, i.e. for each operand.
- Each copy writes to the destination of the ϕ-function.
- A clever register allocator will put a_0, a_1 and a_2 in the same register and remove the COPY
- We will later see that there are some complications we must take into account to avoid bugs when doing the translation from SSA form, but the principle is to insert copy statements.

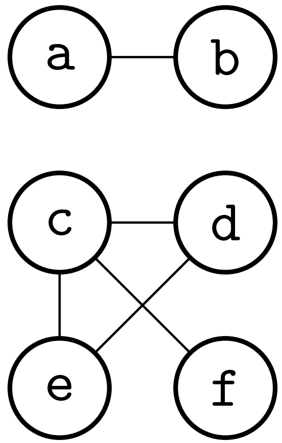
Register Allocation

```
a = 1
b = a + 2
c = a - b
d = c
e = d + 1
f = d - e
return c + f
```

- Which variables cannot use the same register?
- How many registers are needed?

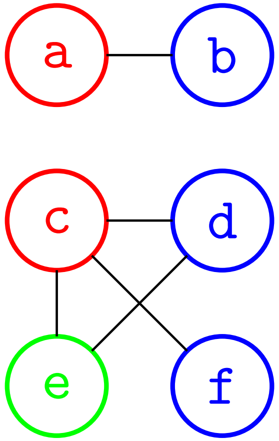
The Interference Graph

```
a = 1
b = a + 2
c = a - b
d = c
e = d + 1
f = d - e
return c + f
```



Coloring the Interference Graph

```
a = 1
b = a + 2
c = a - b
d = c
e = d + 1
f = d - e
return c + f
```



- This interference graph needs three colors.
- Can we use fewer colors?

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 II (II = 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 II, and data dependencies are also checked with respect to II.
- If a valid schedule with II is not found, II 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 $x(IA + a_0)$ where $I = (i, j)$
- The two references become $x(IA + a_0)$ and $x(IB + b_0)$ with $A = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}$ and $a_0 = \begin{pmatrix} -1 & 0 \end{pmatrix}$, and $B = \begin{pmatrix} 0 & 1 \\ 3 & 0 \end{pmatrix}$ 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 $I_1 A + a_0 = I_2 B + b_0$ there is a dependence between the iterations I_1 and I_2
- Data dependence analysis tests for a possible solution between all references to the same array in a loop nest
- The *dependence distance* is $I_2 - I_1$ (or $I_1 - I_2$, if I_2 comes first)
- The dependence matrix D consists of a 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          */
```

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

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

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

Optimizing Compilers Hall of Fame at Lund University

2023	?	?
2020	Simon Andersson, Mattias Leifsson, Micael Pater	D
2018	Alexander Hansson	D
2016	Johan Ju	E
2014	Karl Hylén	F
2013	Erik Hogeman/Mads Nielsen	D
2012	Martin Nitsche	Math. Göttingen
2011	Linus Åkesson	PhD/CS
2010	Joakim Andersson/Jon Steen	D
2009	Manfred Dellkrantz/Jesper Öqvist	D
2008	Jonas Paulsson	D
2007	Björn Carlin/Hans Gylling	π/D
2006	Fredrik Nilsson	D
2005	Mats Mattsson	π
2004	Jonas Åström	π
2003	Alexander Malmberg	D
2002	Richard Johansson	D
2001	Bo Do/Per Fransson	CS
2000	Per Cederberg	PhD/Robotics