Contents

- Motivation
- Overview of optimising compiler internals
- Control flow analysis
- Scalar optimisations on SSA Form
- Register allocation
- Instruction scheduling
- Vectorisation

The compiler is the programmer's most important tool

- The programmer with knowledge about optimising compilers knows what the compiler can optimise faster and better than himself/herself as well as the limitations of compilers, ie, how to help it getting the best performance.

- The competent programmer focuses on writing code which is correct, efficient, and easy to maintain.

- Using optimising compilers improves programmer productivity.

- If your project spends 100 programmer hours to improve the performance by one percent, management should consider getting better compilers!

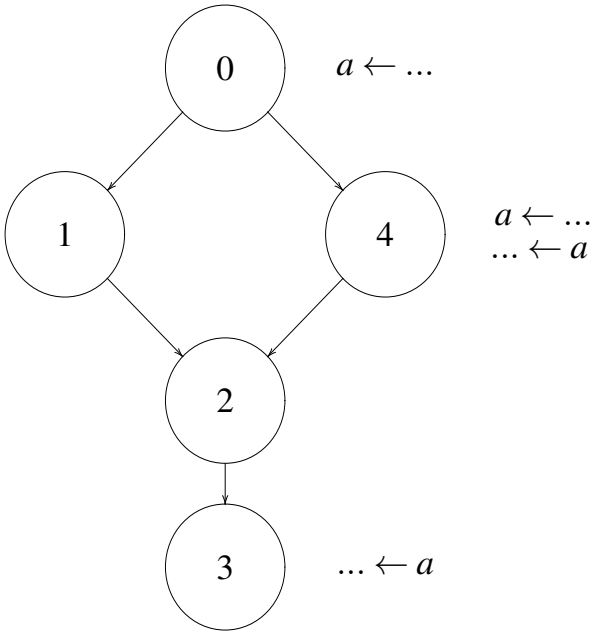
- Using different aggressive optimising compilers is more likely to expose programming bugs/nonportable code than using only one compiler, especially for C/C++.

Overview of the internals of an optimising compiler

- Lexical, syntactic and semantic analysis: very interesting and sometimes challenging. Output is an abstract syntax tree (AST).
- Code generation: rewrites the AST to three-address code — similar to assembler
- Control flow analysis: represents a function as a directed graph of straight line code
- Initial optimisations such as constant propagation
- High-order transformations: vectorisation, parallelisation, locality optimisation
- Scalar optimisations
- Instruction scheduling and register allocation

Control-flow graph (CFG)

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

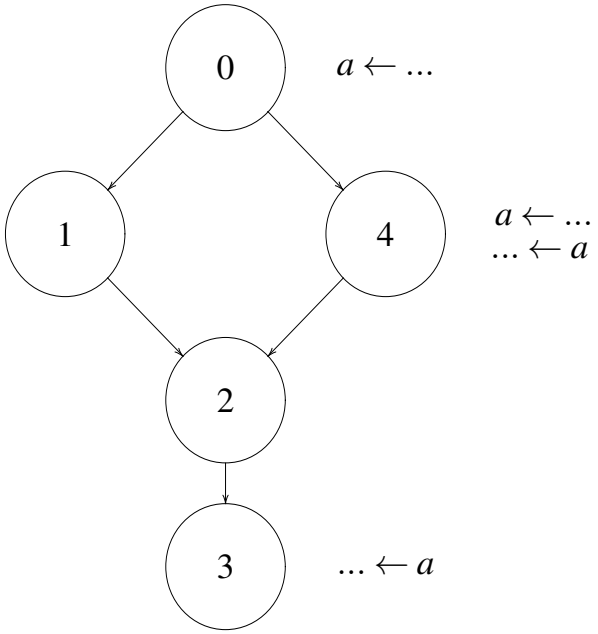


Basic block: sequence of instructions with no label or branch

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

Special node: S is the first node in the CFG

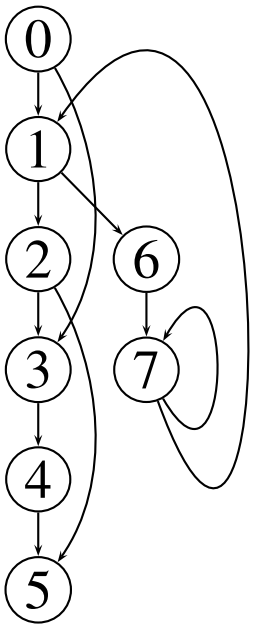
Dominance in the CFG



- u dominates v if all paths from S to v include u.
- 2 dominates 3, S dominates all nodes.

Dominance analysis: finding who dominates who

- In the Optimising compilers course we study an efficient dominance analysis algorithm from 1979 invented by Tarjan and Lengauer (still fastest even if two recent linear algorithms have been aimed to beat it)



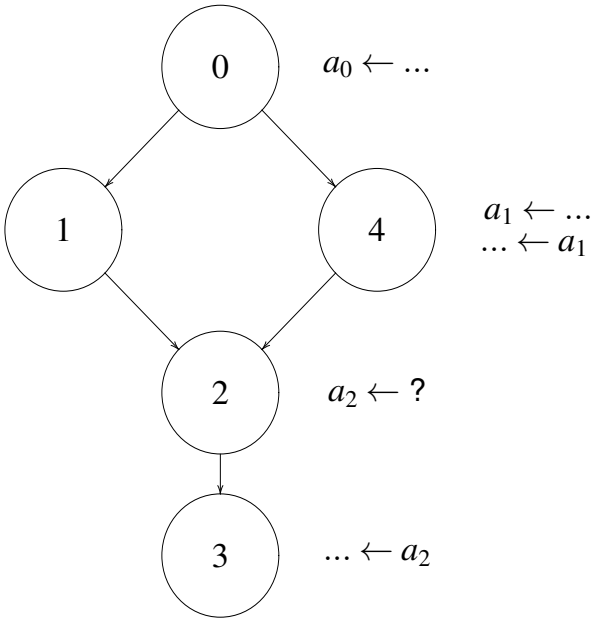
Static Single Assignment: SSA form

- 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 optimisation algorithms
- Invented by IBM Research Yorktown Heights in New York

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

The ϕ-function

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

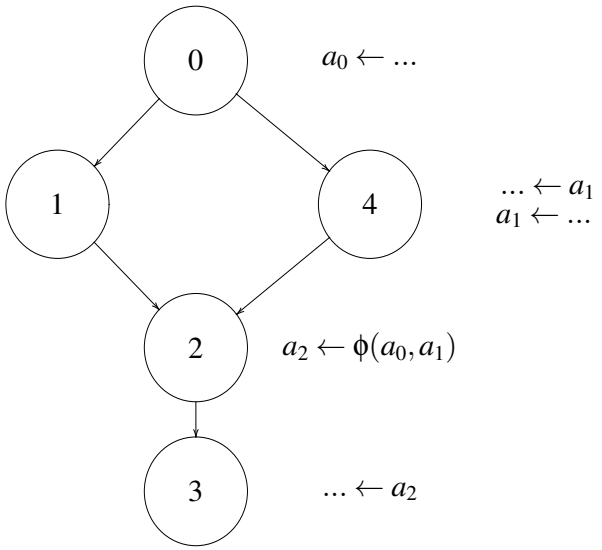


In node 2: if we came from node 1 we let $a_2 \leftarrow a_0$ and if we came from node 4 we let $a_2 \leftarrow a_1$. This operation is called the ϕ-function and is written:

$$a_2 \leftarrow \phi(a_0, a_1)$$

Our example translated to SSA form

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



- We insert a ϕ-function where the paths from two different definitions 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)

Constant Propagation with Conditional Branches

```
int f(int n, int a, int b)
{
    int   y;

    if (n > 0)
        y = 3;
    else
        y = b;

    return a * y;
}

z = f(1, 4, x);   ===>   z = 12;
```

Hash-based value numbering
Useful rules for integers

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

- SRA = shift right arithmetic as in Java. Implementation-defined in C.
- Hash-based value numbering is typically implemented as part of the translation to SSA form
- What is the value $\infty \times 0$ according to IEEE 754 (ie IEC 60559) ?

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, y;

    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, t;

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

Loop-invariant code motion

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

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

- Initialise 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

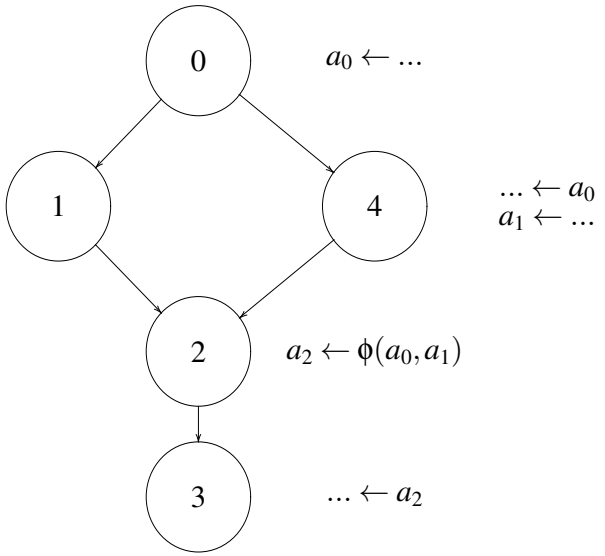
```
s = 4 * i;
do {
    t = load a+s;
    x = x + t;
    i = i + 1;
    s = s + 4;
} while (i < N);
```

```
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

```
a0 = u + v;
if (...) {
        a2 = a0; /* COPY */
} else {
        a1 = u - v;
        x = a1 * b;
        a2 = a1; /* COPY */
}
y = a2 * b;
```



- A copy is inserted for each operand 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

Register allocation

- Reading a variable from a register is much faster than reading from memory.
- Usually register allocation is the most important optimisation.
- 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 colours (K = number of machine registers).
- NP-complete problem, but there is a good practical solution: remove any node from the graph with degree less than K. Such a node is guaranteed to be given a colour if the rest of the graph can be coloured. Why?

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 dependences 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 dependences are also checked with the 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.

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 dependences 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];
```