Contents

- Administration
- Motivation
- Overview of optimizing compiler internals
- Control flow analysis
- Scalar optimizations on SSA Form
- Register allocation
- Instruction scheduling
- Vectorization
Office hours 12.30 – 13.00 every weekday in E:2190
Course book at Amazon: ISBN 9781725930483
Two compulsory labs: dominance analysis and SSA form
Six voluntary exercises
Twelve lectures
Two projects:
  - One with vcc: constant propagation and dead code elimination
  - One with llvm: a new SSA-optimization
Written exam
New this year:
  - More on LLVM and recent research
  - Exam has two parts: one without book, and one with book.
  - No presentation of article — since the book now contains more new results
  - No LLVM lab — instead LLVM project
- LP1: EDAN75 Optimizing Compilers
- LP2: EDAN70 Project in Computer Science
- LP3 — LP4: EDA920 Degree Project in Computer Science
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 buying/creating better compilers! (the example is from a real product and optimization is so time-consuming because it has been optimized for decades)
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. GCC has errors in how it implements `extern inline`:
  - ”pure macro” by GCC
  - ”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).
- 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 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.
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 $u$
The fastest algorithm for finding dominators was discovered by Robert Tarjan 1979.
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???
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

\[ a_0 = u + v \]
\[ a_0 > b \quad ?? \]
\[ y = u \]
\[ a_1 = u - v \]
\[ b = a_1 - 1 \]
\[ a_2 \leftarrow \phi(a_0, a_1) \]
\[ y_0 = a_2 + v \]
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

\[ x_0 = a_0 + b_0; \]
\[
\text{if (\ldots)} \{ \\
    \ldots; \\
\} \\
y_0 = x_0; \quad /* \text{COPY} */
\]
\[
\text{if (\ldots)} \{ \\
    \ldots; \\
\} \\
c_0 = y_0 + 1; \quad /* \text{USE} */
\]

- 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

- $2 \times a \Rightarrow a << 1$
- $a / 2 \Rightarrow a >> 1$ OK if unsigned integer
- $a - a \Rightarrow 0$
- $1 \times a \Rightarrow a$
- $0 \times a \Rightarrow 0$

- 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)

```c
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;
}
```

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

```c
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)
  x = x + a[i];

====>

while (x != y)
  x = x + t;

---

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

====>

do
  x = x + t;
while (x != y);

Which transformation above is valid?
a = u + v;
if (...) {
    ...;
} else {
    a = u - v;
    x = a * b;
}

y = a * b;

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

} else {
    y = t;
More Partial Redundancy Elimination (PRE)

\[
t = a / b;
\]

\[
\text{do do}
\]

\[
x = x + a / b; \quad \Longrightarrow \quad x = x + t;
\]

\[
\text{while (x != y); while (x != y)};
\]

\[
a/b \text{ is partially redundant!}
\]

PRE can move code out of loops without knowledge about loops
Induction Variable Elimination

Also known as Operator strength reduction

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

The primary goal is to get rid of the multiplication
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

```c
s = 4 * i;
do {
s = i * 4;
t = load a+s;
x = x + t;
i = i + 1;
} 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

\[ s = 4 \times i; \]
\[ m = 4 \times N; \]
\[ do \{ \]
\[ \quad t = \text{load } a+s; \]
\[ \quad x = x + t; \]
\[ \quad i = i + 1; \]
\[ \quad s = s + 4; \]
\[ } \text{while} \ (i < N); \]
\[ \]
\[ s = 4 \times i; \]
\[ m = 4 \times N; \]
\[ do \{ \]
\[ \quad t = \text{load } a+s; \]
\[ \quad x = x + t; \]
\[ \quad s = s + 4; \]
\[ } \text{while} \ (s < m); \]

- \( s = i \times b + c \) (we have \( b = 4 \) and \( c = 0 \))
- \( i = \frac{s-c}{b} \)
- \( i < N \Rightarrow \frac{s-c}{b} < N \Rightarrow s < N \times b + c, \text{ if } b > 0 \)
Essentially, a copy is inserted for each operand of the $\phi$-function.

- One copy instruction for each predecessor node, i.e. for each operand.
- Each copy writes to the destination 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.

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.
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 in 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 color 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).

Most good compilers now use his algorithm (LLVM is an exception).
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 color 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 color 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 color which no neighbor has already selected.
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 color 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.
Normally, one loop iteration is executed to completion before the next is started.

In software pipelining the next iteration is started $II$ ($II = \text{initiation interval}$) cycles after the current, without (1) violating data dependencies or (2) using more hardware resources than are available (e.g., 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.
Think that three threads (0, 1, and 2) are running, sharing PC and registers.

While waiting for one producer two other threads are running.
for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++)
        \[ x[2i-1][i+j] = x[3j][i+2] \]

- 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
  \[
  A = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}
  \quad \text{and} \quad
  a_0 = \begin{pmatrix} -1 \\ 0 \end{pmatrix},
  \]
  \[
  B = \begin{pmatrix} 0 & 1 \\ 3 & 0 \end{pmatrix}
  \quad \text{and} \quad
  b_0 = \begin{pmatrix} 0 \\ 2 \end{pmatrix}
  \]
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 $l_1 A + a_0 = l_2 B + b_0$ 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 = \[
    \begin{pmatrix}
        1 & 0 \\
        0 & 1 \\
    \end{pmatrix}
\]
, \[a_0 = (0 \ 0)\]
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 = \[
    \begin{pmatrix}
        1 & 0 \\
        0 & 1 \\
    \end{pmatrix}
\]\n
This \(D\) tells us that neither loop can execute concurrently.
We would like to transform our dependence matrix into
\[ D_T = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \] 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 \((k_1, k_2) = (i, j) \cdot U\)

\[
\text{for (} k_1 = 0; \ k_1 \leq 6; \ k_1++ \) \\
\text{for (} k_2 = \text{max}(0, k_1 - 3); \ k_2 \leq \text{min}(3, k_1); \ k_2++ \) \\
x[k_1 - k_2][k_2] = x[k_1 - k_2 - 1][k_2] + x[k_1 - k_2][k_2 - 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 LTH

<table>
<thead>
<tr>
<th>Year</th>
<th>Group</th>
<th>Programme</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016</td>
<td>Johan Ju</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>Karl Hylén</td>
<td>F</td>
<td>40292</td>
</tr>
<tr>
<td>2013</td>
<td>Erik Hogeman/Mads Nielsen</td>
<td>D</td>
<td>49155</td>
</tr>
<tr>
<td>2012</td>
<td>Martin Nitsche</td>
<td>Math. Göttingen</td>
<td>33526</td>
</tr>
<tr>
<td>2011</td>
<td>Linus Åkesson</td>
<td>PhD/CS</td>
<td>112805</td>
</tr>
<tr>
<td>2010</td>
<td>Joakim Andersson/Jon Steen</td>
<td>D</td>
<td>126616</td>
</tr>
<tr>
<td>2009</td>
<td>Manfred Dellkrantz/Jesper Öqvist</td>
<td>D</td>
<td>950</td>
</tr>
<tr>
<td>2008</td>
<td>Jonas Paulsson</td>
<td>D</td>
<td>18977</td>
</tr>
<tr>
<td>2007</td>
<td>Björn Carlin/Hans Gylling</td>
<td>π/D</td>
<td>1047</td>
</tr>
<tr>
<td>2006</td>
<td>Fredrik Nilsson</td>
<td>D</td>
<td>3388</td>
</tr>
<tr>
<td>2005</td>
<td>Mats Mattsson</td>
<td>π</td>
<td>5738</td>
</tr>
<tr>
<td>2004</td>
<td>Jonas Åström</td>
<td>π</td>
<td>21538</td>
</tr>
<tr>
<td>2003</td>
<td>Alexander Malmberg</td>
<td>D</td>
<td>3744</td>
</tr>
<tr>
<td>2002</td>
<td>Richard Johansson</td>
<td>D</td>
<td>7930</td>
</tr>
<tr>
<td>2001</td>
<td>Bo Do/Per Fransson</td>
<td>CS</td>
<td>65776</td>
</tr>
<tr>
<td>2000</td>
<td>Per Cederberg</td>
<td>PhD/Robotics</td>
<td>10768</td>
</tr>
</tbody>
</table>