Compiler construction

Lecture 8: Control flow graphs and data flow analysis

Alex Gerdes
Vårtermin 2017

Chalmers University of Technology — Gothenburg University

This lecture

- Control-flow graphs
- Liveness analysis
- Register Allocation
- Constant propagation
- Loop optimization
- A larger example

Control-flow graphs

Three-address code

Pseudo-code

To discuss code optimization we employ a (vaguely defined) pseudo-IR called three-address code which uses virtual registers but does not require SSA form.

Instructions

- `x := y # z` where `x`, `y` and `z` are register names or literals and `#` is an arithmetic operator
- `goto L` where `L` is a label
- `if x # y then goto L` where `#` is a relational operator
- `x := y`
- `return x`

Example code

```
    s := 0
    i := 1
L1: if i > n goto L2
    t := i * i
    s := s + t
    i := i + 1
    goto L1
L2: return s
```

Control-flow graph

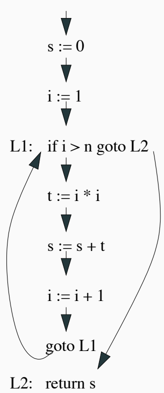
Code as graph

- Each instruction is a node
- Edge from each node to its possible successors

Example code

```
    s := 0
    i := 1
L1: if i > n goto L2
    t := i * i
    s := s + t
    i := i + 1
    goto L1
L2: return s
```

Example as graph



Dataflow analysis

Static analysis

- General approach to code analysis
- Basis for many forms of compiler analysis – but in general we don't know if that path will ever be taken during execution
- Useful for many forms of optimization:
  - Common subexpression elimination
  - Constant propagation
  - Dead code elimination
  - ...
- Results are approximations – we must make sure to err on the correct side
- Within a basic block, simpler methods often suffice

Liveness analysis

Liveness of variables

Definitions and uses

An instruction `x := y # z` defines `x` and uses `y` and `z`.

Liveness of variables

Definitions and uses

An instruction `x := y # z` defines `x` and uses `y` and `z`.

Liveness

A variable `v` is live at a point P in the control-flow graph (CFG) if there is a path from P to a use of `v` along which `v` is not (re)defined.

Liveness of variables

Definitions and uses

An instruction `x := y # z` defines `x` and uses `y` and `z`.

Liveness

A variable `v` is live at a point P in the control-flow graph (CFG) if there is a path from P to a use of `v` along which `v` is not (re)defined.

Uses of liveness information

- Register allocation: a non-live variable need not be kept in register
- Useless-store elimination: a non-live variable need not be stored to memory
- Detecting uninitialized variables: a local variable that is live on function entry

Liveness analysis: concepts

Let n be a node (a single instruction in our example).

Def set

$def(n)$ is the set of variables that are defined in n (a set with 0 or 1 elements)

Use set

$use(n)$ is the set of variables that are used in n

Live-out set

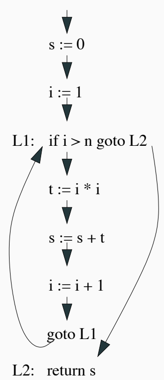
$live_{out}(n)$ is the set of variables that are live at an out-edge of n

Live-in set

$live_{in}(n)$ is the set of variables that are live at an in-edge of n

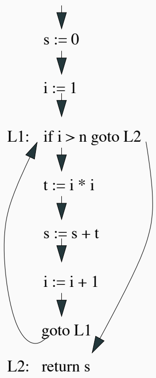
An example

First example revisited



An example

First example revisited



Live-in set

Instr #	Set
1	{ n }
2	{ n, s}
3	{i, n, s}
4	{i, n, s }
5	{i, n, s, t}
6	{i, n, s}
7	{i, n, s}
8	{ s }

How can these be computed?

The dataflow equations

For every node n we have:

$$live_{in}(n) = use(n) \cup (live_{out}(n) - def(n))$$
$$live_{out}(n) = \cup_{s \in succs(n)} live_{in}(s)$$

where $succs(n)$ denote the set of successor nodes to n.

Computation

Let $live_{in}$, def, and use be arrays indexed by nodes.

FOREACH node n DO $live_{in}[n] = \emptyset$
REPEAT
  FOREACH node n DO
    $out = \cup_{s \in succs(n)} live_{in}[s]$
    $live_{in}[n] = use[n] \cup (out - def[n])$
UNTIL no changes in iteration

Solving the equations

Example revisited

Instr	*def*	*use*	*succs*	*live-in*
1	{s}	{}	{2}	{}
2	{i}	{}	{3}	{}
3	{}	{i,n}	{4,8}	{}
4	{t}	{i}	{5}	{}
5	{s}	{s,t}	{6}	{}
6	{i}	{i}	{7}	{}
7	{}	{}	{3}	{}
8	{}	{s}	{}	{}

- Initialization done above
- $live_{in}$ updated from top to bottom in each iteration

Solving the equations

Example revisited

Instr	*def*	*use*	*succs*	*live-in*
1	{s}	{}	{2}	{}
2	{i}	{}	{3}	{}
3	{}	{i,n}	{4,8}	{}
4	{t}	{i}	{5}	{}
5	{s}	{s,t}	{6}	{}
6	{i}	{i}	{7}	{}
7	{}	{}	{3}	{}
8	{}	{s}	{}	{}

- Initialization done above
- $live_{in}$ updated from top to bottom in each iteration
- But is there a better order?

Liveness: a backwards problem

Fixpoint iteration

- We iterate until no live sets change during an iteration; we have reached a fixpoint of the equations
- The number of iterations (and thus the amount of work) depends on the order in which we use the equations within an iteration
- Since liveness info propagates from successors to predecessors in the CFG, we should start with the last instruction and work backwards

Another node order

Working from bottom to top, we get

Instr	*def*	*use*	*succs*	$live_{in_0}$	$live_{in_1}$	$live_{in_2}$
1	{s}	{}	{2}	{}	{n}	{n}
2	{i}	{}	{3}	{}	{n,s}	{n,s}
3	{}	{i,n}	{4,8}	{}	{i,n,s}	{i,n,s}
4	{t}	{i}	{5}	{}	{i,s}	{i,n,s}
5	{s}	{s,t}	{6}	{}	{i,s,t}	{i,n,s,t}
6	{i}	{i}	{7}	{}	{i}	{i,n,s}
7	{}	{}	{3}	{}	{}	{i,n,s}
8	{}	{s}	{}	{}	{s}	{s}

Implementing data flow analysis

Data structures

- Any standard data structure for graphs will work
- For sets of variables one may use bit arrays with one bit per variable; then union is bit-wise or, intersection bit-wise and complement bit-wise negation

Termination

The live sets grow monotonically in each iteration, so the number of iterations is bounded by $V \cdot N$, where N is number of nodes and V the number of variables. In practice, for realistic code, the number of iterations is much smaller.

Basic blocks

Motivations

- Control-graph with instructions as nodes become big
- Between jumps, the graph structure is trivial (straight-line code)

Basic blocks

Motivations

- Control-graph with instructions as nodes become big
- Between jumps, the graph structure is trivial (straight-line code)

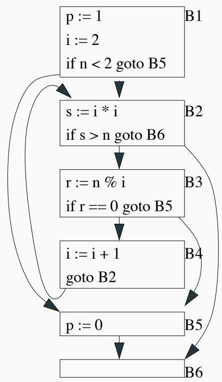
Definition

- A basic block starts at a labelled instruction or after a conditional jump
- First basic block starts at beginning of function
- A basic block ends at a (conditional) jump

We ignore code where an unlabeled statement follows an unconditional jump (such code is unreachable).

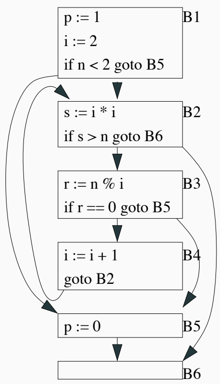
Example

Testing if `n` is prime



Example

Testing if `n` is prime



Notes

- Edges correspond to branches
- Jump destinations are now blocks, not instructions
- We may insert empty blocks
- Analysis of control-flow graphs often done on graph with basic blocks as nodes

Liveness analysis for CFGs of basic blocks

We can easily modify data flow analysis to work on control flow graphs of basic blocks.

With knowledge of $live_{in}$ and $live_{out}$ for basic blocks it is easy to find the set of live variables at each instruction.

How do the basic concepts need to be modified to apply to basic blocks?

Modified definitions for CFG of basic blocks

Let n be a node in a control-flow graph representing a basic block.

Def set
$def(n)$ is the set of variables that are defined in an instruction in n

Use set
$use(n)$ is the set of variables that are used in an instruction in n before a possible redefinition of the variable

Live-out set
$live_{out}(n)$ is the set of variables that are live at an out-edge of n

Live-in set
$live_{in}(n)$ is the set of variables that are live at an in-edge of n

Register Allocation

Register allocation

An important code transformation
When translating an IR with (infinitely many) virtual registers to code for a real machine, we must:

- assign virtual registers to physical registers
- write register values to memory (spill), at program points when the number of live virtual registers exceeds the number of available registers

Register allocation is very important; good allocation can make a program run an order of magnitude faster (or more) as compared to poor allocation.

The interference graph

Live sets and register usage

- A variable is live at a point in the CFG, if it may be used in the remaining code without assignment in between
- If two variables are live at the same point in the CFG, they must be in different registers
- Conversely, two variables that are never live at the same time can share a register

Interfering variables

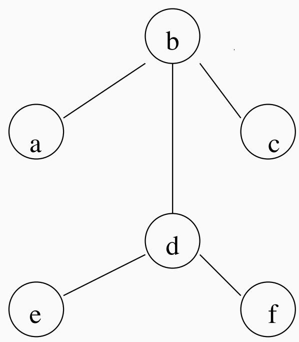
- We say that variables `x` and `y` interfere if they are both live at some point
- The register interference graph has variables as nodes and edges between interfering variables

Which variables interfere?

```
void bubble_sort(int[] a) {
  int i, j, t, n;
  n = a.length;
  for (i = 0; i < n; i++) {
    for (j = 1; j < n - i; j++) {
      if (a[j - 1] > a[j]) {
        t        = a[j - 1];
        a[j - 1] = a[j];
        a[j]     = t;
      }
    }
  }
}
```

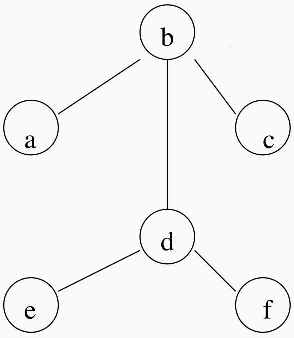
An example

How many registers are needed?



An example

How many registers are needed?

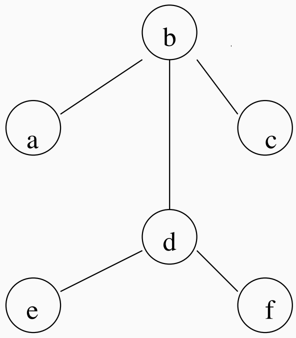


Answer: Two!

Use one register for *a*, *c* and *d*, the other for *b*, *e* and *f*.

An example

How many registers are needed?



Answer: Two!

Use one register for *a*, *c* and *d*, the other for *b*, *e* and *f*.

Reformulation

To assign K registers to variables given an interference graph can be seen as colouring the nodes of the graph with K colours, with adjacent nodes getting different colours.

Register allocation by graph colouring

The algorithm (K colours available)

1. Find a node n with less than K edges and remove n and its edges from the graph and put it on a stack
2. Repeat with remaining graph until either
   a. only K nodes remain or
   b. all remaining nodes have at least K adjacent edges

Register allocation by graph colouring

The algorithm (K colours available)

1. Find a node n with less than K edges and remove n and its edges from the graph and put it on a stack
2. Repeat with remaining graph until either
   a. only K nodes remain or
   b. all remaining nodes have at least K adjacent edges

Comments

2.a In the first case, give each remaining node a distinct colour and pop nodes from the stack, inserting them back into the graph with their edges and colouring them

2.b In the second case, we may need to spill a variable to memory. Optimistic algorithm: choose one variable and push on the stack. Later, when popping the stack, we may be lucky and find that the neighbours use at most $K - 1$ colours.

Complexity

A hard problem

- The problem to decide whether a graph can be K-coloured is NP-complete
- The simplify/select algorithm on the previous slide works well in practice; its complexity is $O(n^2)$, where n is the number of virtual registers used
- When optimistic algorithm fails, memory store and fetch instructions must be added and algorithm restarted
- Heuristics to choose variable to spill:
  - Little use+def within loop
  - Interference with many other variables

Move instructions

An example

```
t := s
x := s + 1
y := t + 2
...
```

`s` and `t` interfere, but if `t` is not later redefined, they may share a register

Move instructions

An example

```
t := s
x := s + 1
y := t + 2
...
```

`s` and `t` interfere, but if `t` is not later redefined, they may share a register

Coalescing

Move instructions `t := s` can sometimes be removed and the nodes `s` and `t` merged in the interference graph.

Conditions:

- No interference between `s` and `t` for other reasons
- The graph must not become harder to colour

Linear scan register allocation

Compilation time vs code quality

- Register allocation based on graph colouring produces good code, but requires significant compilation time
- For JIT compiling allocation time is a problem
- The Java HotSpot compiler uses a linear scan register allocator
- Much faster and in many cases only 10% slower code

The linear scan algorithm

Preliminaries

- Number all the instructions 1, 2, ..., in some way
  - for now, think of numbering them from top to bottom
  - Other instruction orderings improves the algorithm; a depth first ordering is recommended
- Do a simplified liveness analysis, assigning a live range to each variable.

  A live range is an interval of integers starting with the number of the instruction where the variable is first defined and ending with the number where it is last used.
- Sort live ranges in order of increasing start points into list L

The linear scan algorithm

The algorithm

- Maintain a list, called A, of live ranges that have been assigned registers; A is sorted by increasing end points and initially empty
- Traverse L and for each interval I:
  - Traverse A and remove intervals with end points before start point of I
  - If length of A is smaller than number of registers, add I to A; otherwise spill either I or an element of A
  - In the latter case, the choice of interval to spill is usually to keep interval with longest remaining range in A

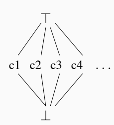
Constant propagation

Simple constant propagation

A dataflow analysis based on SSA form

- Uses values from a lattice L with elements
- $\bot$: unreachable, as far as the analysis can tell
- $c_1, c_2, c_3, \ldots$: the value is constant, as indicated
- $\top$: yet unknown, may be constant
- Each variable v is assigned an initial value $val(v) \in L$:
  - variables with definitions $v := c$ get $val(v) = c$
  - input variables/parameters v get $val(v) = \top$
  - and the rest get $val(v) = \bot$

The lattice L



The lattice order

$\bot \leq c \leq \top$ for all c

c_i and c_j not related

Propagation phase 1

Iteration

- Initially, place all names n with $val(n) \neq \top$ on a worklist
- Iterate by picking a name from the worklist, examining its uses and computing *val* of the RHS's, using rules as

$$\begin{aligned} 0 \cdot x &= 0 \quad \text{(for any } x\text{)} \\ x \cdot \bot &= \bot \\ x \cdot \top &= \top \quad (x \neq 0) \end{aligned}$$

  plus ordinary multiplication for constant operands
- For ϕ-functions, we take the join $\vee$ of the arguments, where $\bot \vee x = x$ for all x, $\top \vee x = \top$ for all x, and

$$c_i \vee c_j = \begin{cases} \top, & \text{if } c_i \neq c_j \\ c_i, & \text{otherwise} \end{cases}$$

Propagation phase 2

Iteration, continued

Update *val* for the defined variables, putting variables that get a new value back on the worklist.

Terminate when worklist is empty.

Termination

Values of variables on the worklist can only increase (in lattice order) during iteration. Each value can only have its value increased twice.

Loop optimization

Optimizations of loops

In computationally demanding applications, most of the time is spent in executing (inner) loops.

Thus, an optimizing compiler should focus its efforts in improving loop code.

The first task is to identify loops in the code. In the source code, loops are easily identified, but how to recognize them in a low level IR code?

A loop in a CFG is a subset of the nodes that

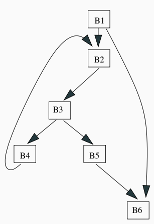
- has a header node, which dominates all nodes in the loop
- has a back edge from some node in the loop back to the header
  A back edge is an edge where the head dominates the tail

Dominators

Definition

- In a CFG, node n dominates node m if every path from the start node to m passes through n
- Particular case: we consider each node to dominate itself
- Concept has many uses in compilation

Prime test CFG



Moving loop-invariant code out of the loop

A simple example

```
for (i = 0; i < n; i++)
  a[i] = b[i] + 3 * x;
```

should be replaced by

```
t = 3 * x;
for (i = 0; i < n; i++)
  a[i] = b[i] + t;
```

Moving loop-invariant code out of the loop

A simple example

```
for (i = 0; i < n; i++)
  a[i] = b[i] + 3 * x;
```

should be replaced by

```
t = 3 * x;
for (i = 0; i < n; i++)
  a[i] = b[i] + t;
```

We need to insert an extra node (a pre-header) before the header.

Moving loop-invariant code out of the loop

A simple example

```
for (i = 0; i < n; i++)
  a[i] = b[i] + 3 * x;
```

should be replaced by

```
t = 3 * x;
for (i = 0; i < n; i++)
  a[i] = b[i] + t;
```

We need to insert an extra node (a pre-header) before the header.

Not quite as simple

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[i][j] = b[i][j] + 10 * i
            + 3 * x;
```

should be replaced by

```
t = 3 * x;
for (i = 0; i < n; i++) {
  u = 10 * i + t;
  for (j = 0; j < n; j++)
    a[i][j] = b[i][j] + u;
}
```

Induction variables

Basic

A basic induction variable is an (integer) variable which has a single definition in the loop body, which increases its value with a fixed (loop-invariant) amount. For example:

```
n = n + 3
```

A basic IV will assume values in arithmetic progression when the loop executes.

Derived

Given a basic IV we can find a collection of derived IVs, each of which has a single def of the form:

```
m = a * n + b;
```

where `a` and `b` are loop-invariant.

The def can be extended to allow RHS of the form `a * k + b` where also `k` is an already established derived IV.

Strength reduction for IVs

- `n` is a basic IV (only def is to increase by 1)
- `k` is derived IV

```
while (n < 100) {
   k = 7 * n + 3;
   a[k]++;
   n++;
}
```

Strength reduction for IVs

- `n` is a basic IV (only def is to increase by 1)
- `k` is derived IV

```
while (n < 100) {
   k = 7 * n + 3;
   a[k]++;
   n++;
}
```


- Replace multiplication involved in def of derived IV by addition

```
k = 7 * n + 3;
while (n < 100) {
   a[k]++;
   n++;
   k += 7;
}
```

Strength reduction for IVs

- `n` is a basic IV (only def is to increase by 1)
- `k` is derived IV

```
while (n < 100) {
   k = 7 * n + 3;
   a[k]++;
   n++;
}
```

- Replace multiplication involved in def of derived IV by addition
- *Could there be some problem with this transformation?*

```
k = 7 * n + 3;
while (n < 100) {
   a[k]++;
   n++;
   k += 7;
}
```

Strength reduction for IVs, continued

- The loop might not execute at all, in which case k would not be evaluated
- Better to perform loop inversion first

```
if (n < 100) {
   k = 7 * n + 3;
   do {
     a[k]++;
     n++;
     k + =7;
   } while ( n < 100);
}
```

Strength reduction for IVs, continued

- The loop might not execute at all, in which case k would not be evaluated
- Better to perform loop inversion first


```
if (n < 100) {
   k = 7 * n + 3;
   do {
     a[k]++;
     n++;
     k + =7;
   } while ( n < 100);
}
```


- If n is not used after the loop, it can be eliminated from the loop

```
if (n < 100) {
   k = 7 * n + 3;
   do {
     a[k]++;
     k += 7;
   } while (k < 703);
}
```

Loop unrolling

```
for (i = 0; i < 100; i++)
  a[i] = a[i] + x[i];
```

```
for (i = 0; i < 100; i += 4) {
  a[i]   = a[i] + x[i];
  a[i+1] = a[i+1] + x[i+1];
  a[i+2] = a[i+2] + x[i+2];
  a[i+3] = a[i+3] + x[i+3];
}
```

Loop unrolling


```
for (i = 0; i < 100; i++)
  a[i] = a[i] + x[i];
```

```
for (i = 0; i < 100; i += 4) {
  a[i]   = a[i] + x[i];
  a[i+1] = a[i+1] + x[i+1];
  a[i+2] = a[i+2] + x[i+2];
  a[i+3] = a[i+3] + x[i+3];
}
```


- In which ways is this an improvement?
- What could be the disadvantages?

A larger example

An example of optimization in LLVM

```
int f () {
  int i, j, k;
  i = 8;
  j = 1;
  k = 1;
  while (i != j) {
    if (i == 8)
      k = 0;
    else
      i++;
    i = i + k;
    j++;
  }
  return i;
}
```

An example of optimization in LLVM

```
int f () {
  int i, j, k;
  i = 8;
  j = 1;
  k = 1;
  while (i != j) {
    if (i == 8)
      k = 0;
    else
      i++;
    i = i + k;
    j++;
  }
  return i;
}
```

Comments

Human reader sees, with some effort, that the C/JAVALETTE function `f` returns 8.

We follow how LLVM's optimizations will discover this fact.

Step 1: Naive translation to LLVM

```
define i32 @f()  {
entry:
  %i = alloca i32
  %j = alloca i32
  %k = alloca i32
  store i32 8, i32* %i
  store i32 1, i32* %j
  store i32 1, i32* %k
  br label %while.cond
while.cond:
  %tmp = load i32* %i
  %tmp1 = load i32* %j
  %cmp = icmp ne i32 %tmp, %tmp1
  br i1 %cmp, label %while.body,
              label %while.end
while.body:
  %tmp2 = load i32* %i
  %cmp3 = icmp eq i32 %tmp2, 8
  br i1 %cmp3, label %if.then,
               label %if.else
if.then:
  store i32 0, i32* %k
  br label %if.end
if.else:
  %tmp4 = load i32* %i
  %inc = add i32 %tmp4, 1
  store i32 %inc, i32* %i
  br label %if.end
if.end:
  %tmp5 = load i32* %i
  %tmp6 = load i32* %k
  %add = add i32 %tmp5, %tmp6
  store i32 %add, i32* %i
  %tmp7 = load i32* %j
  %inc8 = add i32 %tmp7, 1
  store i32 %inc8, i32* %j
  br label %while.cond
while.end:
  %tmp9 = load i32* %i
  ret i32 %tmp9
}
```

Step 2: Translating to SSA form (`opt -mem2reg`)

```
define i32 @f() {
entry:
  br label %while.cond

while.cond:
  %k.1 = phi i32 [ 1, %entry ],
                 [ %k.0, %if.end ]
  %j.0 = phi i32 [ 1, %entry ],
                 [ %inc8, %if.end ]
  %i.1 = phi i32 [ 8, %entry ],
                 [ %add, %if.end ]
  %cmp = icmp ne i32 %i.1, %j.0
  br i1 %cmp, label %while.body,
              label %while.end

while.body:
  %cmp3 = icmp eq i32 %i.1, 8
  br i1 %cmp3, label %if.then,
               label %if.else

if.then:
  br label %if.end

if.else:
  %inc = add i32 %i.1, 1
  br label %if.end

if.end:
  %k.0 = phi i32 [ 0, %if.then ],
                 [ %k.1, %if.else ]
  %i.0 = phi i32 [ %i.1, %if.then ],
                 [ %inc, %if.else ]
  %add = add i32 %i.0, %k.0
  %inc8 = add i32 %j.0, 1
  br label %while.cond

while.end:
  ret i32 %i.1
}
```

Step 3: Sparse Conditional Constant Propagation (`opt -sccp`)

```
define i32 @f() {
entry:
  br label %while.cond

while.cond:
  %j.0 = phi i32 [ 1, %entry ],
                 [ %inc8, %if.end ]
  %k.1 = phi i32 [ 1, %entry ],
                 [ 0, %if.end ]
  %cmp = icmp ne i32 8, %j.0
  br i1 %cmp, label %while.body,
              label %while.end

while.body:
  br i1 true, label %if.then,
              label %if.else

if.then:
  br label %if.end

if.else:
  br label %if.end

if.end:
  %inc8 = add i32 %j.0, 1
  br label %while.cond

while.end:
  ret i32 8
}
```

Step 4: CFG Simplification (`opt -simplifycfg`)

```
define i32 @f() {
entry:
  br label %while.cond

while.cond:
  %j.0 = phi i32 [ 1, %entry ],
                 [ %inc8, %if.end ]
  %k.1 = phi i32 [ 1, %entry ],
                 [ 0, %if.end ]
  %cmp = icmp ne i32 8, %j.0
  br i1 %cmp, label %if.end,
              label %while.end

if.end:
  %inc8 = add i32 %j.0, 1
  br label %while.cond

while.end:
  ret i32 8
}
```

Step 4: CFG Simplification (`opt -simplifycfg`)


```
define i32 @f() {
entry:
  br label %while.cond

while.cond:
  %j.0 = phi i32 [ 1, %entry ],
                 [ %inc8, %if.end ]
  %k.1 = phi i32 [ 1, %entry ],
                 [ 0, %if.end ]
  %cmp = icmp ne i32 8, %j.0
  br i1 %cmp, label %if.end,
              label %while.end

if.end:
  %inc8 = add i32 %j.0, 1
  br label %while.cond

while.end:
  ret i32 8
}
```


Comments

If the function terminates, the return value is 8.

`opt` has not yet detected that the loop is certain to terminate.

Step 5: Dead Loop Deletion (opt -loop-deletion)

```
define i32 @f() {
entry:
  br label %while.end

while.end:
  ret i32 8
}
```

Step 5: Dead Loop Deletion (opt -loop-deletion)

```
define i32 @f() {
entry:
  br label %while.end

while.end:
  ret i32 8
}
```

One more -simplifycfg step yields finally

```
define i32 @f() {
entry:
  ret i32 8
}
```

Step 5: Dead Loop Deletion (opt -loop-deletion)

```
define i32 @f() {
entry:
  br label %while.end

while.end:
  ret i32 8
}
```

One more -simplifycfg step yields finally

```
define i32 @f() {
entry:
  ret i32 8
}
```

For realistic code, dozens of passes are performed, some of them repeatedly. Many heuristics are used to determine order.

Use `opt -O3` for a default selection.

Summing up

On optimization

- We have only looked at a few of many, many techniques
- Modern optimization techniques use sophisticated algorithms and clever data structures
- Frameworks such as LLVM make it possible to get the benefits of state-of-the-art techniques in your own compiler project