Compiler construction 2012

Lecture 9

Code optimization

- Control-flow graph and basic blocks
- Data-flow analysis
- Liveness analysis

Optimization: desired properties

Improve the code

- Make execution faster.
- Make execution consume less power.
- Make program smaller.

These goals can be contradictory.

Don't change semantics

- Don't change values returned.
- Don't change side effects.
- Don't change runtime errors(!).
- Don't change termination properties.

Often subtle points.

Full optimization is impossible

Full employment theorem for compiler writers

We cannot build a compiler that optimizes all programs fully for program size.

Proof: The smallest non-terminating program without visible effects is `while (true) {}`
A fully optimizing compiler would translate any non-terminating program to this – and thus solve the halting problem.

Similar results for other optimization criteria.

Optimization at different stages

Where/when should we optimize?

We can optimize at different stages:

- Source code.
- Abstract syntax trees.
- LLVM/JVM byte code or other IR.
- Native code.

Except for source code, compilers do optimization at all these stages.

Inlining

Replace function call by body

Parameters need to be substituted by arguments.
Renaming of vars may be needed.

+ Function call overhead disappears.
+ Activation record disappears.
+ Memory traffic reduced.
+ New optimization opportunities.
- Code becomes bigger.

This is often done at AST level.

For imperative code (with statements and return),
rewrite to return a var and place the var at the call site.

In the rest of the lecture, we focus on three address code/native code optimization.

Code optimization

Improvement opportunities

- Naive syntax-directed translation often gives code that can be "obviously" improved.
- Compiler-generated code such as e.g. address calculations for array elements even more so.
- One improvement often opens for other improvements.

Consequences

- If you know that subsequent optimizations will be done, do not try to be clever in the first code generation step.
- Never rule out an optimization as useless by thinking that "the programmer would never write that" – the compiler itself might do so!

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

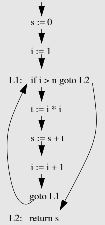
Code as graph

- 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



Static vs dynamic analysis

Dynamic analysis

If in some execution of the program . . .

Dynamic properties are in general undecidable.
Compare with the halting problem:
“P halts” vs “P reaches instruction I”.

Static analysis

If there is a path in the control-flow graph . . .

Basis for many forms of compiler analysis –
but in general we don’t know if that path will ever be taken during execution.
Results are approximations – we must make sure to err on the correct side.

Dataflow analysis

A static analysis

- General approach to code analysis.
- Useful for many forms of **intraprocedural optimization**:
  - Common subexpression elimination,
  - Constant propagation,
  - Dead code elimination,
  - . . .
- Within a basic block, simpler methods often suffice.

Example: 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 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.
- Optimizing SSA form; non-live vars don’t need Φ-functions.

Liveness analysis: Concepts

Def sets

The **def set** *def*(n) of a node n is the set of variables that are defined in n (a set with 0 or 1 elements).

Use sets

The **use set** *use*(n) of a node n is the set of variables that are used in n.

Live-out sets

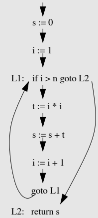
The **live-out set** *live-out*(n) of a node n is the set of variables that are live at an out-edge of n.

Live-in sets

The **live-in set** *live-in*(n) of a node n is the set of variables that are live at an in-edge of n.

An example

1st example revisited



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

Live-in sets

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

$$\begin{aligned} \textit{live-out}(n) &= \cup_{s \in \textit{succs}(n)} \textit{live-in}(s). \\ \textit{live-in}(n) &= \textit{use}(n) \cup (\textit{live-out}(n) - \textit{def}(n)). \end{aligned}$$

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 = $\cup_{s \in succs(n)} live\text{-}in[s]$
    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 (to be completed in class).
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.
  (Since the program contains a loop, this is just a heuristic).

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; one should arrange for *succs* to be fast.
- For sets of variables one may use bit arrays with one bit per variable. Then union is bit-wise or, intersection bit-wise and 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 nr of nodes and V nr of variables. In practice, for realistic code, the number of iterations is much smaller.

Node ordering

A heuristically good order can be found by doing a depth-first search of the CFG and reversing the node ordering.

Basic blocks

Motivations

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

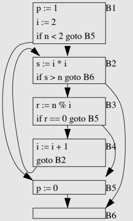
Definition

- 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



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 CFG graphs 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

Def sets

The **def set** *def*(n) of a node n in a CFG is the set of variables that are defined in an instruction in n.

Use sets

The **use set** *use*(n) of a node n is the set of variables that are used in an instruction in n **before** a possible redefinition of the variable.

Live-out sets

The **live-out set** *live-out*(n) of a node n is the set of variables that are live at an out-edge of n.

Live-in sets

The **live-in set** *live-in*(n) of a node n is the set of variables that are live at an in-edge of n.

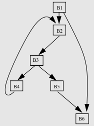
Another dataflow problem: dominators

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



Questions

- Write dataflow equations for dominance.
- How would you solve the equations?

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
}
```

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
}
```

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 -std-compile-opts` for a default selection.

What now?

- Next Tuesday: Last lecture; more on optimization.
- Book time for oral exam; see course web site.