Advanced C Programming

Compilers

Sebastian Hack
hack@cs.uni-sb.de

Christoph Weidenbach
weidenbach@mpi-inf.mpg.de

20.01.2009

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Contents

Goals

- Get an impression of what compilers can do
- Write programs in a way such that compilers can optimize them well

- Get an impression of what compilers cannot do
- Do some important optimizations by hand

Compilers

Architecture

- Syntactic / semantic analysis of the input program
- Dependent on the programming language

- Heart of the compiler
- Independent from language and target architecture
- most optimizations implemented here

- Transform the program to machine code
- Dependent on target architecture
- Implement resource constraints of machine/runtime-system

Optimizations

- Optimization is the wrong word
- It is a mathematical term describing the task of solving an optimization problem
- Compiler "optimizations" merely transform the program
  - Should thus be called transformations
  - We call them optimizations anyway ☺
- Many interesting optimizations are NP-complete or uncomputable
- Since compilation speed also matters:
  - Much in compilers is about finding fast heuristics for extremely difficult problems
- Challenging engineering task:
  - Very diverse inputs
  - Complex data structures
  - Complex invariants
  - No tolerance of failure: Must work for every input

Optimizations

- Compiler writers have a mathematically provable job guarantee
- The full employment theorem

Optimizations

- Compiler writers have a mathematically provable job guarantee
- The full employment theorem

Given: A program P that does not emit anything
Wanted: The smallest binary for P

Theorem

There exists no compiler that can produce such a binary for every P

Optimizations

- Compiler writers have a mathematically provable job guarantee
- The full employment theorem

Given: A program P that does not emit anything
Wanted: The smallest binary for P

Theorem

There exists no compiler that can produce such a binary for every P

Proof.

If P does not terminate, its smallest implementation is

$$L1 : \texttt{jmp}\ L1$$

To this end, the compiler must determine whether P holds. □

Program Representations

- Compilers process data like any other program
- However, the data they process are programs
- To get an idea of what compilers can do, we need to understand how they represent programs
- Every "end" uses its own intermediate representation (IR)
- The effectiveness of many optimizations are dependent on the degree of abstraction and the shape of the IR
- Most compilers use ≤ 4 IRs

Program Representations

Front End

- Abstract Syntax Tree (AST)
- Program represented by syntactical structure
- Basically a large tree and a name table
- Nodes represent type of structural entity:
  Function, Statement, Operator, ...
- Mainly used for:
  - Name resolution
  - Type checking
  - High-level transformations
    (loop transformations)

Program Representation

AST

Source

```
int sum_upto(int n) {
    int i, res = 0;
    for (i = 0; i < n; ++i)
        res += i;
    return res;
}
```

AST

```
FUNCTION_DECL name:sum_upto
  ARG name:n type:int
  BODY
    STATEMENT_LIST
      VAR_DECL name:i type:int
      ...
      FOR_LOOP
        ASSIGN
          VAR_EXPR Name:i
          CONST_EXPR Value:0
        CMP_EXPR Op:<
          VAR_EXPR Name:i
          VAR_EXPR Name:n
       ...
```

Program Representations

"Middle" to Back End

- Control-Flow Graphs (CFG)
  - High-level control structures (for, if, ...) gone
  - Nodes of the CFG: Basic Blocks
  - Edges represent flow of control
- Instructions in a basic block are in "triple form"
  - Each instruction has the form

$$z \leftarrow op(x_1, \ldots, x_n) \qquad \text{Often: } n = 2$$

  - No expression trees anymore
  - Notion of a statement no longer present
  - $z, x_1, \ldots, x_n$ scalar variables ☞ machine types

Definition (Basic Block)

A basic block B is a maximal sequence of instructions $I_1, \ldots, I_n$ for which

1. I_i is a control-flow predecessor of I_{i+1}
2. If I_i is executed so is I_j

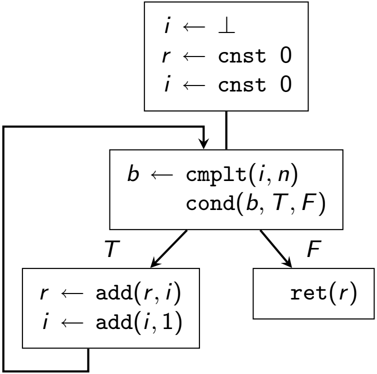
Program Representation

CFG/Tripe-Code

Source

```
int sum_upto(int n) {
    int i, r = 0;
    for (i = 0; i < n; ++i)
        r += i;
    return r;
}
```

Tripe-code CFG



Program Representations

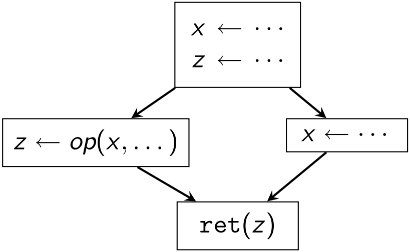
Back End

- Nowadays similar to middle end:
  - CFGs with machine instructions
  - Registers instead of variables
- At the very end, a list of assembly instructions is generated
- CFG is flattened
- Flattening important:
  - Use fall-throughs ☞ safe jump instructions
  - Arrange blocks carefully to aid branch prediction
- Other “minor” stuff to care about:
  - Instruction encoding
  - Alignment
  - Data Layout
  - ...

Contents

Dead Code Elimination

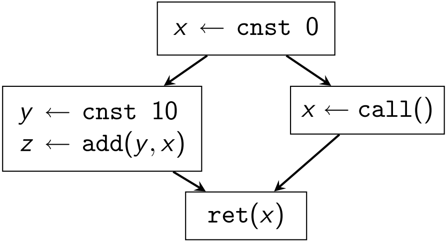
- Eliminate Code which has no effect
- Must not be written by the user
- Can also result as "garbage" from other transformation



- Definition of x in right branch is dead
- the value computed there will never be used
- How to find dead computations?
- Data-flow analysis

Constant Folding

- Compute constant expressions during compile time



- Addition in left block can be optimized to

$$z \leftarrow \texttt{cnst}\ 10$$

- The use of x in the bottom cannot
- x has unknown contents when coming from the right branch
- Again, use data-flow analysis to determine whether variable has known constant contents

Static Single Assignment (SSA)

- Performing data-flow analyses all the time is laborious
- Each time the program changes, analysis information has to be updated
- Both transformations needed following information:

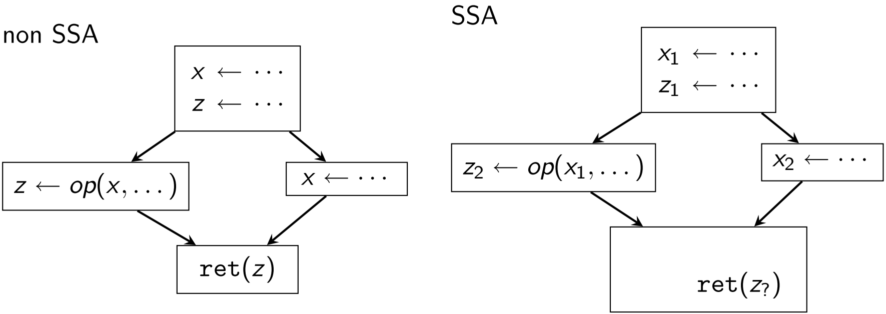
Reaching Definitions

For a use of a variable x, which are the definitions of x that can write the value read at the use of x

- Solution:
  - Encode this directly in the IR
- Allow every variable to only have one instruction that writes its value
- At each use of that variable there is exactly one definition reaching
- Variables and program points are now identical

Dead Code Elimination

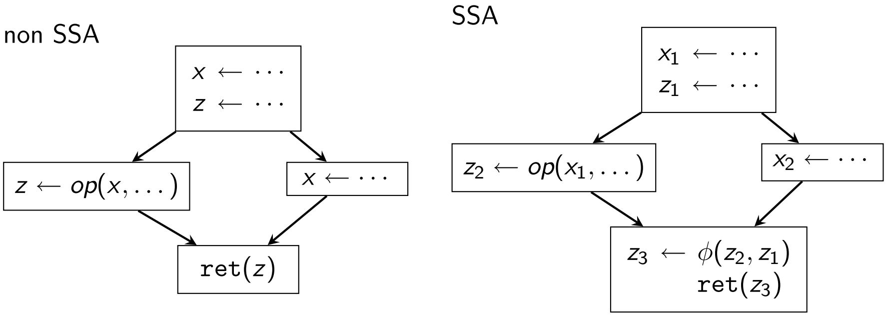
Revisited — SSA



- Which z is used at the return?

Dead Code Elimination

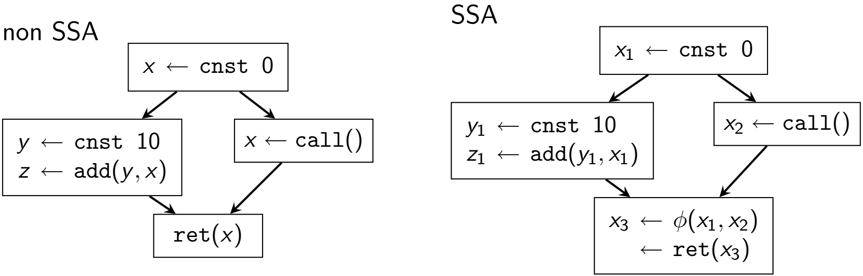
Revisited — SSA



- Which z is used at the return?
- Use ϕ-functions to propagate SSA variables over control flow
- Each variable which has no use is dead (x_2)
- Use that criterion transitively

Constant Folding

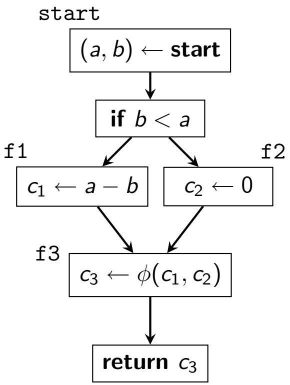
Revisited — SSA



- Each variable has only one definition
- Either the value at the definition was constant or not
- we see that x_3 is not constant because not all arguments of the ϕ are constant

SSA

. . . is functional programming (Kelsey 1995)




```
fun start a b = if   b < a
                 then f1 a b
                 else f2

fun f1 a b     = let c = b-a
                 in  f3 c

fun f2         = f3 0

fun f3 c       = c
```

- Each block is a function
- In FP each variable can be bind only once (here we go!)
- Control flow modeled by function evaluations

Contents

Scalar Variables, Memory, and State

- Up to now all variables are “scalar”:
  - resemble machine types (int, float, double), no arrays or structs
- And all variables were “alias-free”:
  - each variable was only accessible by a single name
- Every modification of the variable happened through that name
- Under SSA this is equivalent to the variable concept in FP
- In FP there is no difference between the name and the variable
- Scalar, alias-free variables are good for code generation
  - They can be put into a register

Scalar Variables, Memory, and State

- Up to now all variables are “scalar”:
  - resemble machine types (int, float, double), no arrays or structs
- And all variables were “alias-free”:
  - each variable was only accessible by a single name
- Every modification of the variable happened through that name
- Under SSA this is equivalent to the variable concept in FP
- In FP there is no difference between the name and the variable
- Scalar, alias-free variables are good for code generation
  - They can be put into a register

- What about non-scalar variables?
- What about variables referenced by pointers?
- We are able to reference the same variable through different names
- In imperative programming names and variables are not the same
- This makes life much harder for the compiler

Scalar Variables, Memory, and State

How are non-scalar variables implemented?

- Arrays
  - Arrays define potentially aliased variables
  - Each array element can be accessed by an indexing expression
  - The value of the index expression might not be known at compile time
  - To disambiguate two accesses `a[i]` and `a[j]`, need to prove `i` $\neq$ `j`
- Structs
  - ... are simpler
  - Unless the address of an element is taken, they can be "scalarized"

```
int foo(void)
    vec3_t vec;
    ...
}
```

```
int foo(void)
    float x, y, z;
    ...
}
```

Scalar Variables, Memory, and State

Aliased Variables

```
int global_var;
int foo(int *p) {
    global_var = 2;
    *p = 3;
    return global_var;
}
```

- We cannot optimize to

```
return 2;
```

- p might point to global_var
- global_var is potentially aliased
- How can we find out?
- Look at all callers of foo
- and the passed argument
- Thus: probably also all the callers of the callers and so on
- What, if we do not know all the callers

Scalar Variables, Memory, and State

Aliased Variables

```
int global_var;
int foo(int *p) {
    global_var = 2;
    *p = 3;
    return global_var;
}
```

- We can help the compiler
- If the address of `global_var` is never taken
- and we defined it as `static`
- it can only be modified by functions in the current file
- And never through a second name
- It cannot be aliased
- Be as precise as possible with your declarations

Scalar Variables, Memory, and State

- We "implement" aliased variables by a global memory (Of course it is the other way around ☺)
- This memory belongs to the state
- The main difference between functional and imperative programming is the presence of state
- What else belongs to the state is a question of the programming language's semantics
- How that state is updated is (mostly) decided by the memory model
- For correct compilation, the visible effects on the state and their order have to be preserved
- Both are defined in the PL's semantics
- How do we model the state in the IR?

Scalar Variables, Memory, and State

Representation of Memory

- The memory is also represented as an SSA variable
- Each load and store reads takes a memory variable and gives back a new one

```c
int global_var;
int foo(int *p) {
    global_var = 2;
    *p = 3;
    return global_var;
}
```

$$
\begin{aligned}
c_1 &\leftarrow \texttt{cnst}\ 2 \\
c_2 &\leftarrow \texttt{cnst}\ 3 \\
M_1 &\leftarrow \texttt{getarg}(0) \\
a &\leftarrow \texttt{symcnst}\ \text{global\_var} \\
M_2 &\leftarrow \texttt{store}(M_1, a, c_1) \\
p &\leftarrow \texttt{getarg}(1) \\
M_3 &\leftarrow \texttt{store}(M_2, p, c_2) \\
(M_4, r) &\leftarrow \texttt{load}(M_3, a) \\
& \quad\ \texttt{ret}(M_4, r)
\end{aligned}
$$

- Memory is treated functionally
- Similar to the concept of a monad, cf. Haskell

Scalar Variables, Memory, and State

Representation of Memory

- We can also have multiple memory variables!
- They must however be implemented with the single memory we have
- We must make sure that they represent pairwise disjoint variables

  $$\begin{aligned} M_2 &\leftarrow \texttt{store}(M_1, p, v) \\ M_3 &\leftarrow \texttt{store}(M_1, q, w) \end{aligned}$$

  does only work if $p \neq q$
- Benefit:
  - Variables may be scalarized in some regions of the code
  - order of memory accesses can be changed
    ☞ important for code generation

Scalar Variables, Memory, and State

Points-to Analysis

- Subdivision of memory needs results of points-to analysis
- For each use of a pointer determine an (over-approximated) set of variables the pointer might point to
- One of the hardest analyses
  - interprocedural (whole-program)
  - long runtime, large memory consumption
- Do not count on it
- Many compilers make precision sacrifices to safe compilation time

Summary

- Scalar, alias-free variables are good!
- Many analyses are easy for them
- Most optimizations only work on scalar, alias-free variables
- They can be allocated to a processor register
- Having many scalar variables is no problem
  - The register allocator will decide which ones to spill where

- Know that accesses to non-scalar variables might result in memory accesses
- Always program as scalar as possible
- Always convey as much information as possible
- Do not overly rely on points-to analysis

Being Scalar “Best Practices”

Arrays

Prefer

```c
typedef struct {
    float x, y, z, w;
} vec_t;
```

Over

```c
typedef float vec_t[4];
```

- Compiler might have trouble analysing indexing expressions
- `a.x` is much clearer
- Can be scalarized more easily
- Some compilers do not consider arrays for scalarization

Being Scalar "Best Practices"

Arrays

Prefer

```
int x = p[i];
int y = p[i + 1];
int z = p[i + 2];
```

Over

```
int q = p + i;
int x = *p++;
int y = *p++;
int z = *p++;
```

- Array base pointer stays the same
- Inequality of indexing often easier to analyze than the pointer update
- Compiler will do that transformation itself if he knows that he can save a register

Being Scalar “Best Practices”

Avoid pointer dereferencing

Prefer

```
void isqrt(unsigned long a,
           unsigned long *q,
           unsigned long *r)
{
  unsigned long qq, rr;
  qq = a;
  if (a > 0) {
    while (qq>(rr=a/qq)) {
      qq = (qq + rr) >> 1;
    }
  }
  rr = a - qq * qq;
  *q = qq;
  *r = rr;
}
```

Over

```
void isqrt(unsigned long a,
           unsigned long *q,
           unsigned long *r)
{
  *q = a;
  if (a > 0) {
    while (*q>(*r=a/ *q)) {
      *q = (*q + *r) >> 1;
    }
  }
  *r = a - *q * *q;
}
```

- The left version makes explicit that we assume $\mathtt{q} \neq \mathtt{r}$
- In C99 you could use `restrict`
- But then you rely on the compiler to do it right
- If all these memory accesses stay, performance is worse
- Treat memory accesses like reading from file