Introduction to the LLVM Compiler Framework

SS 2011

Christian Plessl

Paderborn Center for Parallel Computing
University of Paderborn

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Version 1.0.1 – 26.04.2011

Motivation & Outline

- brief overview of a state of the art compiler framework
  - we are using LLVM in our research
  - we will use it as an example in the lecture and in the exercises

- outline
  - overview of the LLVM compiler framework
  - compilation tool flows
  - LLVM intermediate representations
  - optimizations
  - code generation

What is the LLVM Compiler Framework

- modern open-source compiler infrastructure
  - implemented in C++
  - modular and extensible design
  - combines a static compilation tool flow with a virtual machine
- many supported front-ends/languages
  - C, C++, Objective-C (Clang, GCC/dragonegg)
  - Ruby (Rubinius, MacRuby)
  - Python (unloaden swallow)
  - and many more
- many supported CPU architectures in backend
  - ARM, Alpha, Intel x86, Microblaze, MIPS, PowerPC, SPARC, …
- very popular and widely used
  - Apple, AMD, NVidia, Cray, Google, …

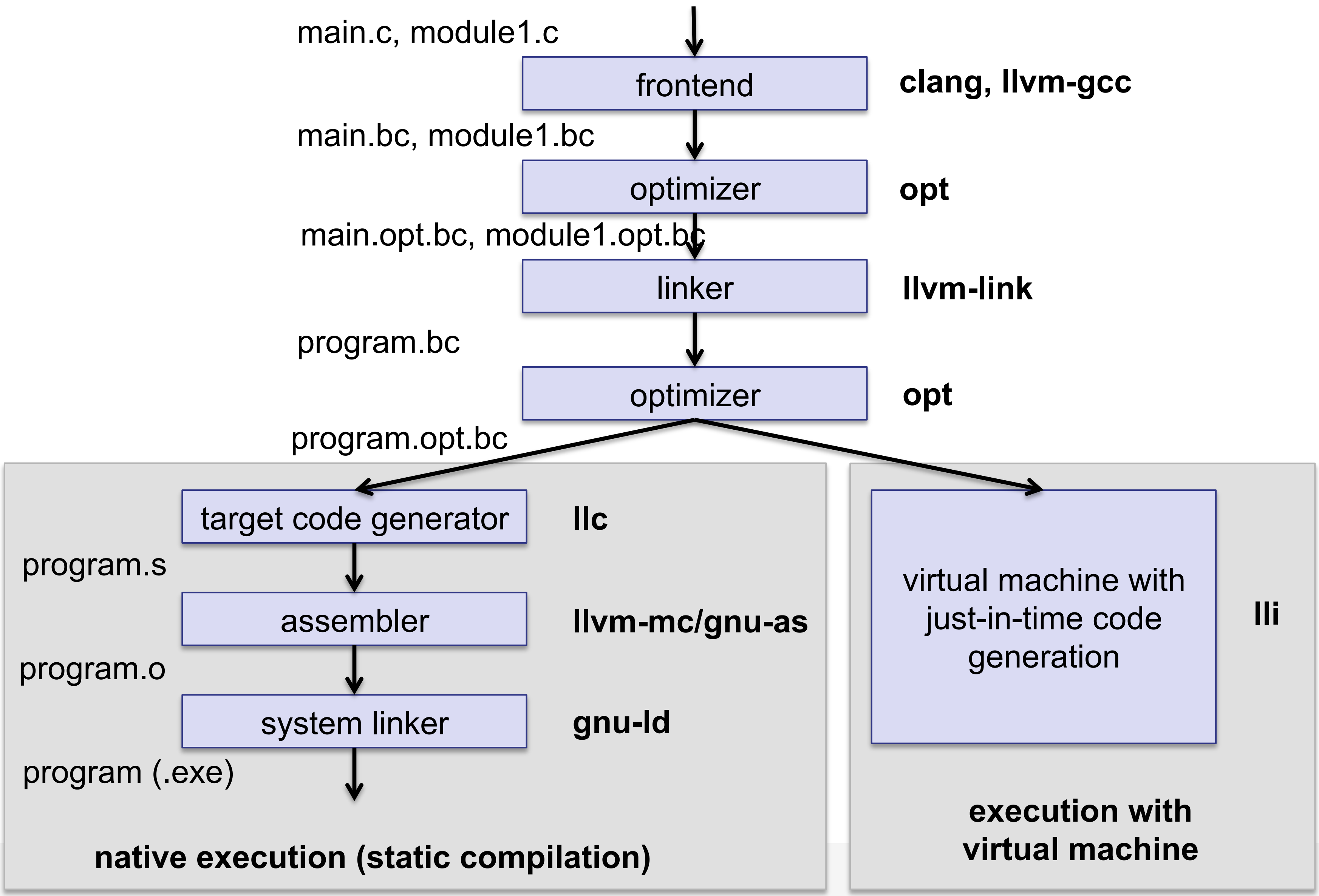
LLVM Design Principle

- separation of the compilation process in frontend / analysis and transformation / backend

- LLVM intermediate representation (LLVM IR) plays a central role in this process
  - all code optimizations are implemented as “LLVM IR to LLVM IR transformation passes”
  - code analysis is also implemented as pass, generated results can be shared between passes

- all target processor-specific optimizations are handled in the backend

Static LLVM Compilation Toolflow



LLVM Intermediate Representation

- basis for all LLVM optimization passes
- low-level assembly language for a “virtual” processor
  - load/store architecture
  - infinite amount of named registers
  - each register is assigned exactly once (static single assignment, SSA)
- exists in three equivalent representations
  - in-memory C++ data structures
  - binary files (LLVM bitcode) (file extension: .bc)
  - human readable assembly notation (file extension: .ll)
- reference
  - http://llvm.org/docs/LangRef.html

LLVM IR Example

```
double circle_area(double radius) {
  return radius * radius * 3.14159265;
}
```

C code

compiling with Clang frontend and LLVM

clang -S -emit-llvm circle_area.c -o circle_area.ll
opt -mem2reg -instsimplify -S circle_area.ll -o circle_area.opt.ll

```
define double @circle_area(double %radius)
nounwind ssp {
  %1 = fmul double %radius, %radius
  %2 = fmul double %1, 0x400921FB53C8D4F1
  ret double %2
}
```

LLVM IR code

note:

- strongly typed data types
- each variable only assigned once (static single assignment)

Running optimizations passes on LLVM IR

C code

```
int test1(int x, int y, int z) {
  return (x & z) ^ (y & z);
}
```

example

- compile this code to LLVM IR
- run series of optimization passes to iteratively improve code

Running optimizations passes on LLVM IR 2

Compiled with Clang (no optimizations)

clang -S -emit-llvm boolean_optimization.c -o boolean_optimization.ll

```
define i32 @test1(i32 %x, i32 %y, i32 %z)
nounwind ssp {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  store i32 %y, i32* %2, align 4
  store i32 %z, i32* %3, align 4
  %4 = load i32* %1, align 4
  %5 = load i32* %3, align 4
  %6 = and i32 %4, %5
  %7 = load i32* %2, align 4
  %8 = load i32* %3, align 4
  %9 = and i32 %7, %8
  %10 = xor i32 %6, %9
  ret i32 %10
}
```

properties of the generated code

- correct, but obviously very inefficient
- each function argument is written to the stack first …
- … and read from the stack later on

Running optimizations passes on LLVM IR 3

First optimization (mem2reg, instsimplify)

opt -mem2reg -instsimplify -S boolean_optimization.ll -o boolean_optimization.opt.ll

```
define i32 @test1(i32 %x, i32 %y, i32 %z)
nounwind ssp {
  %1 = and i32 %x, %z
  %2 = and i32 %y, %z
  %3 = xor i32 %1, %2
  ret i32 %3
}
```

properties of the generated code:

- removed redundant instructions
- used registers instead of stack memory
- instructions of the actual computation remain unchanged

can this code be simplified any further?

Running optimizations passes on LLVM IR 4

Second optimization: instcombine

opt -mem2reg -instcombine -S boolean_optimization.ll -o boolean_optimization.opt.ll

```
define i32 @test1(i32 %x, i32 %y, i32 %z)
nounwind ssp {
  %1 = xor i32 %x, %y
  %2 = and i32 %1, %z
  ret i32 %2
}
```

properties of the generated code:

- further simplification of the code
- instcombine not only removes redundant instructions but changes instructions
- optimization pass did understand the semantics of the boolean operations and figured out that *(x and z) xor (y and z) == z and (x xor y)*

numerous additional optimizations available, consult opt manual page for details

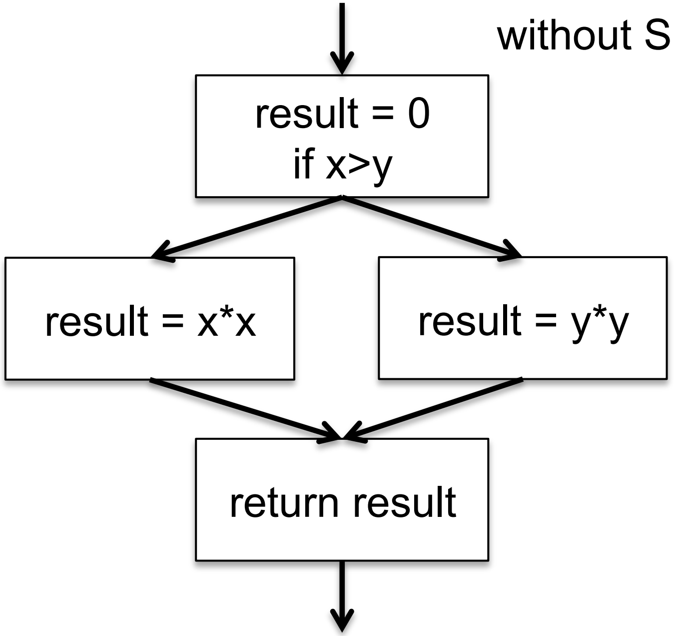
Static Single Assignment 1

- LLVM IR uses static single assignment (SSA) form
  - each virtual register is assigned only once
  - allows to easily track define-use chains, i.e. what values are used by which instructions (useful e.g. for dead code elimination)
- what happens if we need to assign a register several times, e.g. in a loop or in branches?
- example

```
int max_square(int x, int y)
{
  int result = 0;

  if (x>y){
    result = x*x;
  } else {
    result = y*y;
  }
  return result;
}
```

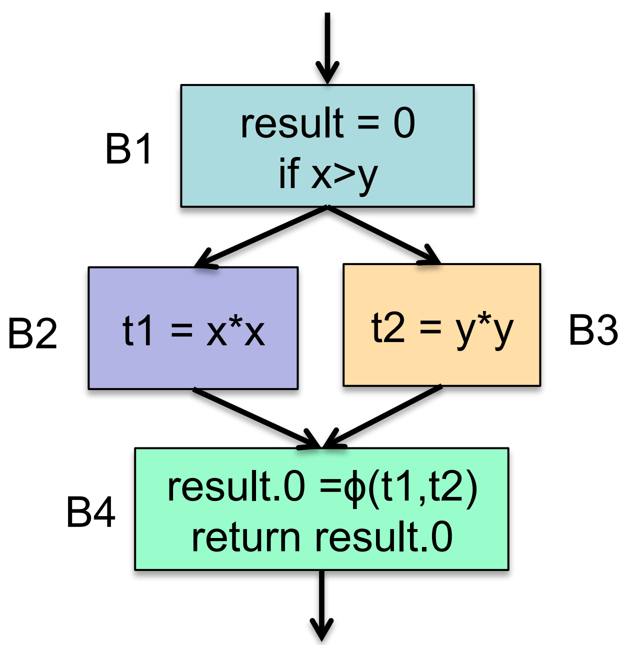
control flow graph without SSA



Static Single Assignment 2

- use of phi-nodes/instructions (ϕ)
  - phi nodes keep track which control-flow path was taken and use the corresponding value (like a multiplexer)
  - not actually implemented, compiler just makes sure that the virtual registers are mapped to the same physical register

control flow graph with SSA




```
define i32 @max_square(i32 %x, i32 %y) {
  %1 = icmp sgt i32 %x, %y
  br i1 %1, label %2, label %4
```

```
; <label>:2
  %3 = mul nsw i32 %x, %x
  br label %6
```

← beginning and name of basic block

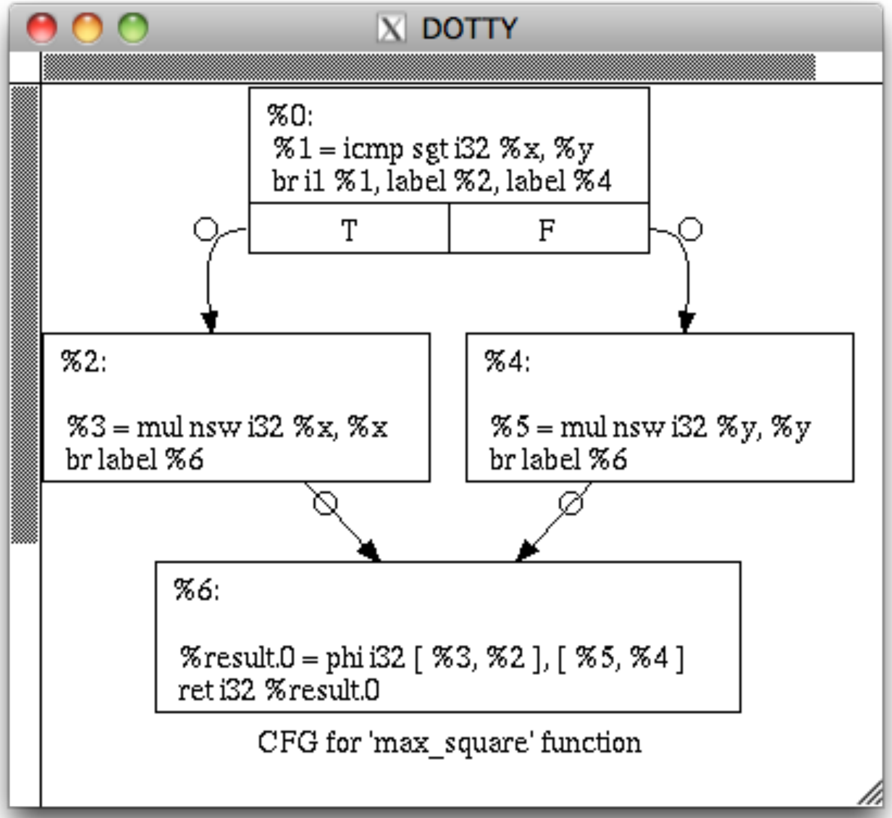
```
; <label>:4
  %5 = mul nsw i32 %y, %y
  br label %6
```

```
; <label>:6
  %result.0 = phi i32 [ %3, %2 ], [ %5, %4 ]
  ret i32 %result.0
}
```

choose reg %3 if control flow enters from BB %2,
choose reg %5 if control flow enters from BB %4

Visualizing Control Flow Graphs

- LLVM has built-in support for visualizing various steps in the compilation process



opt -view-cfg -S phi.opt.ll

- example: using different backends, compilation for MIPS and ARM instruction set

```
max_square:
# BB#0:
        addiu   $sp, $sp, -16
        slt     $2, $5, $4
        beq     $2, $zero, $BB0_2
        nop
# BB#1:
        mult    $4, $4
        j       $BB0_3
        nop
$BB0_2:
        mult    $5, $5
$BB0_3:
        mflo    $2
        addiu   $sp, $sp, 16
        jr      $ra
        nop
```

MIPS assembler code

```
_max_square:    @ @max_square
@ BB#0:
        cmp     r0, r1
        mulle   r2, r1, r1
        mulgt   r2, r0, r0
        mov     r0, r2
        bx      lr
```

ARM assembler code

llc --march=mips phi.ll -o phi.mips.s
llc --march=arm phi.ll -o phi.arm.s

Summary

- LLVM is a modern open source compiler framework
  - very powerful and easy to use
  - human readable IR allows for following optimization steps
  - modular design allows adding own functionality
- LLVM may also be of practical use for you
  - as a replacement for GCC
  - for generating code for embedded processors
  - for learning about compilers and optimizations
  - building your own programming language (frontend) that uses LLVM as a backend (search the web for inspiration)
- acknowledgement
  - this presentation is based partly on materials that have been kindly provided by Tobias Grosser (http://grosser.es/), visit his website for more information on LLVM

Changes

- 2011-05-05 (v1.0.1)
  - fix a couple of minor typos