G53CMP: Lecture 19

LLVM: A Real Compiler Backend

Henrik Nilsson

University of Nottingham, UK

Result SEM G53CMP 2018/19 (1)

Scale: 5 is agree/positive; 1 is disagree/negative.

#	Question	G53CMP	All modules
1	Opportunities to explore	4.17	4.26
2	Challenged me to deliver	4.22	4.09
3	Well organised	4.33	4.18
4	Resources helpful	3.39	3.97
5	Clear marking criteria	3.78	3.86
6	Reasonable workload	3.44	0.98(?)
7	Overall satisfied	3.89	0.98(?)

Result SEM G53CMP 2018/19 (2)

- On the whole, you were happy with the module, but less so than last year (4.53). The marked drop was a surprise to me.
  - Reservations about workload (4.29 last year)
    - Coursework weight effectively 50%
    - 100 h?
  - Resource score also dropped (Cf. 3.98)
- A lot of good feedback. Will be taken aboard!

Result SEM G53CMP 2018/19 (3)

Emerging themes:

- Writing compiler from scratch? Targeting something “real”?
  Difficult balance between e.g.:
  - work load
  - providing an opportunity to study and work with something not too unrealistic
  - covering all key aspects (incl. type checking)
  - ease of debugging
  - freedom of exploring

Result SEM G53CMP 2018/19 (4)

Emerging themes:

- Too large topic for a 10 credit module

  Perhaps extend to 20 credit module?

LLVM (1)

LLVM (formerly Low Level Virtual Machine) is a ***compiler infrastructure project***:

- Highly modular and extensible; at its core:
  - Set of reusable libraries
  - Well-defined interfaces
- Designed for static and dynamic (JIT) compilation and optimzation: compile-time, link-time, load/installation-time, run-time.
- Language agnostic: LLVM-based compilers for Ada, C, C++, Fortran, Haskell, Java bytecode, OpenGL Shading Language, Python, Scala.

LLVM (2)

Some background:

- The LLVM project started in 2000 at the University of Illinois at Urbana-Champaign.
- Directed by Vikram Adve and Chris Lattner.
- Lattner later hired by Apple Inc.
- LLVM integral part of Apple’s development tools for OS X and iOS.
- LLVM is Open Source.
- Adve, Lattner, and Evan Cheng awarded the ACM Software System Award for LLVM in 2012.

Motivations for LLVM (1)

When LLVM started:

- Open-source language implementations tended to be monolithic; e.g. GCC:
  - Extremely hard to reuse individual parts
  - Not even a self-contained intermediate representation
- Implementations tended to either support static or JIT compilation.
- The text-book vision of multiple independent front-ends and back-ends around a shared compiler core hardly ever realised in practice.

LLVM IR (1)

The LLVM ***Intermediate Representation*** (IR) is the “glue” that holds LLVM together.

- Complete, ***self-contained*** representation: a first-class language with well-defined semantics.
- Designed to host mid-level analyses and transformations.
- ***RISC-like*** code.
- Sufficiently low-level to be a suitable translation target for any language.

LLVM IR (2)

- Sufficiently high-level to allow targeting arbitrary concrete architectures; e.g.:
  - Unbounded number of registers
  - Abstraction over calling conventions
- ***Typed***:
  - Base types: integers (of different sizes), floating point numbers
  - Derived types: pointers, arrays, vectors, structures, functions
- ***Static Single Assignment*** (SSA): SSA form for all scalar registers (everything except memory).

LLVM IR (3)

- Three isomorphic forms:
  - Textual format (`.ll`)
  - Compact, on-disk, "bitcode" format (`.bc`)
  - In-memory data structure.

  Some tools:
  - `llvm-as`: `.ll` ⇒ `.bc`
  - `llvm-dis`: `.bc` ⇒ `.ll`

LLVM Modularity (1)

- Each LLVM ***pass***, such as optimizations, is a library component ***transforming*** LLVM IR; e.g.
  - constant folding
  - loop unrolling
  - motion of loop-invariant code
  - inliner
- Passes are written to be as independent as possible; any dependences are declared explicitly.

LLVM Modularity (2)

- A ***pass manager*** can run the available passes in a suitable order, subject only to declared constraints.
- Any particular application only needs to include exactly those passes that are relevant, making for small footprint.

SSA Form (1)

Static Single Assignment (SSA):

- SSA form is a ***property*** of intermediate representations where:
  - each variable is assigned ***exactly*** once
  - every variable is defined (assigned) before used.
- Developed at IBM in the 1980s by researchers Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, Kenneth Zadeck.
- Compilers using SSA include: GCC, LLVM, Oracle's HotSpot JVM, Android's Dalvik and Runtime.

SSA Form (2)

- SSA form can for ***some*** purposes be seen as a ***purely functional*** representation.
- Indeed, there is a ***formal correspondence*** between SSA form and purely functional representations, notably Continuation Passing Style (CPS).
- As a result, many compiler optimizations are simplified and improved.

Static Single Assignment (SSA) Form (3)

Conversion to SSA form by splitting each variable into ***versions***. For example:

$y := 1;\ y := 2;\ x := y$

In SSA form:

$y_1 := 1;\ y_2 := 2;\ x_1 := y_2$

Note that it now is ***manifest*** (no flow analysis needed) where the value assigned to x comes from and that the first assignment to y is dead code.

What about Control Flow Joins? (1)

The obvious question is how to handle joins in the control flow.

Consider:

Before SSA conversion:	SSA form:
x := ...;	x_1 := ...;
if $x > 0$ then	if $x_1 > 0$ then
$x := 1$	$x_2 := 1$
else	else
$x := 2;$	$x_3 := 2;$
$y := x;$	$y_1 := x_{???};$

What about Control Flow Joins (2)

Or consider:

Before SSA conversion:	SSA form:
x := ...;	x_1 := ...;
while $x < 100$ do	while $x_{???} < 100$ do
$x := x * 2;$	$x_2 := x_{???} * 2;$
$y := x$	$y_1 := x_{???}$

ϕ-Functions (1)

A ϕ-function (originally "phoney function") selects and returns ***exactly one*** of its arguments.
Assume first it always picks the "right" argument.
Then we can solve our dilemma as follows:

```
x₁ := ...;
if x₁ > 0 then
    x₂ := 1
else
    x₃ := 2;
x₄ := φ(x₂,x₃);
y₁ := x₄
```


ϕ-Functions (2)

And:

```
x₁ := ...;
while (x₂ := φ₁(x₁,x₃), x₂ < 100) do
    x₃ := x₂ * 2;
y₁ := x₂
```

Also clearly in SSA form!

ϕ-Functions (3)

- A ϕ-function selects an argument according to the ***dynamically*** preceding basic block: from ***where*** did the control reach the ϕ-function?
- "Translating out of" SSA is essentially a matter of joining up the different versions of a variable.
- A ϕ-function translates into ***no code*** if the arguments and results can be stored in the same place (register).
- Otherwise extra copy instructions (assignments) are needed to translate out of SSA.

Where Do ϕ-Functions Go? (1)

ϕ-functions are placed by constructing and analysing the ***control flow graph***:

- A node A ***strictly dominates*** a different node B iff all paths to B go through A.
- A node A ***dominates*** B iff A strictly dominates B or $A = B$.
- A node B is in the ***dominance frontier*** of a node A iff A does ***not*** strictly dominate B, but does dominate an ***immediate predecessor*** of B.

ϕ-functions are placed on the dominance frontier.

Where Do ϕ-Functions Go? (2)

Observation: we only need ϕ-functions for ***live*** variables.

- Pruned SSA: Use live-variable information to decide whether a particular ϕ-function is needed. Expensive computation.
- Semi-pruned SSA: Identify variables that are ***never*** live on entry to a block and omit ϕ-functions for such "block-local" variables. Cheaper to compute.

LLVM Demo

We will translate the following C-code into LLVM IR using the Clang compiler, study the result and run some optimizations on it.

```
int i, m, n;
int main(int argc, char* argv[]) {
    sscanf(argv[1], "%d", &m);
    for (i = 0; i < m; i++) {
        n += i;
    }
    printf("n = %d\n", n);
    return 0;
}
```