Use of the LLVM framework for the MSIL code generation

Artur PIETREK
artur.pietrek@imag.fr

VERIMAG
Kalray (Montbonnot)

DCS seminar
March 27, 2009

PhD thesis

The MSIL code generator is a part of the thesis:

CLI JIT compilation for media processing applications

advisors:

Jean-Claude Fernandez, VERIMAG
Benoît Dupont de Dinechin, Kalray

What is MSIL?

Microsoft Intermediate Language also known as Common Intermediate Language (CIL)

- the lowest-level part of the CLI (Common Language Infrastructure)
- object-oriented assembly language
- platform independent assembly language
- stack-based program representation

Why MSIL?

- along with LLVM "bit-code", the only program representation for Just-In-Time compilation of C programs
- MSIL is mature and has become ECMA standard as a part of CLI (*Common Language Infrastructure*)
- several different Virtual Machines as reference (MS .NET, MS Rotor, Mono, DotGNU)

Motivation

There are few solutions available for generation of the MSIL (Microsoft Intermediate Language) code from C language:

LCC.NET

Princeton LCC + MSIL backend contributed by Microsoft

GCC4NET

Contributed by STMicroelectronics
currently maintained by IRISA (E.Rohou)

LLVM+MSIL

LLVM head (Chris Lattner) agreed that we took ownership of the MSIL code generator

Motivation

Generated intermediate code could be Just-In-Time compiled using one of the implementations of .NET CLR (Common Language Runtime).
These solutions don't provide mechanisms for runtime and offline optimization of programs written in C.

Motivation

Microsoft .NET

Windows

Microsoft Rotor

Windows (XP SP2), FreeBSD, Mac OS X

DotGnu

GNU/Linux, *BSD, Cygwin/Mingw32, Mac OS X, Solaris, AIX

Mono

GNU/Linux, *BSD, Mac OS X, iPhone OS, Solaris, Windows, Nintendo Wii, Sony PlayStation 3

Motivation

As I am interested in the MSIL code generation and Just-In-Time compilation, my goal is to develop a Just-In-Time compiler that will allow for:

- JIT compilation programs originally written in C
- life-long optimization of those programs
- **program specialization based on static and dynamic profiles**

What is LLVM?

LLVM stands for

Low Level Virtual Machine

What is LLVM?

A compilation strategy.

- lifelong analysis and transformation of a program:
  - compile-time
  - link-time
  - install-time
  - run-time
  - idle-time
- use aggressive interprocedural optimizations
- gather and exploit end-user profile information
- tune the application to the user‘s hardware
- generate the native code off-line and run-time

What is LLVM?

A virtual instruction set.

- RISC-like instructions
- low-level object representation
- type information and dataflow informations about operands
- exceptions handling
- source language independent representation

What is LLVM?

A compiler infrastructure.

- implementation of languages and compilation strategy
- optimization and analysis framework
- static backends for X86, X86-64, PowerPC 32/64, ARM, Thumb, IA-64, Alpha, SPARC, MIPS and CellSPU architectures
- Just-In-Time compiler for X86, X86-64, PowerPC 32/64
- portable code generator: C, C++, MSIL

Why LLVM?

- among others, has a frontend for C language
- has an aggressive optimizer
  - scalar
  - interprocedural
  - profile-driven
  - simple loop optimizations
- supports a life-long optimization model
  - link-time
  - install-time
  - run-time
  - off-line
- allows for relatively easy implementation of transformations (optimizations) and targets

Why LLVM?

- LLVM is under active development
- LLVM is freely available under BSD-like license

LLVM MSIL code generator

- one of the existing LLVM's code generators
- current state is experimental rather than fully functional
- it is not under developement anymore

Main project goals

- use of C language as front-end
- generated code has to work with Mono project and .NET
- generated code should be fully ECMA standard compliant
- implementation of LLVM's tests for validation of the generated code

Discovered problems

- lack of initialization of global variables
- calling *external vararg* functions doesn't work with Mono
- incorrect stack management after calling functions
- lack of implementation of LLVM's intrinsics
- lot of unused local variables
- *switch()* should be implemented as MSIL's equivalent rather than set of labels and branches

Current progress

Fixed

- fixed global variables initialization
- implemented solution for *external vararg* functions and Mono project

Current progress

Fixed

- fixed global variables initialization
- implemented solution for *external vararg* functions and Mono project

Work in progress

- writing LLVM's tests for verification of generated code
- writing LLVM's transformations for remaining problems

Plans for the future

- developement of MSIL code interpreter
- developement of MSIL code JIT compiler
- run-time optimization for developed JIT compiler
- **program specialization based on static and dynamic profiles**

John Gough.
Compiling for the .NET Common Language Runtime (CLR).
Prentice Hall PTR, Upper Saddle River, New Jersey, 2002.

Chris Lattner.
LLVM: An Infrastructure for Multi-Stage Optimization.
Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.
See `http://llvm.cs.uiuc.edu`.

Chris Lattner and Vikram Adve.
LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.
In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

Serge Lidin.
Inside Microsoft .NET IL Assembler.
Microsoft Press, Redmond, Washington, 2002.