Dynamic Compilation using LLVM

Alexander Matz
Institute of Computer Engineering
University of Heidelberg
alexander.matz@ziti.uni-heidelberg.de
Outline

- Motivation
- Architecture and Comparison
- Just-In-Time Compilation with LLVM
- Runtime/profile-guided optimization
- LLVM in other projects
- Conclusion and Outlook
Traits of an ideal compiler

- Fast compilation
- (Compile-link-execute model)
- Platform and source independency
- Low startup latency of resulting executable
- Low runtime overhead
- Aggressive optimization
- Zero-effort adaptation to patterns of use at runtime

No current system has all these traits

LLVM aims to fill the gap
What is LLVM

- LLVM ("Low Level Virtual Machine") consists of:
  - A Virtual Instruction Set Architecture not supposed to actually run on a real CPU or Virtual Machine
  - A modular compiler framework and runtime environment to build, run and, most importantly, optimize programs written in arbitrary languages with LLVM frontend

- Primarily designed as a library, not as a „tool“
Existing Technologies
Existing technologies

- Statically compiled and linked (C/C++ etc.)
- Virtual Machine based (Java, C# etc.)
- Interpreted (Javascript, Perl)
Existing technologies

- **Statically compiled and linked (C/C++ etc.)**
  - Static machine code generation early on
  - Platform dependent
  - Optimization over different translation units (.c files) difficult
  - Optimization at link time difficult (no high level information available)
  - Profile-guided optimization requires change of build model
  - Optimization at run-time not possible at all

- **Virtual Machine based (Java, C# etc.)**

- **Interpreted (Javascript, Perl)**
- Statically compiled and linked (C/C++ etc.)
- Virtual Machine based (Java, C# etc.)
  - Keep high level intermediate representation (IR) for as long as possible
  - “Lazy“ machine code generation
  - Platform independent
  - Allows aggressive runtime optimization
  - Only few (fast) low level optimizations possible on that IR
  - Just-In-Time-compiler has to do all the hard and cumbersome work
- Interpreted (Javascript, Perl)
Existing technologies

- Statically compiled and linked (C/C++ etc.)
- Virtual Machine based (Java, C# etc.)
- Interpreted (Javascript, Perl)
  - No native machine code representation generated at all
  - Platform independent
  - Fast build process
  - Optimizations difficult in general
Architecture and Comparison
LLVM aims to combine the advantages without keeping the disadvantages by

- Keeping a low level representation (LLVM IR) of the program at all times
- Adding high level information to the IR
- Making the IR target and source independent
- Difference to statically compiled and linked languages

- Type information is preserved through whole lifecycle
- Machine code generation is the last step and can also happen Just-In-Time
- Difference to VM based languages

- LLVM IR is not supposed to run on a VM
- IR much more low level (no runtime or object model)
- No guaranteed safety (programs written to misbehave still misbehave)
Benefits

- Low Level IR
- High Level Type Information
- Modular/library approach revolving around LLVM IR
Benefits

- **Low Level IR**
  - Potentially ALL programming languages can be translated into LLVM IR
  - Low level optimizations can be done early on
  - Machine code generation is cheap
  - Mapping of generated machine code to corresponding IR is simple

- **High Level Type Information**

- **Modular/library approach revolving around LLVM IR**
Benefits

- **Low Level IR**

- **High Level Type Information**
  - Allows data structure analysis on whole program
  - Examples of now possible optimizations
    - Pool allocators for complex types
    - Restructuring data types
  - Used in another project to prove programs as safe (Control-C, Kowshik et al., 2003)

- **Modular/library approach revolving around LLVM IR**
Benefits

- Low Level IR
- High Level Type Information
- Modular/library approach revolving around LLVM IR
  - All optimization modules can be reused in every project using the LLVM IR
  - Not limited to specific targets (like x86), see other projects using LLVM
  - Huge synergy effects
Just-In-Time Compilation with LLVM
- Lazy machine code generation at runtime
- All target independent optimizations already done at this point
- Target specific optimizations are applied here
- Supposed to keep both native code and LLVM IR with additional information on mapping between them
- Currently two options on x86 architectures with GNU/Linux
Just-In-Time Compilation with LLVM

- **Clang (LLVM frontend) as drop in replacement for gcc**
  - Results in statically linked native executable (much like with gcc)
  - No LLVM IR kept, no more optimizations after linking
  - Executable performance comparable to gcc
- Clang as frontend only

- Results in runnable LLVM bitcode
- No native code kept, but bitcode still optimizable
- Target specific optimizations are applied automatically
- Higher startup latency
Runtime/profile-guided optimization
Runtime/profile-guided optimization

- All optimizations that can not be predicted at compile/link time (patterns of use/profile)
- Needs instrumentation (=performance penalty)
- Examples for profile-guided optimizations
  - Identifying frequently called functions and optimize them more aggressively
  - Rearranging basic code blocks to leverage locality and avoid jumps
  - Recompiling code making risky assumptions (sophisticated but highest performance gain)
Runtime/profile-guided optimization

- **Statically compiled and linked approach:**
  - Compile-link-execute becomes Compile-link-profile-compile-link-execute
  - In most cases the developers, not the users, profile the application
  - Still no runtime optimization

- **Result:** Profile-guided optimization is skipped most of the time
Runtime/profile-guided optimization

- VM based languages approach:
  - High level representation kept at all times
  - Runtime environment profiles the application in the field without manual effort
  - Hot paths analyzed and optimized (Java HotSpot)
  - Expensive optimizations and code generation compete for CPU cycles with running application
Runtime/profile-guided optimization

- **LLVM approach (goal):**
  - Low level representation is kept
  - Runtime environment profiles the application in the field
  - Cheap optimizations are done at runtime
  - Expensive optimizations are done during idle
### Result (ideal)
- Many optimization are already done on LLVM IR before execution
- Runtime and offline optimizers adapt to use pattern and become more dormant over time
- No additional development effort necessary

### Current limitations (on x86 + GNU/Linux)
- No actual optimization at runtime, JIT-Compiler invoked at startup once and does not adapt to patterns of use during execution
- Profile-guided optimization possible, but only between runs
- Instrumentation needs to be manually enabled/disabled
LLVM in other projects
LLVM in other projects

- Ocelot: Allows PTX (CUDA) kernels to run on heterogeneous Systems containing various GPUs and CPUs
- PLANG: Similar project, but limited to execution of PTX kernels on x86 CPUs
- OpenCL to FPGA compiler for the Viroquant project by Markus Gipp
- Idea: Bulk Synchronous Parallel programming model fits many-core-trend perfectly
  - GPU Applications partitioned without technical limitations in mind (thousands or millions of threads, think of PCAM)
  -Threads are reduced and mapped to run on as many (CPU-)cores as available (<100)
  -Automatic mapping to available cores brings back automatic speedup with newer CPUs/GPUs
Both projects can be seen as LLVM frontends when used for x86 exclusively

Benefits
• „Easy“ implementation, PTX is similar to LLVM IR
• Most optimization can be taken „as is“
• x86 code generation already available

Drawbacks
• Information is lost when transforming PTX to LLVM IR
• Big software overhead due to GPU features not being present in CPUs
OpenCL to FPGA compiler for Viroquant

- Can be seen as LLVM backend

- Benefits
  - Compiler already available (OpenCL treated as plain C)
  - Again, optimizations can be taken as is

- Drawbacks
  - Translation from LLVM IR to VHDL is very complex
Conclusion and Outlook
Mature compiler framework used in many projects and as an alternative to gcc (comparable performance)

Interesting new language independent optimizations

Some important features still missing (for x86)
  • Actual runtime optimization
  • Profile-guided optimization without manual intervention
  • Keeping native code with LLVM IR to reduce startup latency

Not yet the „ideal“ compiler
Missing features can be expected to be implemented
- Very active user/developer base (including Apple and NVIDIA)
- Clean codebase, no legacy issues (as opposed to gcc)
- Modular/library approach leverages synergy effects

Performance already (mostly) on level with gcc generated code, may outperform gcc in the future

Projects like Ocelot, PLANG and VHDL Compiler may help to overcome burdens of increasingly complex systems

Profile guided optimization reduces need for good branch prediction in CPUs -> easier, more energy efficient CPUs