LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation

Chris Lattner and Vikram Adve

Presented by: Nirupama Talele

Agenda

- What is LLVM
- LLVM Code Representation
- LLVM Compiler Architecture
- Framework Analysis

LLVM

- “Compiler framework designed to support transparent, lifelong, program analysis and transformation”
- Provides high level info to compiler transformations
  - Compile-time
  - Link-time
  - Run-time
  - In-idle-time

LLVM

- Program analysis should occur through the lifetime of a program
  - Intra-procedural optimizations (link time)
  - Machine-dependent optimizations (install time)
  - Dynamic optimization (run time)
  - Profile-guided optimizations (idle time)

LLVM Difference with VMs

- No high-level constructs
  - classes, inheritance, etc
- No runtime system or object model
- Does not guarantee safety
  - type and memory

LLVM Analysis

- Aim to make lifelong analysis transparent to programmers
- Achieved through two parts:
  - Code Representation
  - Compiler Architecture

LLVM Code Representation

- Key feature: high and low level
- RISC-like instruction set
  - SSA-based representation
- Low-level, language independent type system
- LLVM is complementary to virtual machines(like JVM,Microsoft CLI), not an alternative

LLVM Code Representation

- How Support Lifelong Analysis?
- 5 capabilities
  - Persistent program information
  - Offline code generation
  - User-based profiling/optimization
  - Transparent runtime model
  - Uniform, whole program compilation
- No previous system provides all 5

Instruction Set

- Avoids machine specific constraints
- Infinite set of typed virtual registers
  - In SSA form
  - Includes support for phi functions
  - This allows flow insensitive algo to gain benefits of flow sensitive without expensive Data Flow analysis
- Avoids same code for multiple instructions (overloaded opcodes)
- Is in load/store form -programs transfer values between registers and memory solely via load and store operations using typed pointers

Type Information

- Makes all address arithmetic explicit, exposing it to all LLVM optimizations.

  Example :- X[i].a = 1; (assuming a is third field)

  ```
  %p = getelementptr %xty* %X, long %i, ubyte 3;
  store int 1, int* %p;
  ```

- All addressable objects (“lvalues”) are explicitly allocated

Exception Handling

- Exceptions mechanism based on two instructions
  - invoke
  - unwind
- Isolate code to throw/recover from exceptions to front-end libraries
- Handling automatic variable destructors:
  - An invoke instruction is used to halt unwinding, the destructor is run, then unwinding is continued with the unwind instruction.

LLVM Compiler Architecture

- Remember: goal to enable transformations at link-time, install-time, run-time, and idle-time
- Must be transparent to application developers and end-users
- Efficient enough for use with real-world applications

LLVM Compiler Architecture

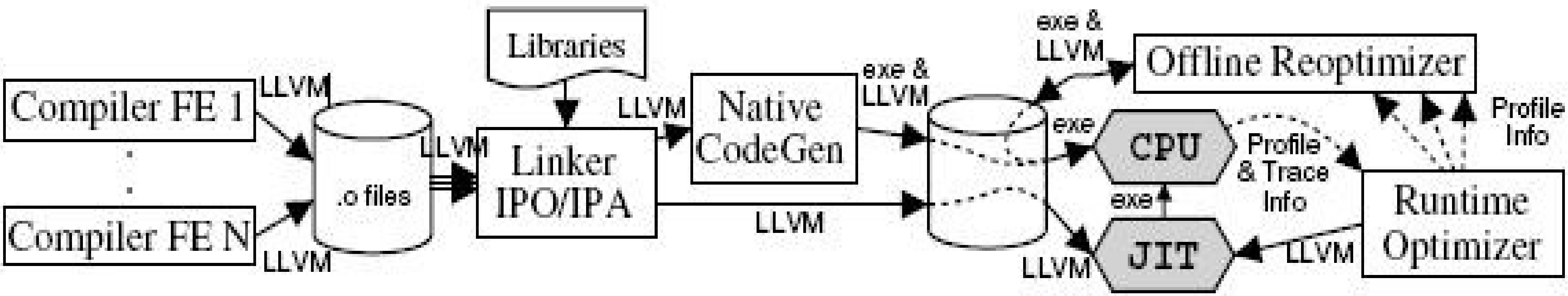

Figure 4: *LLVM system architecture diagram*

- This strategy provides the 5 benefits discussed earlier
- Some limitations
  - Language specific optimizations must be performed on front end
  - Benefit to languages like Java requiring sophisticated runtime systems?

LLVM Compiler Architecture

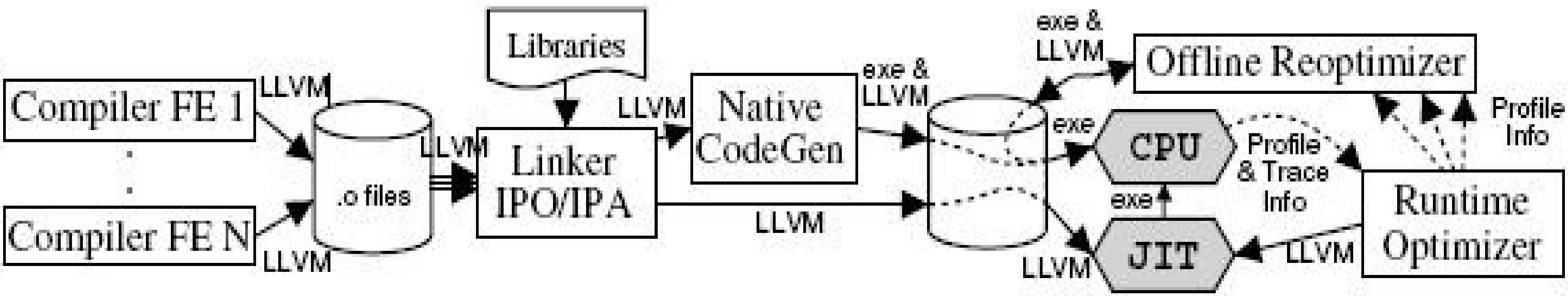

Figure 4: *LLVM system architecture diagram*

- Front-end compiler
  - Translate source code to LLVM representation
  - Perform language specific optimizations
  - Need not perform SSA construction at this time
  - Invoke LLVM passes for global inter procedural optimization at module level

LLVM Compiler Architecture

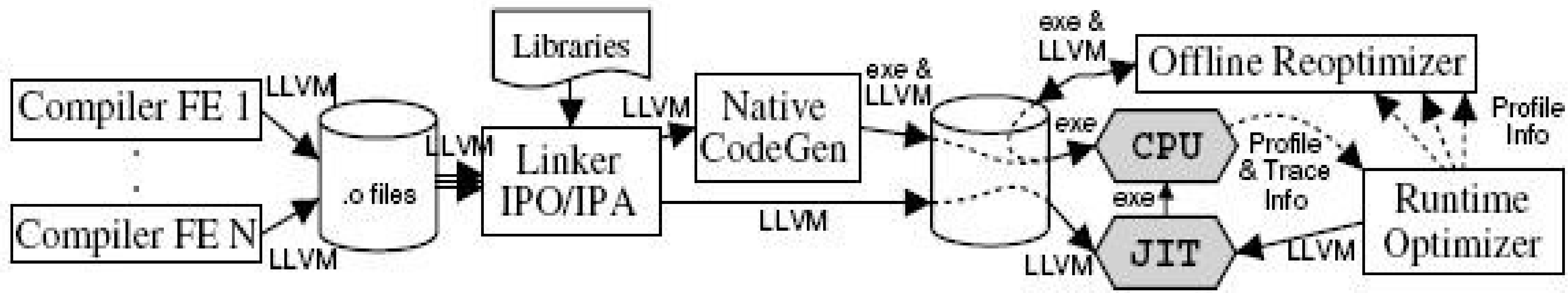

Figure 4: *LLVM system architecture diagram*

- Linker/Interprocedure Optimizer
  - Various analyses occur
    - Points-to analysis
    - Mod/Ref analysis
    - Dead global elimination, dead argument elimination, constant propagation, array bounds check, etc
    - Can be speeded up by adding inter-procedural summaries)

LLVM Compiler Architecture

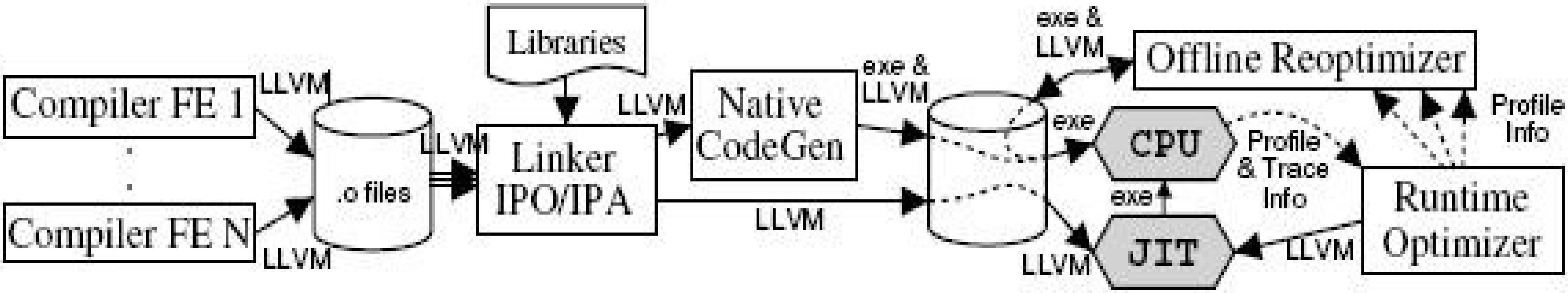

Figure 4: *LLVM system architecture diagram*

- Native Code Generation
  - JIT or Offline
  - Currently supports Sparc V9 and x86 architectures

LLVM Compiler Architecture

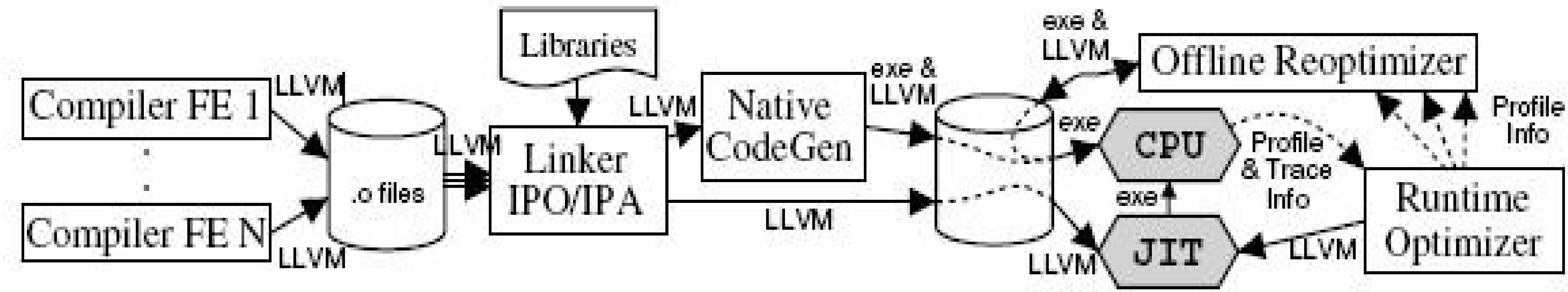

Figure 4: *LLVM system architecture diagram*

- Reoptimizers
  - Identifies frequently run code and ‘hotspots’
  - Performs additional optimizations, thus native code generation can be performed ahead of time
  - Idle-time reoptimizer

LLVM Analysis

- When compiled to LLVM, a program can undergo the following analyses
  - Flow-insensitive, field-sensitive, context-sensitive points-to analysis
  - Uses Data Structure Analysis (DSA)

LLVM Analysis – Code Size

- Relatively compact code size

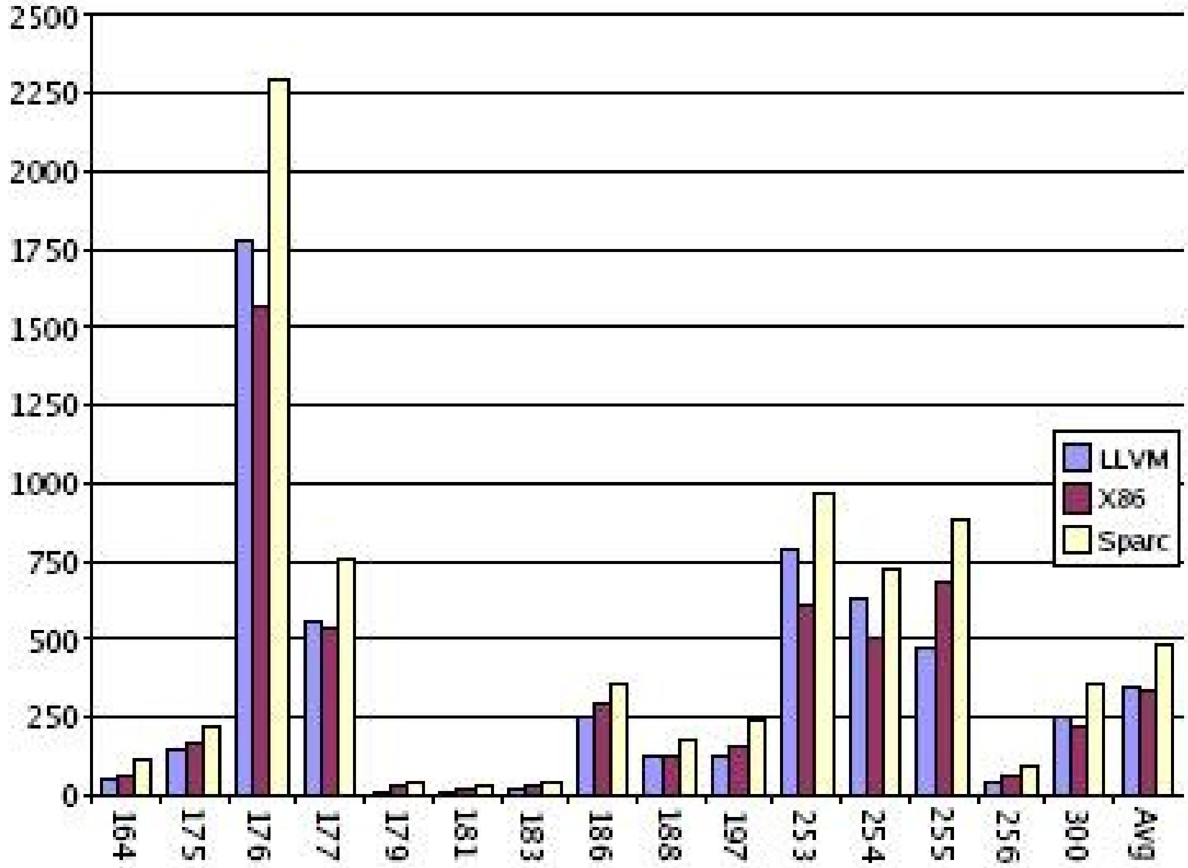

Figure 5: *Executable sizes for LLVM, X86, Sparc (in KB)*

Conclusion

PENNSTATE 1855

- LLVM is language independent
- Optimizations at all stages of software lifetime (compile,link, runtime, etc)
- Compact code size
- Efficient- due to small, uniform instruction set in low level representation
- Future work: can high-level VMs be implemented on top of the LLVM runtime optimization and code generation framework?

Questions?