LLVM, Clang and Embedded Linux Systems

Bruno Cardoso Lopes
University of Campinas
What’s LLVM?
What's LLVM

- Compiler infrastructure
What's LLVM

- Virtual Instruction set (IR)
- SSA
- Bitcode

C source code

```c
int add(int x, int y) {
    return x+y;
}
```

IR

```assembly
define i32 @add(i32 %x, i32 %y) {
  %1 = add i32 %y, %x
  ret i32 %1
}
```
What’s LLVM

• Optimization oriented compiler: compile time, link-time and run-time

• More than 30 analysis passes and 60 transformation passes.
Why LLVM?
Why LLVM?

• Open Source
• Active community
• Easy integration and quick patch review
Why LLVM?

• Code easy to read and understand
• Transformations and optimizations are applied in several parts during compilation
Design
Design

- Written in C++
- Modular and composed of several libraries
- Pass mechanism with pluggable interface for transformations and analysis
- Several tools for each part of compilation
Tools
Tools
Front-end

- Dragonegg
- Gcc 4.5 plugin
- llvm-gcc
- GIMPLE to LLVM IR
Tools
Front-end

- Clang
  - Library approach
  - No cross-compiler generation needed
  - Good diagnostics
  - Static Analyzer
• Optimization are applied to the IR
• *opt* tool

```
$ opt -O3 add.bc -o add2.bc
```

**optimizations**
- Aggressive Dead Code Elimination
- Tail Call Elimination
- Combine Redundant Instructions
- Dead Argument Elimination
- Type-Based Alias Analysis
  ...

```
add.bc  →  add2.bc
```
To o l s
Low Level Compiler

- **llc** tool: invoke the static backends
- Generates assembly or object code

```
$ llc -march=arm add.bc -o add.s
```

```
define i32 @add(i32 %x, i32 %y) {
  %1 = add i32 %y, %x
  ret i32 %1
}
```

```
.globl add
.align 2
add:
  add r0, r1, r0
  bx lr
```
Tools
LLVM Machine Code

- **llvm-mc** tool
- Assembler and Disassembler

```bash
$ llvm-mc -show-encodings -triple armv7-linux add.s
```

```
.globl add
.align 2
add:
    add r0, r1, r0
    bx    lr
```

```
add    r0, r1, r0
    @ encoding: [0x00,0x00,0x81,0xe0]
    bx lr
    @ encoding: [0x1e,0xff,0x2f,0xe1]
```
Tools

- **lli** tool
- Execution Engine library

![Diagram](attachment:image.png)
Tools

libLTO

- **libLTO** library
- Bitcode files treated as native objects
- Enables mixing and matching bitcode w/native objects
Codegen
• LLVM has a target independent code generator.

• Inheritance and overloading are used to specify target specific details.

• TableGen language, created to describe information and generate C++ code.
def ADDPS : PI<0x58,  
    (outs VR256:$dst),  
    (ins VR256:$src1, VR256:$src2),  
    "addps $src2, $src1, $dst",  
    [(set VR256:$dst, (fadd VR256:$src1, VR256:$src2))]>;
def ADDPS : PI<0x58,
  (outs VR256:$dst),
  (ins VR256:$src1, VR256:$src2),
  "addps $src2, $src1, $dst",
  [(set VR256:$dst, (fadd VR256:$src1, VR256:$src2))]>;

Assembly
Codegen

- **TableGen**

def ADDPS : PI<0x58,
       (outs VR256:$dst),
       (ins VR256:$src1, VR256:$src2),
       "addps $src2, $src1, $dst",
       [(set VR256:$dst,  (fadd VR256:$src1, VR256:$src2))]>;
Codegen

• Support several targets

ARM, Alpha, Blackfin, CellSPU, MBlaze, MSP430, Mips, PTX, PowerPC, Sparc, SystemZ, x86, XCore
Codegen

Steps

- Support the target ABI
- Translate IR to real instruction and registers
• Target Calling Convention using TableGen and custom C++ code

• In the front-end

def CC_MipsEABI : CallingConv<[  
    // Promote i8/i16 arguments to i32.
    CClfType<[i8, i16], CCPromoteToType<i32>>>,

    // Integer arguments are passed in integer registers.
    CClfType<[i32], CCAssignToReg<[A0, A1, A2, A3, T0, T1, T2, T3]>>>,
]>>;
Codegen
Translate IR to real instructions

- Back-end
- Legalization phase: nodes could be legal, expanded or customized
- DAGCombine (post and pre legalization)
- Instruction Selection
Codegen
Translate IR to real instructions

- Legalization

<table>
<thead>
<tr>
<th>IR Node</th>
<th>Transformation</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISD::SUB</td>
<td>ISD::SUB</td>
<td>Legal</td>
</tr>
<tr>
<td>ISD::SINT_TO_FP</td>
<td>Several other nodes</td>
<td>Expand</td>
</tr>
<tr>
<td>ISD::CTTZ</td>
<td>ARMISD::RBIT + ISD::CTLZ</td>
<td>Custom</td>
</tr>
</tbody>
</table>
Codegen
Translate IR to real instructions

- **DAGCombine**

  - if $A$ is constant
    \[(\text{mul } x, 2^N + 1) \rightarrow (\text{add } (\text{shl } x, N), x)\]

  - if $C1 \& C2 == C1$
    \[(\text{bfi } A, (\text{and } B, C1), C2) \rightarrow (\text{bfi } A, B, C2)\]
Codegen

Translate IR to real instructions

- Instruction selection
- Tablegen pattern matching
- Custom C++ handling

```cpp
def ADDiu {
    list<dag> Pattern = [(set CPURegs:$dst, 
         (add CPURegs:$src, imm:$c))];
}
```
Codegen
Target Specific Optimizations

- Registered as passes

```c++
bool SparcTargetMachine::addPreEmitPass(PassManagerBase &PM, bool Fast) {
  PM.add(createSparcFPMoverPass(*this));
  PM.add(createSparcDelaySlotFillerPass(*this));
  return true;
}
```
Codegen
ARM

**Archs**
- V4T
- V5TE
- V6
- V6M
- V6T2
- V7A
- V7M

**Processors**
- Cortex M0, A8, A9, M3, M4

**Features**
- Neon, VFP2, VFP3, Thumb2
• Target specific optimizations

PM.add(createARMLoadStoreOptimizationPass());
PM.add(createThumb2SizeReductionPass());
PM.add(createARMConstantIslandPass());
PM.add(createThumb2ITBlockPass());
• **Constant Islands pass:**
  • Limited PC-relative displacements
  • Constants are scattered among instructions in a function
Codegen

ARM

- Load Store optimizer
- Create load/store multiple instructions
- Recognize LDRD and STRD
Codegen

ARM

- Code size reduction
- 32bit instructions to 16bit ones
• IT Block pass
• Recognize instruction suitable to become part of an IT block.
• Supports O32 and EABI.
• Mips I, 4ke and allegrex core (PSP)
• No target specific optimizations.
Questions?