Understanding and writing an LLVM compiler back-end

Embedded Linux Conference 2009
San Francisco, CA

Bruno Cardoso Lopes
bruno.cardoso@gmail.com
Agenda

- What’s LLVM?
- LLVM design
- The back-end
- Why LLVM?
- Who’s using
What's LLVM?

Basics

- Low Level Virtual Machine
- A virtual instruction set
- A compiler infrastructure suite with aggressive optimizations.
A virtual instruction set

- Low-level representation, but with high-level type information
- 3-address-code-like representation
- RISC based, language independent with SSA information. The on-disk representation is called **bitcode**.
A virtual instruction set

int dummy(int a) {
    return a+3;
}

define i32 @dummy(i32 %a) nounwind readnone {
    entry:
        %0 = add i32 %a, 3
        ret i32 %0
}
A compiler infrastructure suite

- Compiler front-end
  - llvm-gcc
  - clang
- IR and tools to handle it
- JIT and static back-ends.
Front-end: llvm-gcc

- GCC based front-end: llvm-gcc
- GENERIC to LLVM instead of GIMPLE
- GIMPLE is only an in-memory IR
- GCC is not modular (intentionally)
Front-end: llvm-gcc

- Very mature and supports Java, Ada, FORTRAN, C, C++ and ObjC.
- Cross-compiler needed for a not native target.

```bash
$ llvm-gcc -O2 -c clown.c -emit-llvm -o clown.bc
$ llvm-extract -func=bozo < clown.bc | llvm-dis

define float @bozo(i32 %lhs, i32 %rhs, float %w) nounwind {
  entry:
    %0 = sdiv i32 %lhs, %rhs
    %1 = sitofp i32 %0 to float
    %2 = mul float %1, %w
    ret float %2
}
```
Front-end: clang

- Clang: C front-end.
- Still under heavy development process
- Better diagnostics
- Integration with IDEs
- No need to generate a cross-compiler
- Static Analyzer
Front-end: clang

$ clang -fsyntax-only ~/bozo.c -pedantic
/tmp/bozo.c:2:17: warning: extension used
typedef float V __attribute__((vector_size(16)));
1 diagnostic generated.

$ clang vect.c -emit-llvm | opt -std-compile-opts | llvm-dis

define <4 x float> @foo(<4 x float> %a, <4 x float> %b) {
  entry:
    %0 = mul <4 x float> %b, %a
    %1 = add <4 x float> %0, %a
    ret <4 x float> %1
}
Optimization oriented compiler

- Provides **compile time, link-time** and **run-time** optimizations (profile-guided transformations collected by a dynamic profiler).
Optimizations

- Compile-time optimizations
- Driven with -O{1,2,3,s} in llvm-gcc
- Link-time (cross-file, interprocedural) optimizations
- 32 analysis passes and 63 transform passes
## Optimizations

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-adce</td>
<td>Aggressive Dead Code Elimination</td>
</tr>
<tr>
<td>-tailcallelim</td>
<td>Tail Call Elimination</td>
</tr>
<tr>
<td>-instcombine</td>
<td>Combine redundant instructions</td>
</tr>
<tr>
<td>-deadargelim</td>
<td>Dead Argument Elimination</td>
</tr>
<tr>
<td>-aa-eval</td>
<td>Exhaustive Alias Analysis Precision Evaluator</td>
</tr>
<tr>
<td>-anders-aa</td>
<td>Andersen's Interprocedural Alias Analysis</td>
</tr>
<tr>
<td>-basicaa</td>
<td>Basic Alias Analysis (default AA impl)</td>
</tr>
</tbody>
</table>
Set of tools

- **llc** - invoke static back-ends.
- **lli** - bitcode interpreter, use JIT
- **bugpoint** - reduce code from crashes
- **opt** - run optimizations on bitcodes
- **llvm-extract** - extract/delete functions and data
- **llvm-dis, llvm-as, llvm-ld, ...**
$ clang vect.c -emit-llvm | opt -std-compile-opts | llvm-dis

define <4 x float> @foo(<4 x float> %a, <4 x float> %b) {
  entry:
  %0 = mul <4 x float> %b, %a
  %1 = add <4 x float> %0, %a
  ret <4 x float> %1
}

$ clang vect.c -emit-llvm | opt -std-compile-opts | llc
  -march=x86 -mcpu=yonah

  _foo:
  mulps %xmm0, %xmm1
  addps %xmm0, %xmm1
  movaps %xmm1, %xmm0
  ret
• **llvmc**, a llvm based compiler driver, like gcc

```
llvmc -O2 x.c y.c z.c -o xyz
llvmc -O2 x.c -o x.o
llvmc -O2 y.c -o y.o
llvmc -O2 z.c -o z.o
llvmc -O2 x.o y.o z.o -o xyz
```
Putting it all together

```c
float bozo(int lhs, int rhs, float w)
{
    float c = (lhs/rhs)*w;
    return c;
}
```

```
llvm-gcc --emit-llvm -c bozo.c -o bozo.bc
```

```
llvm-extract -func=bozo bozo.bc | opt -std-compile-opts
```
Putting it all together

```c
define float @bozo(i32 %lhs, i32 %rhs, float %w)
{
  entry:
  %0 = sdiv i32 %lhs, %rhs
  %1 = sitofp i32 %0 to float
  %2 = mul float %1, %w
  ret float %3
}
```

```
bozo:
stmfd sp!, {r4, lr}
sub sp, sp, #8
mov r4, r2
bl __divsi3
bl __floatsisf
```

```
... | llc -march=arm
```

```
size bozo, ..bozo
```

Optimizations

Back-end
Design

- Well written C++
- Everything implemented as passes
- Easy to plug/unplug transformations and analysis
- Pluggable register allocators
- Modular and Pluggable optimization framework
- Library approach (Runtime, Target, Code Generation, Transformation, Core and Analysis libraries)
The back-end

- **Targets:**
  
  Alpha, ARM, C, CellSPU, IA64, Mips, MSIL, PowerPC, Sparc, X86, X86_64, XCore, PIC-16

- **JIT for X86, X86-64, PowerPC 32/64, ARM**

$ lli hello.bc
The back-end

- Stable back-ends: X86/X86_64, PowerPC 32/64 and ARM
- Each back-end is a standalone library.
Back-end tasks

- Support the target ABI
- Translate the IR to real instructions and registers.
  - Instruction selection
  - Scheduling
- Target specific optimizations
How's that done?

- 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.
TableGen

- TableGen can describe the architecture Calling Convention, Instruction, Registers, ...

- High and low level representations at the same time, e.g. bit fields and DAGs could be represented

def RET {
    // Instruction MipsInst FR
    field bits<32> Inst = {..., rd{4}, rd{3}, rd{2}, rd{1}, ...};
    dag OutOperandList = (outs);
    dag InOperandList = (ins CPURegs:$target);
    string AsmString = "jr $target";
    list<dag> Pattern = [(MipsRet CPURegs:$target)];
    ...
    bits<6> funct = { 0, 0, 0, 0, 1, 0 };  
}
Registers

```python
def ZERO : MipsGPRReg< 0, "ZERO">, DwarfRegNum<[0]>;
def AT : MipsGPRReg< 1, "AT">, DwarfRegNum<[1]>;
def V0 : MipsGPRReg< 2, "2">, DwarfRegNum<[2]>;
```

Subtargets

```python
ARM
    def : Proc<"arm1176jzf-s", [ArchV6, FeatureVFP2]>;

PPC
```
Target specific Nodes

def MipsJmpLink : SDNode<"MipsISD::JmpLink",
SDT_MipsJmpLink,
[SDNPHasChain,
SDNPOutFlag]>;

Instructions

let isReturn=1, hasDelaySlot=1, isBarrier=1, hasCtrlDep=1, ... in
def RET : FR <0x00, 0x02, (outs), (ins CPURegs:$target),
"jr\t$target", [(MipsRet CPURegs:$target)], IIBranch>;
Calling Conventions

Describe target specific ABI information

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

   // Integer arguments are passed in integer registers.
   CCIfType<[i32], CCAssignToReg<[A0, A1, A2, A3, T0, T1, T2, T3]>>,
   ...
   CCIfType<[f32], CCIfSubtarget:"isNotSingleFloat()",
             CCAssignToReg<[F12, F14, F16, F18]>>,

   // Integer values get stored in stack slots that are 4 bytes in
   // size and 4-byte aligned.
   CCIfType<[i32, f32], CCAssignToStack<4, 4>>,
   ...
]>
```
Legalization and Lowering

- Specify which nodes are legal on the target.
- Not legal ones can be expanded or customized (lowered) to target specific nodes.
- Some nodes must always be customized: e.g. CALL, FORMAL_ARGUMENTS, RET

```c
setOperationAction(ISD::JumpTable, MVT::i32, Custom);
setOperationAction(ISD::ConstantPool, MVT::i32, Custom);
setOperationAction(ISD::DYNAMIC_STACKALLOC, MVT::i32, Custom);

if (!Subtarget->hasSEInReg()) {
  setOperationAction(ISD::SIGN_EXTEND_INREG, MVT::i8, Expand);
  setOperationAction(ISD::SIGN_EXTEND_INREG, MVT::i16, Expand);
}
```
int dummy(int a) {
    return a+3;
}

define i32 @dummy(i32 %a) nounwind readnone {
    entry:
    %0 = add i32 %a, 3
    ret i32 %0
}
Legalization and Lowering

LowerOperation(SDOperand Op, SelectionDAG &DAG)
{
    switch (Op.getOpcode())
    {
    case ISD::CALL:               return LowerCALL(Op, DAG);
    case ISD::JumpTable:          return LowerJumpTable(Op, DAG);
    case ISD::GlobalAddress:      return LowerGlobalAddress(Op, DAG);
    case ISD::RET:                return LowerRET(Op, DAG);
    case ISD::DYNAMIC_STACKALLOC: return LowerDYNAMIC_STACKALLOC(Op, DAG);
    ...
isel input for dummy:entry
Instruction Selection

- After legalization and lowering
- Nodes are matched with target instructions defined by TableGen or handled by special C++ code

```cpp
def ADDiu {
    list<dag> Pattern = [(set CPURegs:$dst, 
                         (add CPURegs:$b, immSExt16:$c))];
}
```

```cpp
case MipsISD::JmpLink: {
    if (TM.getRelocationModel() == Reloc::PIC_) {
        ....
    }
```
Patterns

- Patterns used to help instruction selection

```c
// Small immediates
def : Pat<(i32 immSExt16:$in),
   (ADDiu ZERO, imm:$in)>;

// Arbitrary immediates
def : Pat<(i32 imm:$imm), (ORi (LUi (HI16 imm:$imm)), (LO16 imm:$imm))>;

def : Pat<(not CPURegs:$in),
   (NOR CPURegs:$in, ZERO)>;
```
Target optimizations

- Registered as passes

```c++
bool ARMTargetMachine::addPreEmitPass(PassManagerBase &PM, bool Fast) {
    if (!Fast && !DisableLdStOpti && !Subtarget.isThumb())
        PM.add(createARMLoadStoreOptimizationPass());

    if (!Fast && !DisableIfConversion && !Subtarget.isThumb())
        PM.add(createIfConverterPass());

    PM.add(createARMConstantIslandPass());
    return true;
}

bool SparcTargetMachine::addPreEmitPass(PassManagerBase &PM, bool Fast) {
    PM.add(createSparcFPMoverPass(*this));
    PM.add(createSparcDelaySlotFillerPass(*this));
    return true;
}
```
Quick roadmap to a new back-end

- Start with Mips and Sparc backends as references.
- Describe target specific information with TableGen.
- Implement ABI and Lowering.
- Peepholes and target specific optimizations.
- Compile the llvm test suite.
Why LLVM?

- Doesn't have a very steep learning curve compared to other known compilers
- Easy to apply custom transformations and optimizations in anytime of compilation
- Open source - BSD based license
- Very active community
- Patches get reviewed and integrated to mainline quickly
<table>
<thead>
<tr>
<th>Who’s using?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPython</strong></td>
</tr>
<tr>
<td><strong>QEmu</strong></td>
</tr>
<tr>
<td><strong>PyPy</strong></td>
</tr>
<tr>
<td><strong>Adobe</strong></td>
</tr>
<tr>
<td><strong>Apple</strong></td>
</tr>
<tr>
<td><strong>Microchip</strong></td>
</tr>
<tr>
<td><strong>RapidMind</strong></td>
</tr>
</tbody>
</table>
Contact

- llvmdev mailing list
- #llvm @ irc.oftc.net
Understanding and writing an LLVM compiler back-end

- Thanks !!!
- Questions !?

Bruno Cardoso Lopes
bruno.cardoso@gmail.com