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
 - Ilvm-gcc
 - clang
- IR and tools to handle it
- JIT and static back-ends.

Front-end
$$\longrightarrow$$
 IR \longrightarrow Back-end

Front-end: Ilvm-gcc

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

Front-end: Ilvm-gcc

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

```
$ llvm-gcc -02 -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
}</pre>
LLVM assembly
```

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

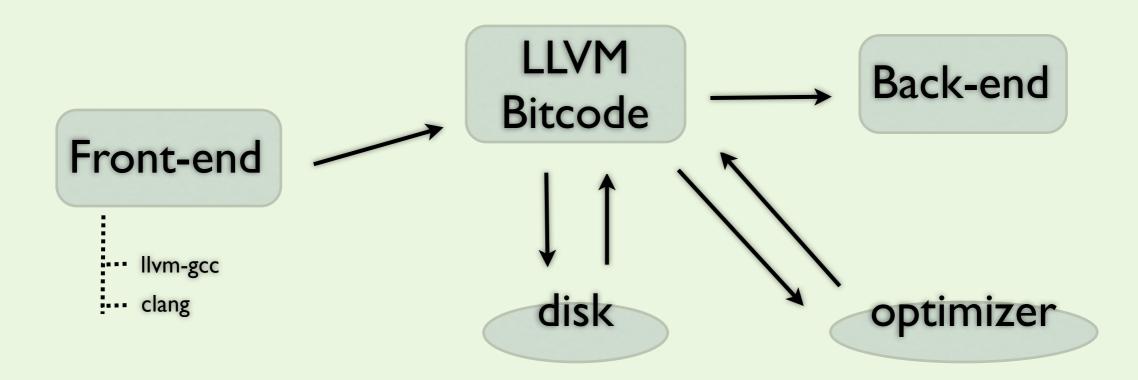
```
$ clang -fsyntax-only ~/bozo.c -pedantic
/tmp/bozo.c:2:17: warning: extension used
typedef float V __attribute__((vector_size(16)));
1 diagnostic generated.
```

```
$ 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 Ilvm-gcc
- Link-time (cross-file, interprocedural) optimizations
- 32 analysis passes and 63 transform passes

Optimizations

-adce	Aggressive Dead Code Elimination
-tailcallelim	Tail Call Elimination
-instcombine	Combine redundant instructions
-deadargelim	Dead Argument Elimination
-aa-eval	Exhaustive Alias Analysis Precision Evaluator
-anders-aa	Andersen's Interprocedural Alias Analysis
-basicaa	Basic Alias Analysis (default AA impl)

Set of tools

- Ilc invoke static back-ends.
- Ili bitcode interpreter, use JIT
- bugpoint reduce code from crashes
- opt run optimizations on bitcodes
- Ilvm-extract extract/delete functions and data
- Ilvm-dis, Ilvm-as, Ilvm-ld, ...

LLVM Usage Examples

```
$ 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
```

LLVM Driver

• Ilvmc, a llvm based compiler driver, like gcc

```
llvmc -02 x.c y.c z.c -0 xyz

llvmc -02 x.c -0 x.o
llvmc -02 y.c -0 y.o
llvmc -02 z.c -0 z.o
llvmc -02 x.o y.o z.o -0 xyz
```

Putting it all together

float bozo(int lhs, int rhs, float w) float c = (lhs/rhs)*w; return c; Front-end llvm-gcc --emit-llvm -c bozo.c -o bozo.bc **Optimizations** llvm-extract -func=bozo bozo.bc | opt -std-compile-opts

Putting it all together

Optimizations

Back-end

```
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} mov r1, r4
sub sp, sp, #8
bl __mulsf3
mov r4, r2 add sp, sp, #8
bl __divsi3 ldmfd sp!, {r4, pc}
bl __floatsisf .size bozo, .-bozo
```

Design

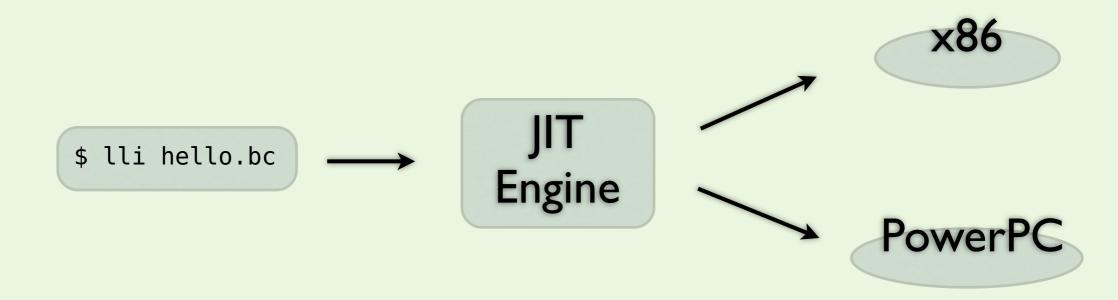
- Well written C++
- Everything implemented as passes
- Easy to plug/unplug transformations and analysis
- Pluggable register allocators
- Modular and Pluggable optimization framework
- Library approach (Runtime, Target, Code Generation, Transformation, Core and Analysis libraries)

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



The back-end

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

ARMISelDAGToDAG.cpp ARMInstrVFP.td ARMJITInfo.cpp ARMCodeEmitter.cpp ARMLoadStoreOptimizer.cpp ARMConstantIslandPass.cpp ARMInstrInfo.td ARMInstrThumb.td

ARMSubtarget.cpp ARMInstrFormats.td ARMRegisterInfo.h ARMRegisterInfo.td

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

```
def ZER0 : MipsGPRReg< 0, "ZER0">, DwarfRegNum<[0]>;
def AT : MipsGPRReg< 1, "AT">, DwarfRegNum<[1]>;
def V0 : MipsGPRReg< 2, "2">, DwarfRegNum<[2]>;
```

Subtargets

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

PPC
  def : Processor<"g4+", G4PlusItineraries, [Directive750, FeatureAltivec]>;
```

Target specific Nodes

Instructions

Calling Conventions

Describe target specific ABI information

Legalization and Lowering

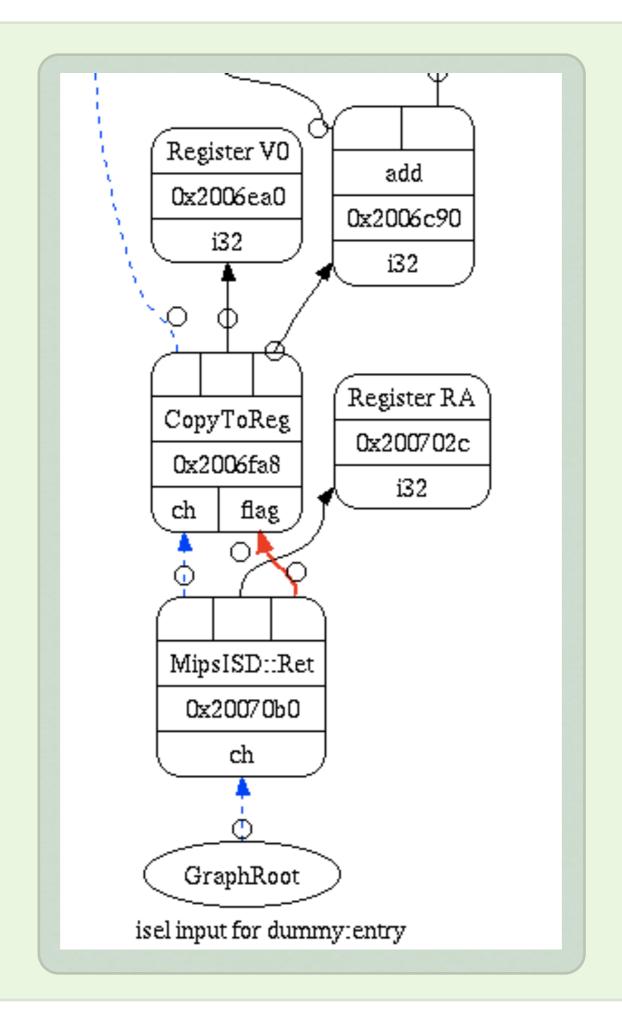
- Specify which nodes are legal on the target.
- Not legal ones can be expanded or customized (lowered) to target specific nodes.
- Some nodes must always be customized : e.g.
 CALL, FORMAL_ARGUMENTS, RET

```
Register #1025
                                                                    EntryToken
                                                                    0x19044a4
                                                                                  0x2006d98
                                                                                     i32
                                                                       ch
                                                                                            Constant: 3
                                                                             CopyFromReg
                                                                                            0x2006c0c
                   int dummy(int a) {
                                                                              0x2006e1c
                      return a+3;
                                                                                               i32
                                                                              i32
                                                                                     ch
                                                                                          ArgFlags AF=<>
                                                                                 add
                                                                                             0x2006d14
                                                                              0x2006c90
                                                                                                ch
                                                                                 i32
define i32 @dummy(i32 %a) nounwind readnone {
entry:
     %0 = add i32 %a, 3
                                                                                 ret
     ret i32 %0
                                                       Before lowering
                                                                              0x2006f24
                                                                                 ch
                                                                              GraphRoot
                                                                          legalize input for dummy:entry
```

Legalization and Lowering

```
Node Lowering

CALL, FORMAL_ARGUMENTS, Ret,
GlobalAdress, JumpTable
```



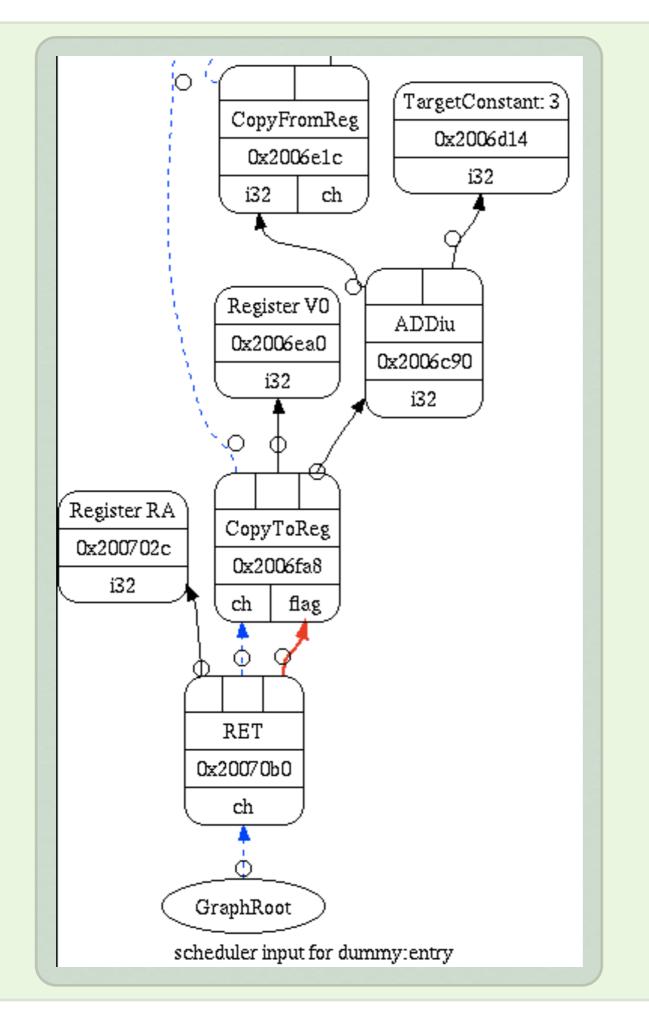
Instruction Selection

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

Direct instruction matching

```
C++ code handling
```

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



Patterns

Patterns used to help instruction selection

Target optimizations

Registered as passes

```
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 IIvm 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

Who's using?

CPython	LLVM JIT
QEmu	dynamic binary translator converted to LLVM JIT
РуРу	JIT
Adobe	Hydra language CPU JIT
Apple	OpenGL JIT Compiler, IPhone
Microchip	Static Back-end
RapidMind	Static Back-end

Contact

- Ilvmdev mailing list
- #Ilvm @ irc.oftc.net

Understanding and writing an LLVM compiler back-end

- Thanks !!!
- Questions !?

Bruno Cardoso Lopes bruno.cardoso@gmail.com