The LLVM MIPS and ARM back-end

Embedded Linux Conference 2009
Grenoble, France

Bruno Cardoso Lopes
bruno.cardoso@gmail.com
Agenda

- What’s LLVM?
- Why?
- LLVM design
-CodeGen
- The MIPS and ARM back-end
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**.
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.
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, ...
GCC based front-end: llvm-gcc

GIMPLE to LLVM IR

GCC is not modular (intentionally)
• 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
```

```llvm
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
}
```
Clang:  C, ObjC front-end.

C++ in progress

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><code>-adce</code></td>
<td>Aggressive Dead Code Elimination</td>
</tr>
<tr>
<td><code>-tailcallelim</code></td>
<td>Tail Call Elimination</td>
</tr>
<tr>
<td><code>-instcombine</code></td>
<td>Combine redundant instructions</td>
</tr>
<tr>
<td><code>-deadargelim</code></td>
<td>Dead Argument Elimination</td>
</tr>
<tr>
<td><code>-aa-eval</code></td>
<td>Exhaustive Alias Analysis Precision Evaluator</td>
</tr>
<tr>
<td><code>-anders-aa</code></td>
<td>Andersen's Interprocedural Alias Analysis</td>
</tr>
<tr>
<td><code>-basicaa</code></td>
<td>Basic Alias Analysis (default AA impl)</td>
</tr>
</tbody>
</table>
$ 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
Putting it all together

Front-end

Optimizations

```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
```
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
}

... | llc -march=arm

bozo:
  stmfd sp!, {r4, lr}
  mov r1, r4
  sub sp, sp, #8
  mov r4, r2
  bl __divsi3
  bl __floatsisf
  ldmfd sp!, {r4, pc}
  .size bozo, .-bozo
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
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, MSP430, SystemZ, Blackfin

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

```bash
$ lli hello.bc
```
The back-end

- Each back-end is a standalone library.

- ARMISelDAGToDAG.cpp
- ARMJITInfo.cpp
- ARMCCodeEmitter.cpp
- ARMLoadStoreOptimizer.cpp
- ARMInstrInfo.td
- ARMInstrThumb.td
- ARMInstrVFP.td
- ARMSubtarget.cpp
- ARMInstrFormats.td
- ARMConstantIslandPass.cpp
- ARMRegisterInfo.h
- ARMRegisterInfo.td
Back-end tasks

• Codegen
• 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** 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

```cpp
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 ZERO : MipsGPRReg< 0, "ZERO">, 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 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.
  - Legal nodes can be directly mapped to a target machine instruction.
  - Expansion replaces the node with distinct nodes to represent the same operation.
  - Customization transforms the node into target specific nodes.
setOperationAction(ISD::JumpTable, MVT::i32, Custom);
setOperationAction(ISD::DYNAMIC_STACKALLOC, MVT::i32, Custom);
setOperationAction(ISD::SELECT, 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
}
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);
    ...
  }
}
Some nodes must always be transformed into target specific ones (customized): CALL, FORMAL_ARGUMENTS, RET
Instruction Selection

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

Direct instruction matching

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

C++ code handling

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

- Patterns used to help instruction selection
- Perform peephole optimizations.

```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))>;

// Instructions for CPU
def : Pat<(not CPURegs:$in),
    (NOR CPURegs:$in, ZERO)>;
```
DAG after instruction selection

Matched from target specific node

Matched from legal node
Target optimizations

- Registered as passes

```cpp
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;
}
```

```cpp
bool SparcTargetMachine::addPreEmitPass(PassManagerBase &PM, bool Fast) {
    PM.add(createSparcFPMoverPass(*this));
    PM.add(createSparcDelaySlotFillerPass(*this));
    return true;
}
```
Mips

• Supports O32 and EABI.
• Mips I and allegrex core (PSP)
• No target specific optimizations.
ARM

- Support several flavors:
  - v4t, v5t, v5te, v6, v6t2, v7a,
- Features:
  - vfp2, vfp3, neon, thumb2
• Constant Islands pass:
  • ARM specific optimization.
  • Limited PC-relative displacements.
  • Constants are scattered among instructions in a function.
• Load/store optimizer
  • Performs load/store related peephole optimizations
  • Merge several load/store instructions into one or more load/store multiple instructions
ARM

- Thumb2
    - Code size reduction pass
    - 32-bit insns to 16-bit ones
    - 32-bit insns to 2addr 16-bit ones
    - 32-bit load/store to 6-bit ones
    - IT Block pass.
The LLVM MIPS and ARM back-end

Embedded Linux Conference 2009
Grenoble, France

Bruno Cardoso Lopes
bruno.cardoso@gmail.com