The LLVM Compiler
Framework and Infrastructure

Vikram Adve
vadve@cs.uiuc.edu

Chris Lattner
lattner@cs.uiuc.edu

http://llvm.cs.uiuc.edu/

LCPC Tutorial: September 22, 2004
Acknowledgements

UIUC Contributors:
- Tanya Brethour
- Misha Brukman
- Cameron Buschardt
- John Criswell
- Alkis Evlogimenos
- Brian Gaeke
- Ruchira Sasanka
- Anand Shukla
- Bill Wendling

External Contributors:
- Henrik Bach
- Nate Begeman
- Jeff Cohen
- Paolo Invernizzi
- Brad Jones
- Vladimir Merzliakov
- Vladimir Prus
- Reid Spencer

Funding:
This work is sponsored by the NSF Next Generation Software program through grants EIA-0093426 (an NSF CAREER award) and EIA-0103756. It is also supported in part by the NSF Operating Systems and Compilers program (grant #CCR-9988482), the NSF Embedded Systems program (grant #CCR-0209202), the MARCO/DARPA Gigascale Systems Research Center (GSRC), IBM through the DARPA-funded PERCS project, and the Motorola University Partnerships in Research program.
The LLVM Compiler System

- The LLVM Compiler Infrastructure
  - Provides reusable components for building compilers
  - Reduce the time/cost to build a new compiler
  - Build static compilers, JITs, trace-based optimizers, ...

- The LLVM Compiler Framework
  - End-to-end compilers using the LLVM infrastructure
  - C and C++ are robust and aggressive:
    - Java, Scheme and others are in development
  - Emit C code or native code for X86, Sparc, PowerPC
Three primary LLVM components

- The LLVM *Virtual Instruction Set*
  - The common language- and target-independent IR
  - Internal (IR) and external (persistent) representation

- A collection of well-integrated libraries
  - Analyses, optimizations, code generators, JIT compiler, garbage collection support, profiling, …

- A collection of tools built from the libraries
  - Assemblers, automatic debugger, linker, code generator, compiler driver, modular optimizer, …
Tutorial Overview

- Introduction to the running example
- LLVM C/C++ Compiler Overview
  - High-level view of an example LLVM compiler
- The LLVM Virtual Instruction Set
  - IR overview and type-system
- LLVM C++ IR and important API’s
  - Basics, PassManager, dataflow, ArgPromotion
- Important LLVM Tools
  - opt, code generator, JIT, test suite, bugpoint
- Example applications of LLVM
Running example: arg promotion

Consider use of by-reference parameters:

```c
int callee(const int &X) {
    return X+1;
}
int caller() {
    return callee(4);
}
```

compiles to

```c
int callee(const int *X) {
    return *X+1;  // memory load
}
int caller() {
    int tmp;      // stack object
    tmp = 4;      // memory store
    return callee(&tmp);
}
```

We want:

- Eliminated load in callee
- Eliminated store in caller
- Eliminated stack slot for ‘tmp’
Why is this hard?

- **Requires interprocedural analysis:**
  - Must change the prototype of the callee
  - Must update all call sites → we must **know** all callers
  - What about callers outside the translation unit?

- **Requires alias analysis:**
  - Reference could alias other pointers in callee
  - Must know that loaded value doesn’t change from function entry to the load
  - Must know the pointer is not being stored through

- **Reference might not be to a stack object!**
Tutorial Overview

- Introduction to the running example
- LLVM C/C++ Compiler Overview
  - High-level view of an example LLVM compiler
- The LLVM Virtual Instruction Set
  - IR overview and type-system
- LLVM C++ IR and important API’s
  - Basics, PassManager, dataflow, ArgPromotion
- Important LLVM Tools
  - opt, code generator, JIT, test suite, bugpoint
- Example applications of LLVM
The LLVM C/C++ Compiler

- From the high level, it is a standard compiler:
  - Compatible with standard makefiles
  - Uses GCC 3.4 C and C++ parser

- Distinguishing features:
  - Uses LLVM optimizers, not GCC optimizers
  - .o files contain LLVM IR/bytecode, not machine code
  - Executable can be bytecode (JIT’ed) or machine code
Looking into events at compile-time

C file $\rightarrow$ llvmgcc $\rightarrow$ .o file

C to LLVM Frontend $\rightarrow$ Compile-time Optimizer

“cc1” $\rightarrow$ LLVM IR Parser $\rightarrow$ LLVM Verifier

LLVM IR File Writer

Dead Global Elimination, IP Constant Propagation, Dead Argument Elimination, Inlining, Reassociation, LICM, Loop Opts, Memory Promotion, Dead Store Elimination, ADCE, …

C++ file $\rightarrow$ llvmg++ $\rightarrow$ .o file

C++ to LLVM Frontend $\rightarrow$ Compile-time Optimizer

“cc1plus” $\rightarrow$ “gccas”

Modified version of GCC
Emits LLVM IR as text file
Lowers C++ AST to LLVM

“gccas” $\rightarrow$ “gccas”

Modified version of G++
Emits LLVM IR as text file
Lowers C++ AST to LLVM

40 LLVM Analysis & Optimization Passes

LLVM .bc File Writer

http://llvm.cs.uiuc.edu/

Chris Lattner – lattner@cs.uiuc.edu
Looking into events at link-time

LLVM Linker → LLVM Linker

Link-time Optimizer → .o file

.llc file for LLVM JIT

20 LLVM Analysis & Optimization Passes

Optionally “internalizes”: marks most functions as internal, to improve IPO

Perfect place for argument promotion optimization!

Native Code Backend

"llc" → Native executable

"llc –march=c" → Native executable

C Code Backend

C Compiler

"gcc" → Native executable

C Compiler

Native executable

Link in native .o files and libraries here

Chris Lattner – lattner@cs.uiuc.edu

http://llvm.cs.uiuc.edu/
Goals of the compiler design

- **Analyze and optimize as early as possible:**
  - Compile-time opts reduce modify-rebuild-execute cycle
  - Compile-time optimizations reduce work at link-time (by shrinking the program)

- **All IPA/IPO make an open-world assumption**
  - Thus, they all work on libraries and at compile-time
  - “Internalize” pass enables “whole program” optzn

- **One IR (without lowering) for analysis & optzn**
  - Compile-time optzns can be run at link-time too!
  - The same IR is used as input to the JIT

*IR design is the key to these goals!*
Tutorial Overview

- Introduction to the running example
- LLVM C/C++ Compiler Overview
  - High-level view of an example LLVM compiler
- The LLVM Virtual Instruction Set
  - IR overview and type-system
- LLVM C++ IR and important API’s
  - Basics, PassManager, dataflow, ArgPromotion
- Important LLVM Tools
  - opt, code generator, JIT, test suite, bugpoint
- Example applications of LLVM

http://llvm.cs.uiuc.edu/
Goals of LLVM IR

- Easy to produce, understand, and define!
- Language- and Target-Independent
  - AST-level IR (e.g. ANDF, UNCOL) is not very feasible
    - Every analysis/xform must know about ‘all’ languages
- One IR for analysis and optimization
  - IR must be able to support aggressive IPO, loop opts, scalar opts, … high- and low-level optimization!
- Optimize as much as early as possible
  - Can’t postpone everything until link or runtime
  - No lowering in the IR!
LLVM Instruction Set Overview #1

- **Low-level and target-independent semantics**
  - RISC-like three address code
  - Infinite virtual register set in SSA form
  - Simple, low-level control flow constructs
  - Load/store instructions with typed-pointers

- **IR has text, binary, and in-memory forms**

```assembly
loop:
  %i.1 = phi int [ 0, %bb0 ], [ %i.2, %loop ]
  %AiAddr = getelementptr float* %A, int %i.1
  call void %Sum(float %AiAddr, %pair* %P)
  %i.2 = add int %i.1, 1
  %tmp.4 = settlt int %i.1, %N
  br bool %tmp.4, label %loop, label %outloop

For (i = 0; i < N; ++i)
  Sum(&A[i], &P);
```

http://llvm.cs.uiuc.edu/
High-level information exposed in the code

- Explicit dataflow through SSA form
- Explicit control-flow graph (even for exceptions)
- Explicit language-independent type-information
- Explicit typed pointer arithmetic
- Preserve array subscript and structure indexing

```llvm
loop:

  %i.1 = phi int [ 0, %bb0 ], [ %i.2, %loop ]
  %AiAddr = getelementptr float* %A, int %i.1
  call void %Sum(float %AiAddr, %pair* %P)
  %i.2 = add int %i.1, 1
  %tmp.4 = setlt int %i.1, %N
  br bool %tmp.4, label %loop, label %outloop
```

For (i = 0; i < N; ++i)

Sum(&A[i], &P);
The entire type system consists of:
- Primitives: void, bool, float, ushort, opaque, ...
- Derived: pointer, array, structure, function
- No high-level types: type-system is language neutral!

Type system allows arbitrary casts:
- Allows expressing weakly-typed languages, like C
- Front-ends can implement safe languages
- Also easy to define a type-safe subset of LLVM

See also: docs/LangRef.html
Lowering source-level types to LLVM

- **Source language types are lowered:**
  - Rich type systems expanded to simple type system
  - Implicit & abstract types are made explicit & concrete

- **Examples of lowering:**
  - References turn into pointers: \( T & \rightarrow T^* \)
  - Complex numbers: `complex float \rightarrow \{ float, float \}`
  - Bitfields: `struct X \{ int Y:4; int Z:2; \} \rightarrow \{ int \}`
  - Inheritance: `class T : S \{ int X; \} \rightarrow \{ S, int \}`
  - Methods: `class T \{ void foo(); \} \rightarrow void foo(T*)`

- **Same idea as lowering to machine code**
**LLVM Program Structure**

- **Module contains Functions/GlobalVariables**
  - Module is unit of compilation/analysis/optimization

- **Function contains BasicBlocks/Arguments**
  - Functions roughly correspond to functions in C

- **BasicBlock contains list of instructions**
  - Each block ends in a control flow instruction

- **Instruction is opcode + vector of operands**
  - All operands have types
  - Instruction result is typed

---

http://llvm.cs.uiuc.edu/
Our example, compiled to LLVM

```c
int callee(const int *X) {
    return *X + 1; // load
}
int caller() {
    int T; // on stack
    T = 4; // store
    return callee(&T);
}
```

```llvm
internal int %callee(int* %X) {
    %tmp.1 = load int* %X
    %tmp.2 = add int %tmp.1, 1
    ret int %tmp.2
}
int %caller() {
    %T = alloca int
    store int 4, int* %T
    %tmp.3 = call int %callee(int* %T)
    ret int %tmp.3
}
```

Linker “internalizes” most functions in most cases

http://llvm.cs.uiuc.edu/
Our example, desired transformation

```c
internal int %callee(int %X.val) {
  %tmp.2 = add int %X.val, 1
  ret int %tmp.2
}
int %caller() {
  %T = alloca int
  store int 4, int* %T
  %tmp.3 = call int %callee(%T)
  ret int %tmp.3
}
```

Other transformation (-mem2reg) cleans up the rest

```c
int %caller() {
  %tmp.3 = call int %callee(int 4)
  ret int %tmp.3
}
```
Tutorial Overview

- Introduction to the running example
- LLVM C/C++ Compiler Overview
  - High-level view of an example LLVM compiler
- The LLVM Virtual Instruction Set
  - IR overview and type-system
- LLVM C++ IR and important API’s
  - Basics, PassManager, dataflow, ArgPromotion
- Important LLVM Tools
  - opt, code generator, JIT, test suite, bugpoint
- Example applications of LLVM
LLVM Coding Basics

- **Written in modern C++, uses the STL:**
  - Particularly the vector, set, and map classes

- **LLVM IR is almost all doubly-linked lists:**
  - Module contains lists of Functions & GlobalVariables
  - Function contains lists of BasicBlocks & Arguments
  - BasicBlock contains list of Instructions

- **Linked lists are traversed with iterators:**

  ```
  Function *M = ...
  for (Function::iterator I = M->begin(); I != M->end(); ++I) {
      BasicBlock &BB = *I;
      ...
  }
  ```

See also: [docs/ProgrammersManual.html](http://llvm.cs.uiuc.edu/docs/ProgrammersManual.html)
LLVM Pass Manager

- Compiler is organized as a series of ‘passes’:
  - Each pass is one analysis or transformation

- Four types of Pass:
  - ModulePass: general interprocedural pass
  - CallGraphSCCPass: bottom-up on the call graph
  - FunctionPass: process a function at a time
  - BasicBlockPass: process a basic block at a time

- Constraints imposed (e.g. FunctionPass):
  - FunctionPass can only look at “current function”
  - Cannot maintain state across functions

See also: docs/WritingAnLLVMPass.html
Services provided by PassManager

- **Optimization of pass execution:**
  - Process a function at a time instead of a pass at a time.
  - Example: If F, G, H are three functions in input pgm: “FFFFFGGGGHHHH” not “FGHFGHFGHFGH”
  - Process functions in parallel on an SMP (future work)

- **Declarative dependency management:**
  - Automatically fulfill and manage analysis pass lifetimes.
  - Share analyses between passes when safe:
    - e.g. “DominatorSet live unless pass modifies CFG”

- **Avoid boilerplate for traversal of program**

See also: [docs/WritingAnLLVMPass.html](http://llvm.cs.uiuc.edu/docs/WritingAnLLVMPass.html)
Arg Promotion is a CallGraphSCCPass:
- Naturally operates bottom-up on the CallGraph
  - Bubble pointers from callees out to callers

```cpp
#include "llvm/CallGraphSCCPass.h"

struct SimpleArgPromotion : public CallGraphSCCPass {
  virtual void getAnalysisUsage(AnalysisUsage &AU) const {
    AU.addRequired<AliasAnalysis>();        // Get aliases
    AU.addRequired<TargetData>();           // Get data layout
    CallGraphSCCPass::getAnalysisUsage(AU); // Get CallGraph
  }
};
```

Arg Promotion requires AliasAnalysis info
- To prove safety of transformation
  - Works with any alias analysis algorithm though
Finally, implement `runOnSCC` (line 65):

```cpp
bool SimpleArgPromotion::
runOnSCC(const std::vector<CallGraphNode*> &SCC) {
  bool Changed = false, LocalChange;
  do {
    LocalChange = false;
    // Attempt to promote arguments from all functions in this SCC.
    for (unsigned i = 0, e = SCC.size(); i != e; ++i)
      LocalChange |= PromoteArguments(SCC[i]);
    Changed |= LocalChange;  // Remember that we changed something.
  } while (LocalChange);
  return Changed;            // Passes return true if something changed.
}
```

```cpp
static int foo(int ***P) {
  return ***P;
}
```

```cpp
static int foo(int P_val_val_val) {
  return P_val_val_val;
}
```
LLVM Dataflow Analysis

- LLVM IR is in SSA form:
  - use-def and def-use chains are always available
  - All objects have user/use info, even functions

- Control Flow Graph is always available:
  - Exposed as BasicBlock predecessor/successor lists
  - Many generic graph algorithms usable with the CFG

- Higher-level info implemented as passes:
  - Dominators, CallGraph, induction vars, aliasing, GVN, …

See also: docs/ProgrammersManual.html
Arg Promotion: safety check #1/4

#1: Function must be “internal” (aka “static”)

```cpp
88: if (!F || !F->hasInternalLinkage()) return false;
```

#2: Make sure address of F is not taken

- In LLVM, check that there are only direct calls using F

```cpp
99: for (Value::use_iterator UI = F->use_begin(); UI != F->use_end(); ++UI) {
    CallSite CS = CallSite::get(*UI);
    if (!CS.getInstruction()) // "Taking the address" of F.
        return false;
}
```

#3: Check to see if any args are promotable:

```cpp
114: for (unsigned i = 0; i != PointerArgs.size(); ++i) {
    if (!isSafeToPromoteArgument(PointerArgs[i]))
        PointerArgs.erase(PointerArgs.begin()+i);
    if (PointerArgs.empty()) return false; // no args promotable
```
#4: Argument pointer can only be loaded from:

- No stores through argument pointer allowed!

// Loop over all uses of the argument (use-def chains).
138: for (Value::use_iterator UI = Arg->use_begin();
    UI != Arg->use_end(); ++UI) {

  // If the user is a load:
  if (LoadInst *LI = dyn_cast<LoadInst>(*UI)) {

    // Don't modify volatile loads.
    if (LI->isVolatile()) return false;
    Loads.push_back(LI);
  } else {
    return false; // Not a load.
  }
}
#5: Value of "*P" must not change in the BB

- We move load out to the caller, value cannot change!

```cpp
    // Get AliasAnalysis implementation from the pass manager.
   AliasAnalysis &AA = getAnalysis<AliasAnalysis>();

    // Ensure *P is not modified from start of block to load
   if (AA.canInstructionRangeModify(BB->front(), *Load, Arg, LoadSize))
        return false;  // Pointer is invalidated!
```

See also: [docs/AliasAnalysis.html](http://llvm.cs.uiuc.edu/docs/AliasAnalysis.html)
175: for (pred_iterator PI = pred_begin(BB), E = pred_end(BB);
    PI != E; ++PI)  // Loop over predecessors of BB.
    // Check each block from BB to entry (DF search on inverse graph).
    for (idf_iterator<BasicBlock*> I = idf_begin(*PI);
        I != idf_end(*PI); ++I)
            // Might *P be modified in this basic block?
        if (AA.canBasicBlockModify(**I, Arg, LoadSize))
            return false;
Arg Promotion: xform outline #1/4

#1: Make prototype with new arg types: #197
   ❖ Basically just replaces ‘int*’ with ‘int’ in prototype

#2: Create function with new prototype:

214: Function *NF = new Function(NFTy, F->getLinkage(),
               F->getName());
       F->getParent()-->>getFunctionList().insert(F, NF);

#3: Change all callers of F to call NF:

     // If there are uses of F, then calls to it remain.
    221: while (!F->use_empty()) {
        // Get a caller of F.
        CallSite CS = CallSite::get(F->use_back());
Arg Promotion: xform outline #2/4

#4: For each caller, add loads, determine args

- Loop over the args, inserting the loads in the caller

220: `std::vector<Value*> Args;`

226: `CallSite::arg_iterator AI = CS.arg_begin();`
    `for (Function::aiterator I = F->abegin(); I != F->aend();`  
    `    ++I, ++AI)`
    `if (!ArgsToPromote.count(I)) // Unmodified argument.`
    `    Args.push_back(*AI);`  
    `else {`  
    `    // Insert the load before the call.`
    `    LoadInst *LI = new LoadInst(*AI, (*AI)->getName()+".val", Call); // Insertion point`
    `    Args.push_back(LI);`  
    `}`
Arg Promotion: xform outline #3/4

#5: Replace the call site of F with call of NF

    // Create the call to NF with the adjusted arguments.
    242: Instruction *New = new CallInst(NF, Args, "", Call);

    // If the return value of the old call was used, use the retval of the new call.
    if (!Call->use_empty())
        Call->replaceAllUsesWith(New);

    // Finally, remove the old call from the program, reducing the use-count of F.
    Call->getParent()->getInstList().erase(Call);

#6: Move code from old function to new Fn

    259: NF->getBasicBlockList().splice(NF->begin(),
        F->getBasicBlockList());
#7: Change users of F’s arguments to use NF’s

264: for (Function::aiterator I = F->abegin(), I2 = NF->abegin();
    I != F->aend(); ++I, ++I2)
    if (!ArgsToPromote.count(I)) { // Not promoting this arg?
        I->replaceAllUsesWith(I2);    // Use new arg, not old arg.
    } else {
        while (!I->use_empty()) {    // Only users can be loads.
            LoadInst *LI = cast<LoadInst>(I->use_back());
            LI->replaceAllUsesWith(I2);
            LI->getParent()->getInstList().erase(LI);
        }
    }

#8: Delete old function:

286: F->getParent()->getFunctionList().erase(F);
Tutorial Overview

- Introduction to the running example
- LLVM C/C++ Compiler Overview
  - High-level view of an example LLVM compiler
- The LLVM Virtual Instruction Set
  - IR overview and type-system
- LLVM C++ IR and important API’s
  - Basics, PassManager, dataflow, ArgPromotion
- Important LLVM Tools
  - opt, code generator, JIT, test suite, bugpoint
- Example applications of LLVM
LLVM tools: two flavors

- **“Primitive” tools: do a single job**
  - `llvm-as`: Convert from `.ll` (text) to `.bc` (binary)
  - `llvm-dis`: Convert from `.bc` (binary) to `.ll` (text)
  - `llvm-link`: Link multiple `.bc` files together
  - `llvm-prof`: Print profile output to human readers
  - `llvmc`: Configurable compiler driver

- **Aggregate tools: pull in multiple features**
  - `gccas/gccld`: Compile/link-time optimizers for C/C++ FE
  - `bugpoint`: automatic compiler debugger
  - `llvm-gcc/llvm-g++`: C/C++ compilers

See also: [docs/CommandGuide/](http://llvm.cs.uiuc.edu/docs/CommandGuide/)
opt tool: LLVM modular optimizer

- Invoke arbitrary sequence of passes:
  - Completely control PassManager from command line
  - Supports loading passes as plugins from .so files
    
    ```
    opt -load foo.so -pass1 -pass2 -pass3 x.bc -o y.bc
    ```

- Passes “register” themselves:
  
  ```
  61: RegisterOpt<SimpleArgPromotion> X("simpleargpromotion",
      "Promote 'by reference' arguments to 'by value'");
  ```

- From this, they are exposed through opt:
  
  ```
  > opt -load libsimpleargpromote.so -help
    ...
    -sccp         - Sparse Conditional Constant Propagation
    -simpleargpromotion - Promote 'by reference' arguments to 'by
    -simplifycfg    - Simplify the CFG
    ...
  ```

http://llvm.cs.uiuc.edu/
Running Arg Promotion with opt

- **Basic execution with ‘opt’:**
  - `opt -simpleargpromotion in.bc -o out.bc`
  - Load .bc file, run pass, write out results
  - Use “-load filename.so” if compiled into a library
  - PassManager resolves all dependencies

- **Optionally choose an alias analysis to use:**
  - `opt -basicaa -simpleargpromotion` (default)
  - Alternatively, `-steens-aa`, `-anders-aa`, `-ds-aa`, ...

- **Other useful options available:**
  - `-stats`: Print statistics collected from the passes
  - `-time-passes`: Time each pass being run, print output
Example -stats output (gccas 176.gcc)

... Statistics Collected ...

23426 adce - Number of instructions removed
1663 adce - Number of basic blocks removed
5052592 bytecode - Number of bytecode bytes written
57489 cfgsimplify - Number of blocks simplified
4186 constmerge - Number of global constants merged
211 dse - Number of stores deleted
15943 gcse - Number of loads removed
54245 gcse - Number of instructions removed
253 inline - Number of functions deleted because all callers found
3952 inline - Number of functions inlined
9425 instcombine - Number of constant folds
160469 instcombine - Number of insts combined
208 licm - Number of load insts hoisted or sunk
4982 licm - Number of instructions hoisted out of loop
350 loop-unroll - Number of loops completely unrolled
30156 mem2reg - Number of alloca's promoted
2934 reassociate - Number of insts with operands swapped
650 reassociate - Number of insts reassociated
67 scalarrepl - Number of allocas broken up
279 tailcallelim - Number of tail calls removed
25395 tailduplicate - Number of unconditional branches eliminated

http://llvm.cs.uiuc.edu/
Example -time-passes (gccas 176.gcc)

--- Pass execution timing report ---

---User Time--  --System Time--  --User+System--  ---Wall Time---  --- Name ---
16.2400 ( 23.0%)  0.0000 (  0.0%)  16.2400 ( 22.9%)  16.2192 ( 22.9%)  Global Common Subexpression Elimination
11.1200 ( 15.8%)  0.0499 ( 13.8%)  11.1700 ( 15.8%)  11.1028 ( 15.7%)  Reassociate expressions
6.5499 (  9.3%)   0.0300 (  8.3%)   6.5799 (  9.3%)   6.5824 (  9.3%)  Bytecode Writer
3.2499 (  4.6%)   0.0100 (  2.7%)   3.2599 (  4.6%)   3.2140 (  4.5%)  Scalar Replacement of Aggregates
3.0300 (  4.3%)   0.0499 ( 13.8%)   3.0800 (  4.3%)   3.0382 (  4.2%)  Combine redundant instructions
2.6599 (  3.7%)   0.0100 (  2.7%)   2.6699 (  3.7%)   2.7339 (  3.8%)  Dead Store Elimination
2.1600 (  3.0%)   0.0300 (  8.3%)   2.1900 (  3.0%)   2.1924 (  3.1%)  Function Integration/Inlining
2.1600 (  3.0%)   0.0100 (  2.7%)   2.1700 (  3.0%)   2.1125 (  2.9%)  Sparse Conditional Constant Propagation
1.6600 (  2.3%)   0.0000 (  0.0%)   1.6600 (  2.3%)   1.6389 (  2.3%)  Aggressive Dead Code Elimination
1.4999 (  2.1%)   0.0100 (  2.7%)   1.5099 (  2.1%)   1.4462 (  2.0%)  Tail Duplication
1.5000 (  2.1%)   0.0000 (  0.0%)   1.5000 (  2.1%)   1.4410 (  2.0%)  Post-Dominator Set Construction
1.3200 (  1.8%)   0.0000 (  0.0%)   1.3200 (  1.8%)   1.3722 (  1.9%)  Canonicalize natural loops
1.2700 (  1.8%)   0.0000 (  0.0%)   1.2700 (  1.7%)   1.2717 (  1.7%)  Merge Duplicate Global Constants
1.0300 (  1.4%)   0.0000 (  0.0%)   1.0300 (  1.4%)   1.1418 (  1.6%)  Combine redundant instructions
0.9499 (  1.3%)   0.0400 ( 11.1%)   0.9899 (  1.4%)   0.9979 (  1.4%)  Raise Pointer References
0.9399 (  1.3%)   0.0100 (  2.7%)   0.9499 (  1.3%)   0.9688 (  1.3%)  Simplify the CFG
0.9199 (  1.3%)   0.0300 (  8.3%)   0.9499 (  1.3%)   0.8993 (  1.2%)  Promote Memory to Register
0.9600 (  1.3%)   0.0000 (  0.0%)   0.9600 (  1.3%)   0.8742 (  1.2%)  Loop Invariant Code Motion
0.5600 (  0.7%)   0.0000 (  0.0%)   0.5600 (  0.7%)   0.6022 (  0.8%)  Module Verifier
...

http://llvm.cs.uiuc.edu/
Analyze tool: Visualize analysis results

- Print most LLVM data structures
  - Dominators, loops, alias sets, CFG, call graph, …
  - Converts most LLVM data structures to ‘dot’ graphs

http://llvm.cs.uiuc.edu/
LLC Tool: Static code generator

- Compiles LLVM $\rightarrow$ native assembly language
  - Currently for X86, Sparc, PowerPC (others in alpha)
  - `llc file.bc -o file.s -march=x86`
  - `as file.s -o file.o`

- Compiles LLVM $\rightarrow$ portable C code
  - `llc file.bc -o file.c -march=c`
  - `gcc -c file.c -o file.o`

- Targets are modular & dynamically loadable:
  - `llc -load libarm.so file.bc -march=arm`

http://llvm.cs.uiuc.edu/
The LLVM Code Generator

- **Target independent:**
  - Driven by an algorithm independent target description
  - Data layout, Register, Instruction, Scheduling, ...

- **Basic code generator layout:**
  - All passes are replaceable
    - e.g. Trivial to change and add register allocators
  - Targets can add custom passes
    - e.g. X86 has special support for FP stack

Exposes all target-specific details about a function (calling conventions, etc)

4 algorithms available today
llc -regalloc=foo

See also: docs/CodeGenerator.html
Porting LLVM to a new target

- **LLVM targets are very easy to write:**
  - Anecdotal evidence suggests 1 week for a basic port
  - … for someone familiar with the target machine and compilers in general, but not with LLVM

- **LLVM targets are written with “tablegen” tool**
  - Simple declarative syntax
  - Designed to factor out redundancy in target description

- **Some C++ code is still required**
  - Primarily in the instruction selector
  - Continuing work to improve this

See also: [docs/TableGenFundamentals.html](http://llv.cs.uiuc.edu/docs/TableGenFundamentals.html) and [WritingAnLLVMBackend.html](http://llv.cs.uiuc.edu/WritingAnLLVMBackend.html)
LLI Tool: LLVM Execution Engine

- LLI allows direct execution of .bc files
  - E.g.: lli grep.bc -i foo *.c

- LLI uses a Just-In-Time compiler if available:
  - Uses same code generator as LLC
    - Optionally uses faster components than LLC
  - Emits machine code to memory instead of “.s” file
  - JIT is a library that can be embedded in other tools

- Otherwise, it uses the LLVM interpreter:
  - Interpreter is extremely simple and very slow
  - Interpreter is portable though!
C and C++ Program Test Suite

- **Large collection of programs and benchmarks:**
  - Standard suites (e.g. SPEC 95/2000, Olden, Ptdist, McCat, Stanford, Freebench, Shootout…)
  - Individual programs: sgefa, siod, sim, pi, povray, …
  - Proprietary suites (e.g. SPEC) require suite source

- **Consistent build environment:**
  - Easy add hooks to build for profiling/instrumentation
  - Easy to get performance numbers from entire test suite

- **Entire test suite is checked every night:**
  - Hosted on Linux, Solaris, FreeBSD on X86, Sparc & PPC

See also: [docs/TestingGuide.html](http://llvm.cs.uiuc.edu/docs/TestingGuide.html)
Integrated Debugging Tools

- Extensive assertions throughout code
  - Find problems as early as possible (close to source)
- LLVM IR Verifier: Checks modules for validity
  - Checks type properties, dominance properties, etc.
  - Automatically run by opt
  - Problem found?: print an error message and abort
- LLVM IR Leak Detector
  - Efficient and simple “garbage collector” for IR objects
  - Ensure IR objects are deallocated appropriately
The Bugpoint automated bug finder

- Simple idea: automate ‘binary’ search for bug
  - Bug isolation: which passes interact to produce bug
  - Test case reduction: reduce input program

- Optimizer/Codegen crashes:
  - Throw portion of test case away, check for crash
    - If so, keep going
    - Otherwise, revert and try something else
  - Extremely effective in practice

- Simple greedy algorithms for test reduction

- Completely black-box approach

See also: docs/Bugpoint.html
Debugging Miscompilations

- **Optimizer miscompilation:**
  - Split testcase in two, optimize one. Still broken?
  - Keep shrinking the portion being optimized

- **Codegen miscompilation:**
  - Split testcase in two, compile one with CBE, broken?
  - Shrink portion being compiled with non CBE codegen

- **Code splitting granularities:**
  - Take out whole functions
  - Take out loop nests
  - Take out individual basic blocks
How well does this thing work?

- **Extremely effective:**
  - Can often reduce a 100K LOC program and 60 passes to a few basic blocks and 1 pass in 5 minutes
  - Crashes are found much faster than miscompilations
    - no need to run the program to test a reduction

- **Interacts with integrated debugging tools**
  - Runtime errors are detected faster

- **Limitations:**
  - Program must be deterministic
    - ... or modified to be so
  - Finds “a” bug, not “the” bug
Tutorial Overview

- Introduction to the running example
- LLVM C/C++ Compiler Overview
  - High-level view of an example LLVM compiler
- The LLVM Virtual Instruction Set
  - IR overview and type-system
- LLVM C++ IR and important API’s
  - Basics, PassManager, dataflow, ArgPromotion
- Important LLVM Tools
  - opt, code generator, JIT, test suite, bugpoint
- Example applications of LLVM
Use Case 1: Edge or Path Profiling

**Goal:** Profiling Research or PGO

- **Implementation:**
  - *FunctionPass:* LLVM-to-LLVM transformation
  - *Instrumentation:* Use CFG, intervals, dominators
  - *Code generation:* Use C or any native back end
  - *Profile feedback:* Use profile query interface

- **Core extensions needed:** *none*

- **Major LLVM Benefits**
  - Language-independence, CFG, very simple IR
Use Case 2: Alias Analysis

**Goal:** Research on new alias analysis algorithms

- **Implementation:**
  - *ModulePass:* Whole-program analysis pass on LLVM
  - Use type information; SSA; heap/stack/globals
  - Compare *SimpleAA*, *Steensgard’s*, *Andersen’s*, *DSA*
  - Evaluate many clients via *AliasAnalysis* interface

- **Core extensions needed:** *none*

- **Major LLVM Benefits**
  - Language-independence, type info, SSA, DSA, IPO
  - *AliasAnalysis* interface with many pre-existing clients
Use Case 3: LDS Prefetching

**Goal:** Prefetching linked data structures

- **Implementation:**
  - *ModulePass:* Link-time LLVM-to-LLVM transformation
  - *Code transformations:* use type info, loop analysis, unrolling, prefetch insertion
  - *Data transformations* (e.g., adding history pointers): use *strong* type info from DSA, IPO

- **Core extensions needed:**
  - Prefetch operation: add as *intrinsic* (in progress)

- **Major LLVM Benefits**
  - Language-independence, type info, DSA, IPO
Use Case 4: Language Front end

**Goal:** Use LLVM to implement a new language

- **Implementation:**
  - Parser (say to AST), Semantic checking
  - AST-to-LLVM translator

- **Core extensions needed:** depends
  - High-level type system is omitted by design

- **Major LLVM Benefits**
  - Low-level, but powerful type system
  - Very simple IR to generate (e.g., compare GCC RTL)
  - Extensive global and IP optimization framework
  - JIT engine, native back-ends, C back-end
Use Case 5: JIT Compiler

**Goal:** Write JIT compiler for a bytecode language

- **Implementation:**
  - Extend the LLVM JIT framework
  - *Simple JIT:* Fast translation from bytecode to LLVM (then use LLVM JIT + GC)
  - *Optimizing JIT:* Language-specific optimizations + fast translation (then use LLVM optimizations, JIT, GC)

- **Core extensions needed:** *none in general*

- **Major LLVM Benefits**
  - Compact, typed, language-independent IR
  - Existing JIT framework and GC
Use Case 6: Architecture Research

**Goal:** Compiler support for new architectures

- **Implementation:**
  - Add new machine description (or modify one)
  - Add any new LLVM-to-LLVM transformations

- **Core extensions needed:** depends on goals
  - Imminent features: *modulo sched; vector ops*

- **Major LLVM Benefits**
  - Low-level, typed, machine-independent IR
  - Explicit register/memory architecture
  - Aggressive mid-level and back-end compiler framework
  - *Full-system evaluation:* applications, libraries, even OS
Five point LLVM Review

- **Extremely simple IR to learn and use**
  - 1-to-1 correspondence between .ll, .bc, and C++ IR
  - Very positive user reactions

- **Powerful and modular optimizer**
  - Easy to extend, or just use what is already there

- **Clean and modular code generator**
  - Easy to retarget, easy to replace/tweak components

- **Many “productivity tools” (bugpoint, verifier)**
  - Get more done, quicker!

- **Active dev community, good documentation**
  - Mailing lists, IRC, doxygen, extensive docs
Get started with LLVM!

- Download latest release or CVS:  
  http://llvm.cs.uiuc.edu/releases/

- Follow the “Getting Started Guide”:  
  http://llvm.cs.uiuc.edu/docs/GettingStarted.html
  - Walks you through install and setup
  - Lots of other docs available in “docs” directory
  - Join us on mailing lists and IRC

- Happy hacking!

http://llvm.cs.uiuc.edu/  
Chris Lattner – lattner@cs.uiuc.edu