

The LLVM Compiler Framework and Infrastructure

15-745: Optimizing Compilers

David Koes

1/22/2008

Substantial portions courtesy Chris Lattner and Vikram Adve

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++ gcc frontend
- ❖ Backends for C, X86, Sparc, PowerPC, Alpha, Arm, Thumb, IA-64...

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, ...

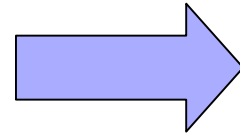
Tutorial Overview

- **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**

Running example: arg promotion

Consider use of by-reference parameters:

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



compiles to

```
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:

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

- ✓ 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**

The LLVM C/C++ Compiler

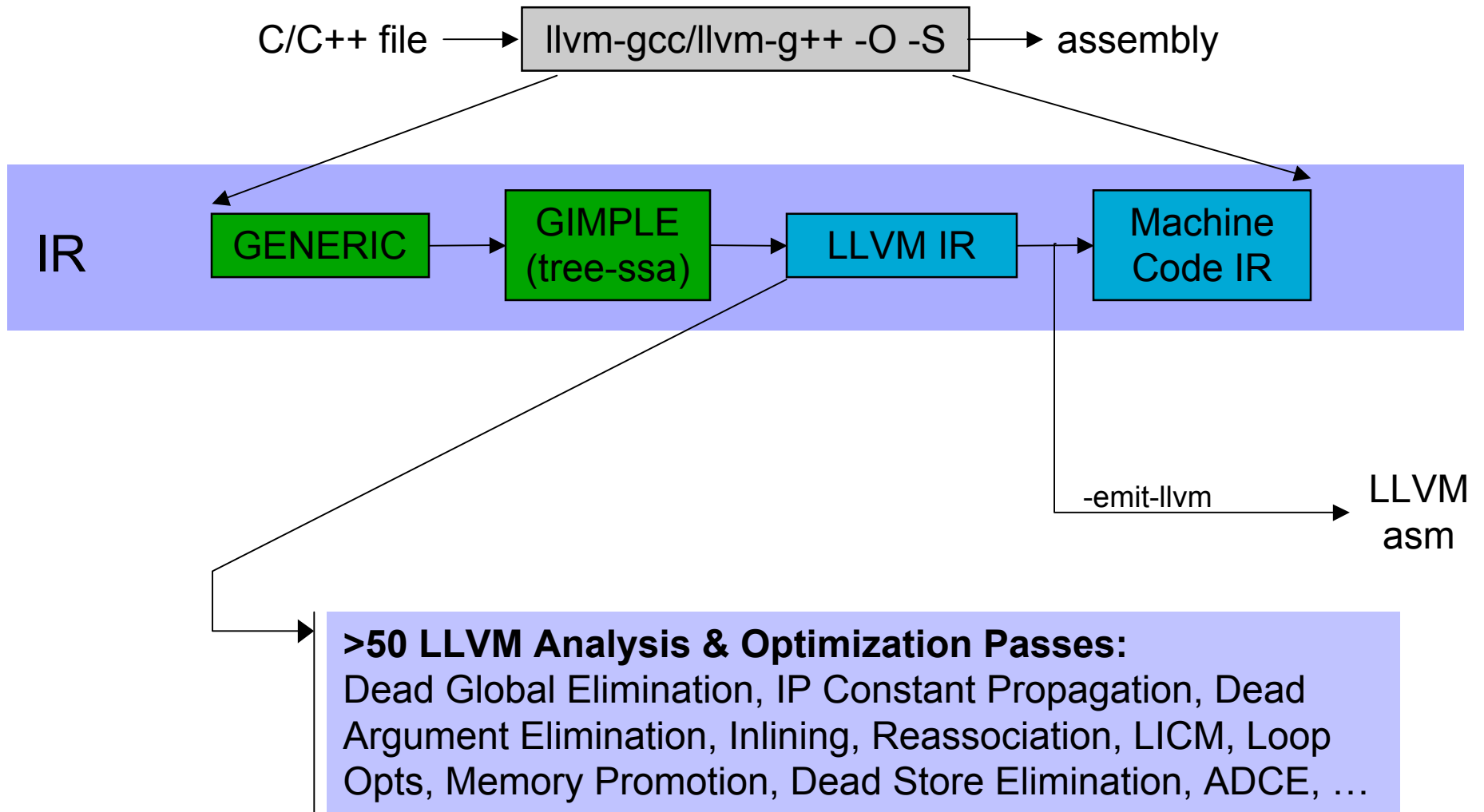
- **From the high level, it is a standard compiler:**

- ❖ Compatible with standard makefiles
- ❖ Uses GCC 4.2 C and C++ parser
- ❖ Generates native executables/object files/assembly

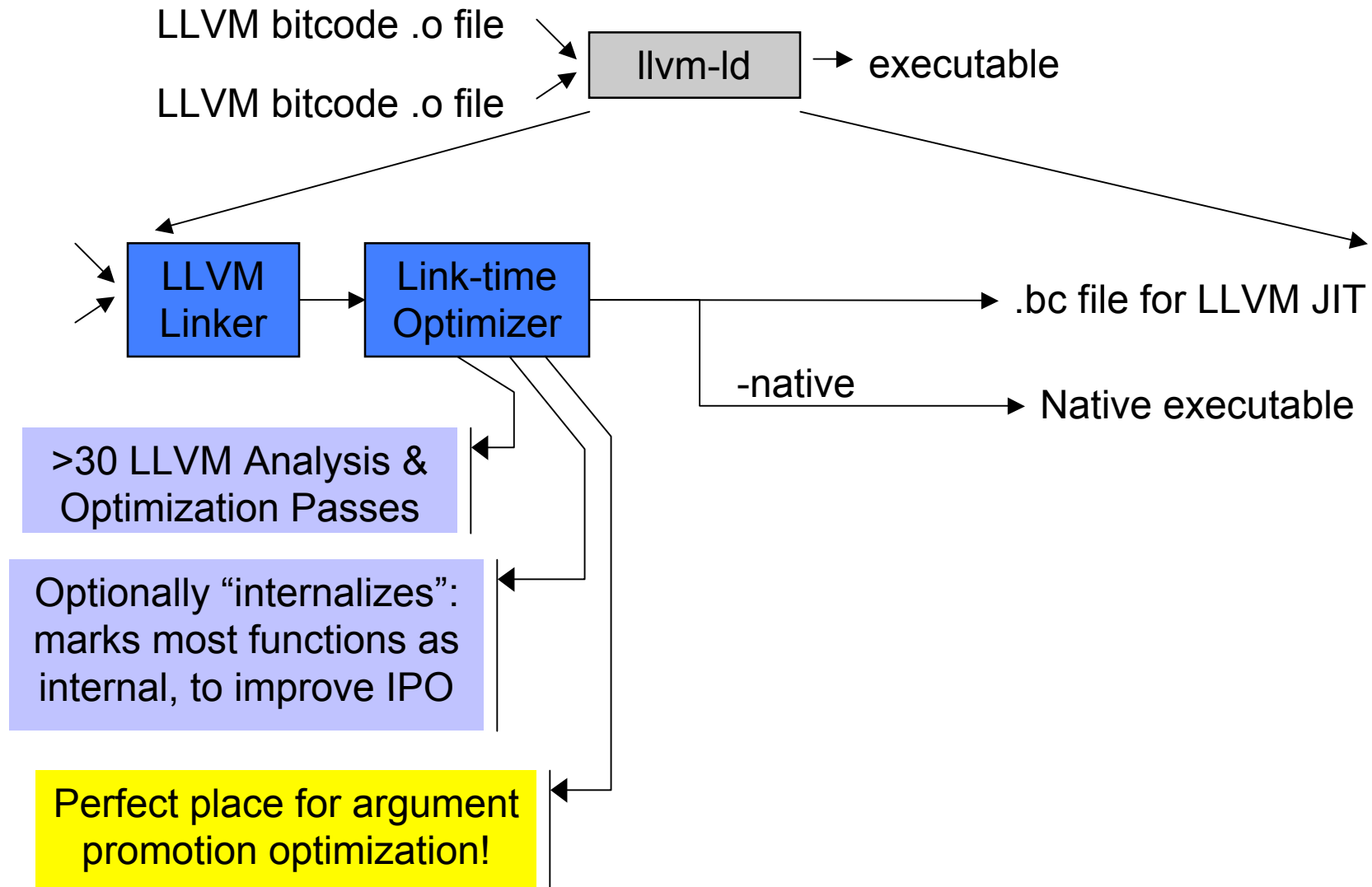
- **Distinguishing features:**

- ❖ Uses LLVM optimizers, not GCC optimizers
- ❖ Pass -emit-llvm to output LLVM IR
 - -S: human readable “assembly”
 - -c: efficient “bitcode” binary

Looking into events at compile-time



Looking into events at link-time



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**

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

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

```
bb:                                ; preds = %bb, %entry  
    %i.1 = phi i32 [ 0, %entry ], [ %i.2, %bb ]  
    %AiAddr = getelementptr float* %A, i32 %i.1  
    call void @Sum( float* %AiAddr, %pair* %P )  
    %i.2 = add i32 %i.1, 1  
    %exitcond = icmp eq i32 %i.2, %N  
    br i1 %exitcond, label %return, label %bb
```

LLVM Instruction Set Overview #2

■ 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

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

```
bb:                                ; preds = %bb, %entry  
    %i.1 = phi i32 [ 0, %entry ], [ %i.2, %bb ]  
    %AiAddr = getelementptr float* %A, i32 %i.1  
    call void @Sum( float* %AiAddr, %pair* %P )  
    %i.2 = add i32 %i.1, 1  
    %exitcond = icmp eq i32 %i.2, %N  
    br i1 %exitcond, label %return, label %bb
```

LLVM Type System Details

■ The entire type system consists of:

- ❖ Primitives: integer, floating point, label, void
 - no “signed” integer types
 - arbitrary bitwidth integers (i32, i64, i1)
- ❖ Derived: pointer, array, structure, function, vector,...
- ❖ 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](https://llvm.org/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& → T*`
- ❖ Complex numbers: `complex float → { float, float }`
- ❖ Bitfields: `struct X { int Y:4; int Z:2; } → { i32 }`
- ❖ Inheritance: `class T : S { int X; } → { S, i32 }`
- ❖ Methods: `class T { void foo(); } → 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

Our example, compiled to LLVM

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

All loads/stores are
explicit in the LLVM
representation

```
define internal i32 @callee(i32* %X) {  
entry:  
    %tmp2 = load i32* %X  
    %tmp3 = add i32 %tmp2, 1  
    ret i32 %tmp3  
}  
  
define internal i32 @caller() {  
entry:  
    %T = alloca i32  
    store i32 4, i32* %T  
    %tmp1 = call i32 @callee( i32* %T )  
    ret i32 %tmp1  
}
```

Our example, desired transformation

```
define i32 @callee(i32* %X) {  
  %tmp2 = load i32* %X  
  %tmp3 = add i32 %tmp2, 1  
  ret i32 %tmp3  
}  
  
define i32 @caller() {  
  %T = alloca i32  
  store i32 4, i32* %T  
  %tmp1 = call i32 @callee( i32* %T )  
  ret i32 %tmp1  
}
```

Other transformation
(-mem2reg) cleans up
the rest

```
define internal i32 @callee1(i32 %X.val)  
{  
  %tmp3 = add i32 %X.val, 1  
  ret i32 %tmp3  
}  
  
define internal i32 @caller() {  
  %T = alloca i32  
  store i32 4, i32* %T  
  %Tval = load i32* %T  
  %tmp1 = call i32 @callee1( i32 %Tval )  
  ret i32 %tmp1  
}
```



```
define internal i32 @caller() {  
  %tmp1 = call i32 @callee1( i32 4 )  
  ret i32 %tmp1  
}
```

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**

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

LLVM Coding Basics cont.

■ BasicBlock doesn't provide a reverse iterator

- ❖ Highly obnoxious when doing the assignment

```
for(BasicBlock::iterator I = bb->end(); I != bb->begin(); ) {  
    --I;  
    Instruction *insn = I;  
    ...  
}
```

■ Traversing successors of a BasicBlock:

```
for (succ_iterator SI = succ_begin(bb), E = succ_end(bb);  
     SI != E; ++SI) {  
    BasicBlock *Succ = *SI;  
    ...  
}
```

■ C++ is not Java

- primitive class variable not a
- you must manage memory
- virtual vs. non-virtual functions
- and much much more...

valgrind to the rescue!

<http://valgrind.org>

LLVM Pass Manager

- **Compiler is organized as a series of ‘passes’:**
 - ❖ Each pass is one analysis or transformation
- **Types of Pass:**
 - ❖ **ModulePass**: general interprocedural pass
 - ❖ **CallGraphSCCPass**: bottom-up on the call graph
 - ❖ **FunctionPass**: process a function at a time
 - ❖ **LoopPass**: process a natural loop 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](https://llvm.org/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:
“FFFFGGGGHHHH” 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

Pass Manager + Arg Promotion #1/2

■ Arg Promotion is a CallGraphSCCPass:

- ❖ Naturally operates bottom-up on the CallGraph
 - Bubble pointers from callees out to callers

```
24: #include "llvm/CallGraphSCCPass.h"  
47: struct SimpleArgPromotion : public CallGraphSCCPass {
```

■ Arg Promotion requires AliasAnalysis info

- ❖ To prove safety of transformation
 - Works with any alias analysis algorithm though

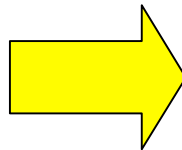
```
48: virtual void getAnalysisUsage(AnalysisUsage &AU) const {  
    AU.addRequired<AliasAnalysis>();           // Get aliases  
    AU.addRequired<TargetData>();             // Get data layout  
    CallGraphSCCPass::getAnalysisUsage(AU);   // Get CallGraph  
}
```

Pass Manager + Arg Promotion #2/2

■ Finally, implement `runOnSCC` (line 65):

```
bool SimpleArgPromotion::
runOnSCC(const std::vector<CallGraphNode*> &SCC) {
    bool Changed = false, LocalChange;
    do {    // Iterate until we stop promoting from this SCC.
        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.
}
```

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



```
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”)

```
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

```
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:

```
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
```

Arg Promotion: safety check #2/4

#4: Argument pointer can only be loaded from:

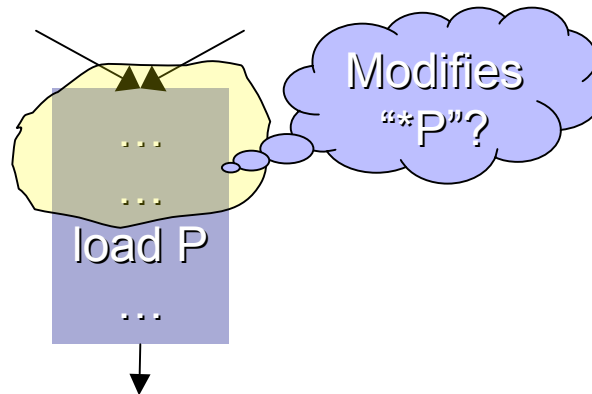
- ❖ No stores through argument pointer allowed!

```
// 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.
    }
}
```

Arg Promotion: safety check #3/4

#5: Value of “*P” must not change in the BB

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

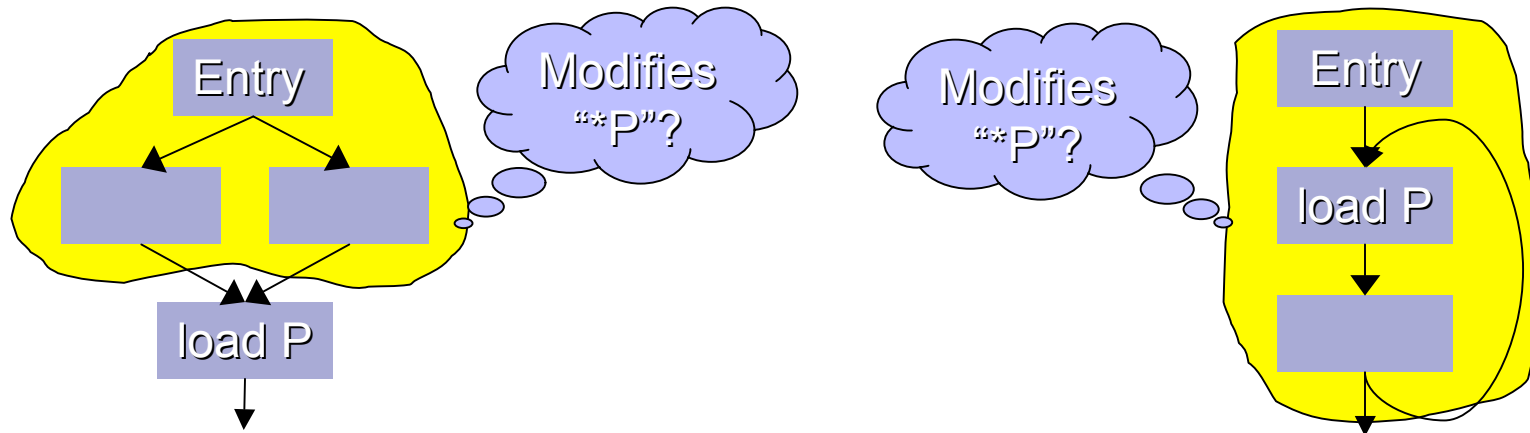


```
// Get AliasAnalysis implementation from the pass manager.  
156: AliasAnalysis &AA = getAnalysis<AliasAnalysis>();  
  
// Ensure *P is not modified from start of block to load  
169: if (AA.canInstructionRangeModify(BB->front(), *Load,  
                                     Arg, LoadSize))  
    return false; // Pointer is invalidated!
```

See also: [docs/AliasAnalysis.html](https://docs/llvm.org/ReleaseDocs/AA.html)

Arg Promotion: safety check #4/4

#6: “*P” cannot change from Fn entry to BB



```
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());
```

Arg Promotion: xform outline #4/4

#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**

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

- ❖ bugpoint: automatic compiler debugger
- ❖ llvm-gcc/llvm-g++: C/C++ compilers

See also: [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  
...
```

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 (176.gcc)

```
====-----
... Statistics Collected ...
====-----

23426 adce          - Number of instructions removed
1663 adce          - Number of basic blocks removed
5052592 bytecodewriter - 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
.....
```

Example -time-passes (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

...

LLC Tool: Static code generator

■ Compiles LLVM → native assembly language

❖ `llc file.bc -o file.s -march=x86`

❖ `as file.s -o file.o`

■ Compiles LLVM → 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`

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!

Assignment 1

■ Due Thursday, Jan 31

- ❖ Start Early
- ❖ Finish Early
- ❖ Go Have Fun
- ❖ Questions?

