Hacking LLVM for Grad Students

Scott A. Carr

Purdue University

hexhive

ACM Software System Award 2012

- **For research, LLVM replaced GCC**
- Used for:
  - Just-in-time compilers
  - Secure browser extensions
  - **Language virtual machines**
  - Static analysis
  - Automatic vectorization
  - GPU programming
  - Software verification
  - Embedded code generators
  - **Language implementations**

Contents

- LLVM Basics
- Your First Pass
- Your Second Pass
- Your N^{th} Pass
- Your Runtime Support Library
- Misc. Tips
- Conclusion

LLVM Basics

Read (some of) the docs

1. Language Reference
   - *Describes IR instructions*
2. Programmer's Manual
   - *How to use LLVM API*
3. Writing an LLVM Pass
4. Community
   - *Brandon Holt*
   - *Adrian Sampson*
   - *Mailing lists*
5. Doxygen

LLVM IR's 3 Forms

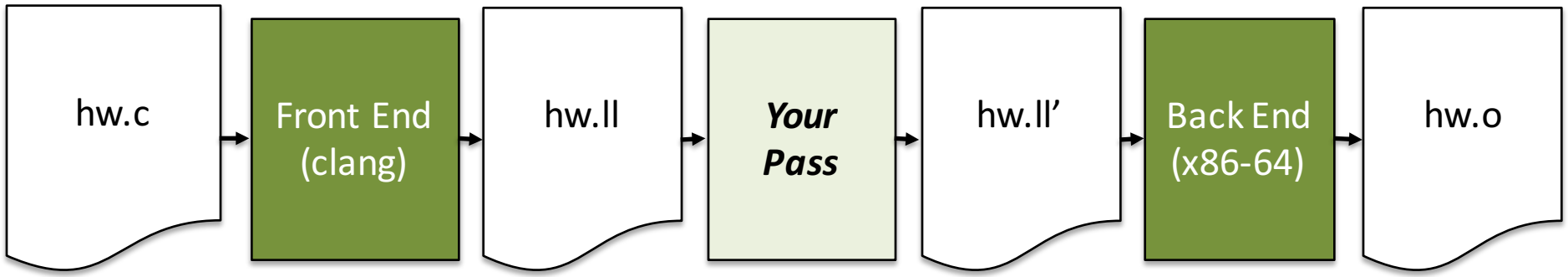
- LLVM is a library for manipulating IR
- Static Single Assignment (SSA)
- Three forms:
  1. In-memory
  2. On-disk bitcode
  3. Human readable assembly

```
%x = add i32 1, %y
```

What Passes Do

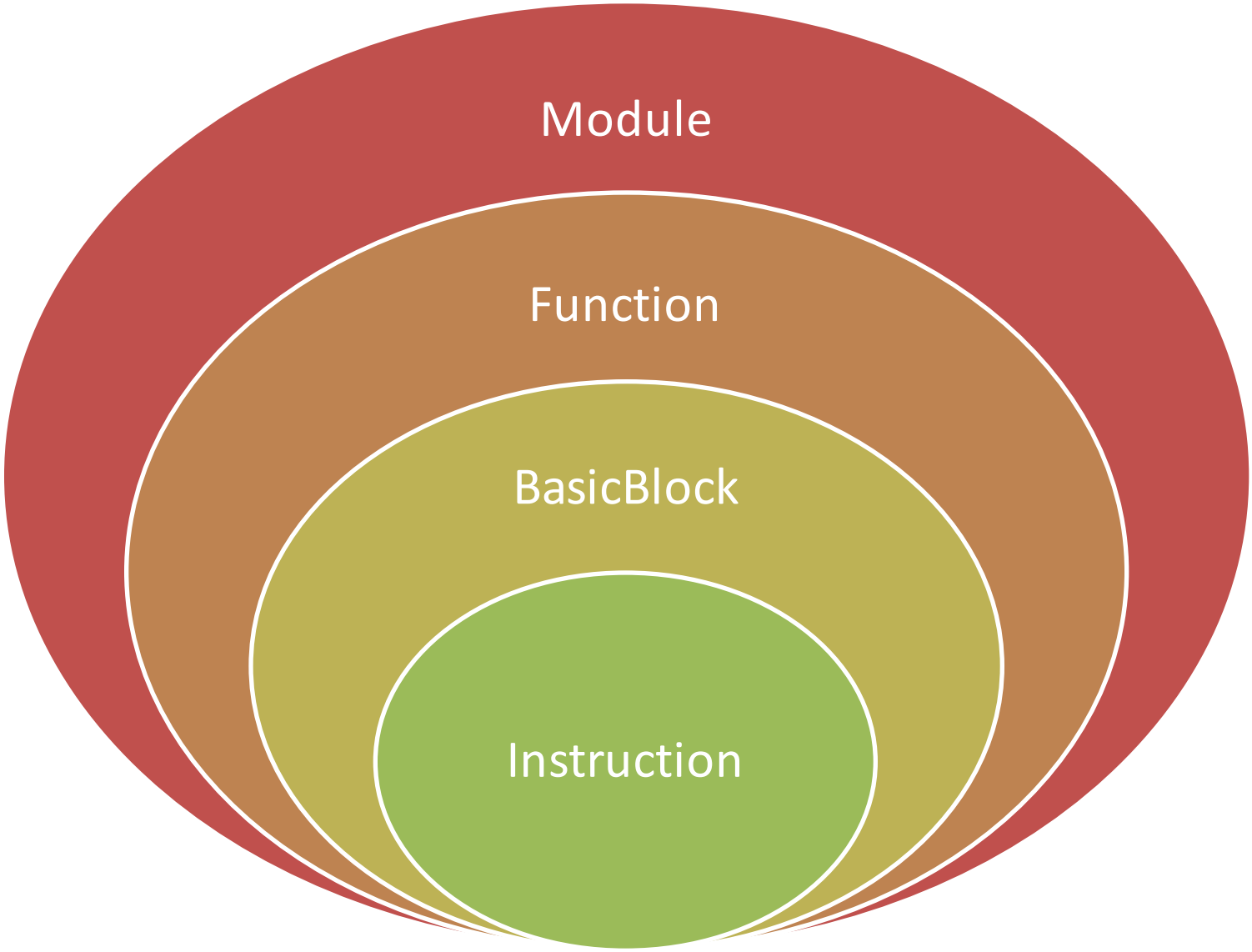
Pass: a class that,

- modifies or analyzes IR and
- is invoked by PassManager



The Important Instructions

LLVM IR	C
`LoadInst`	`x = *y;`
`StoreInst`	`*x = y;`
`GetElementPtrInst`	`obj.x // obj+offset(x)`
`CallInst`	`Foo()`
`BitCastInst`	`(void*)ptr`

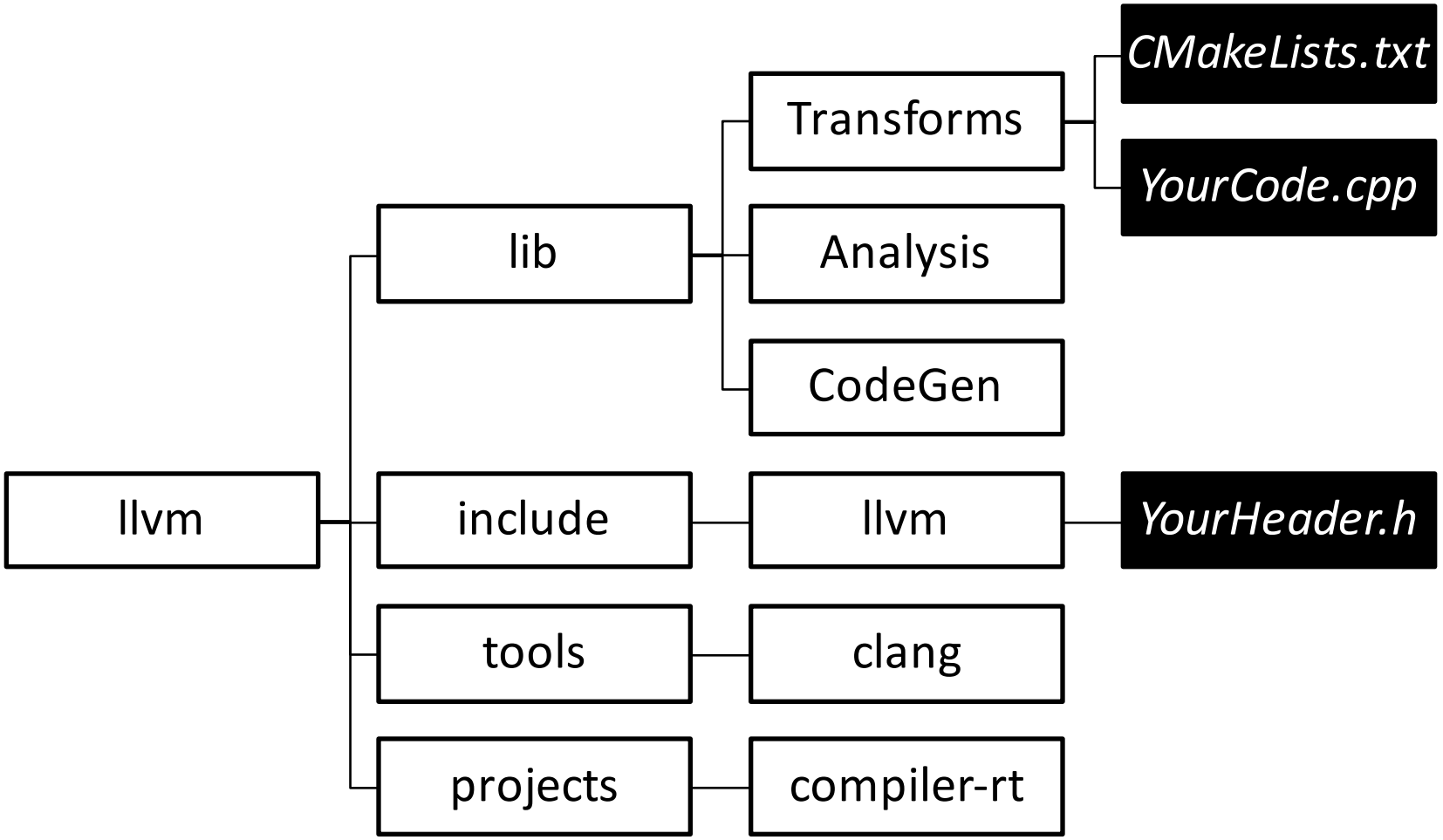

Module
Function
BasicBlock
Instruction

Your First Pass

Your First Pass

- For each function *foo*
  - Print "*Hello: foo*"
- From *Writing an LLVM Pass*

Directory Structure



```
#include "llvm/Pass.h”
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

namespace {
    struct Hello : public FunctionPass
    {
        static char ID;
        Hello() : FunctionPass(ID) {}
        bool runOnFunction(Function &F) override {
            errs() << "Hello: ";
            errs().write_escaped(F.getName()) << '\n';
            return false;
        }
    };
}
char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass", false,
    false);
```

```
#include "llvm/Pass.h”
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

namespace {
    struct Hello : public FunctionPass
    {
        static char ID;
        Hello() : FunctionPass(ID) {}
        bool runOnFunction(Function &F) override {
            errs() << "Hello: ";
            errs().write_escaped(F.getName()) << '\n';
            return false;
        }
    };
}
char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass", false,
    false);
```

```
#include "llvm/Pass.h”
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

namespace {
    struct Hello : public FunctionPass
    {
        static char ID;
        Hello() : FunctionPass(ID) {}
        bool runOnFunction(Function &F) override {
            errs() << "Hello: ";
            errs().write_escaped(F.getName()) << '\n';
            return false;
        }
    };
}
char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass", false,
    false);
```

```
#include "llvm/Pass.h”
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

namespace {
    struct Hello : public FunctionPass
    {
        static char ID;
        Hello() : FunctionPass(ID) {}
        bool runOnFunction(Function &F) override {
            errs() << "Hello: ";
            errs().write_escaped(F.getName()) << '\n';
            return false;
        }
    };
}
char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass", false,
    false);
```

Building

- Use CMake
- BUILD_SHARED_LIBS=ON
- CMAKE_BUILD_TYPE=Debug
- CMAKE_C_FLAGS=-fstandalone-debug
- Use clang (top tree) as your C/C++ compiler

Running Your Pass

```
$ clang –S –emit-llvm pro.c
```

Compile a bitcode version of pro.c, pro.ll

```
$ opt –load Hello.so -hello < pro.ll > pro2.ll
```

Run your pass on pro.ll and produce pro2.ll

```
$ clang pro2.ll
```

Compile pro2.ll to a binary

```
; Module p.ll

define i32 @foo(i32 %x) {
    …
}

define i32 @bar(i32 %x) {
    …
}

define i32 @bar(i32 %x) {
    …
}
```

```
$ opt -load Hello.so -hello < p.ll

Hello: foo
Hello: bar
Hello: baz

$
```

Your Second Pass

Your Second Pass

- For each `StoreInst`,
  - Get its value and pointer operands
  - Call a function, passing these two operands to the function

```
bool runOnFunction(Function& F) {
    for (auto& basicBlock : F) {
        vector<StoreInst> stores;
        for (auto& I : basicBlock) {
            if (auto store = dyn_cast<StoreInst>(&I))  {
                stores.push_back(store);
            }
        }
        for (auto store : stores) {
            // do something
        }
    }
}
```

```
bool runOnFunction(Function& F) {
    for (auto& basicBlock : F) {
        vector<StoreInst> stores;
        for (auto& I : basicBlock) {
            if (auto store = dyn_cast<StoreInst>(&I)) {
                stores.push_back(store);
            }
        }
        for (auto store : stores) {
            // do something
        }
    }
}
```

```
bool runOnFunction(Function& F) {
    for (auto& basicBlock : F) {
        vector<StoreInst> stores;
        for (auto& I : basicBlock) {
            if (auto store = dyn_cast<StoreInst>(&I)) {
                stores.push_back(store);
            }
        }
        for (auto store : stores) {
            // do something
        }
    }
}
```

```
bool runOnFunction(Function& F) {
    for (auto& basicBlock : F) {
        vector<StoreInst> stores;
        for (auto& I : basicBlock) {
            if (auto store = dyn_cast<StoreInst>(&I)) {
                stores.push_back(store);
            }
        }
        for (auto store : stores) {
            // do something
        }
    }
}
```

```
bool runOnFunction(Function& F) {
    for (auto& basicBlock : F) {
        vector<StoreInst> stores;
        for (auto& I : basicBlock) {
            if (auto store = dyn_cast<StoreInst>(&I)) {
                stores.push_back(store);
            }
        }
        for (auto store : stores) {
            // do something
        }
    }
}
```

```
bool runOnFunction(Function& F) {
    for (auto& basicBlock : F) {
        vector<StoreInst> stores;
        for (auto& I : basicBlock) {
            if (auto store = dyn_cast<StoreInst>(&I)) {
                stores.push_back(store);
            }
        }
        for (auto store : stores) {
            // do something
        }
    }
}
```

Doing Something

Create:

`IRBuilder::CreateXXX(...)`

Needs an insertion point

Replace:

`Instruction::replaceAllUsesWith(Instruction *)`

`Uses` and `Users` lists

Delete:

`Instruction::eraseFromParent()`

```
bool runOnFunction(Function& F) {
    …


    for (auto store : stores) {
        // do something
        IRBuilder<> IRB(store);
        auto val = store->getValueOperand();
        auto ptr = store->getPointerOperand();
        auto val2 = IRB.CreateBitCast(val, myValType);
        auto ptr2 = IRB.CreateBitCast(ptr, myPtrType);
        auto fnCall = IRB.CreateCall(myStoreFn, {val2, ptr2});
    }
}
```

```
bool runOnFunction(Function& F) {
    …


    for (auto store : stores) {
        // do something
        IRBuilder<> IRB(store);
        auto val = store->getValueOperand();
        auto ptr = store->getPointerOperand();
        auto val2 = IRB.CreateBitCast(val, myValType);
        auto ptr2 = IRB.CreateBitCast(ptr, myPtrType);
        auto fnCall = IRB.CreateCall(myStoreFn, {val2, ptr2});
    }
}
```

```
bool runOnFunction(Function& F) {
    …


    for (auto store : stores) {
        // do something
        IRBuilder<> IRB(store);
        auto val = store->getValueOperand();
        auto ptr = store->getPointerOperand();
        auto val2 = IRB.CreateBitCast(val, myValType);
        auto ptr2 = IRB.CreateBitCast(ptr, myPtrType);
        auto fnCall = IRB.CreateCall(myStoreFn, {val2, ptr2});
    }
}
```

```
bool runOnFunction(Function& F) {
    …


    for (auto store : stores) {
        // do something
        IRBuilder<> IRB(store);
        auto val = store->getValueOperand();
        auto ptr = store->getPointerOperand();
        auto val2 = IRB.CreateBitCast(val, myValType);
        auto ptr2 = IRB.CreateBitCast(ptr, myPtrType);
        auto fnCall = IRB.CreateCall(myStoreFn, {val2, ptr2});
    }
}
```

```
bool runOnFunction(Function& F) {
    …


    for (auto store : stores) {
        // do something
        IRBuilder<> IRB(store);
        auto val = store->getValueOperand();
        auto ptr = store->getPointerOperand();
        auto val2 = IRB.CreateBitCast(val, myValType);
        auto ptr2 = IRB.CreateBitCast(ptr, myPtrType);
        auto fnCall = IRB.CreateCall(myStoreFn, {val2, ptr2});
    }
}
```

```
; Module p.ll

...

store %x, %y
```

```
$ opt -load Hello.so -hello < p.ll > p2.ll

$ cat p2.ll
```

```
; Module p2.ll

...
%x2 = bitcast i8* %x
%y2 = bitcast i8** %y
call void @myStoreFn(%x2, %y2)
store i32* %x, i32** %y
```

Your Nth Pass

Your Nth Pass

- Use an analysis
  - CFG, CallGraph, `AliasAnalysis`, etc.
- Get the size of variables with `DataLayout`
- Examine library functions with `TargetLibraryInfo`
- Add a command line option
- Use `DebugInfo` or `Metadata`

Pass Manager and Analysis

- BLACK MAGIC!
- Copy another pass
- In `getAnalysisUsage`, `AnalysisUsage.addRequired(…)`
- `INITIALIZE_PASS_DEPENDENCY(…)`

- Module Passes can use Function Analysis
  - `getAnalysis<FnAnalysis>(llvm::Function *)`
- Lots of things are already implemented for you!
- Many analyses are “best effort!”
  - False negatives, not false positives

DataLayout

- `Module::getDataLayout()`
- Query the size of a type
  - Like `sizeof` in C
- Get the offset of a `struct` field
  - For `GetElementPtr`

TargetLibraryInfo

- `getAnalysis<TargetLibraryInfoWrapperPass>().getTLI()`

- What library functions are available?
- isMallocLikeFn
- isFreeCall
- isOperatorNewLikeFn

Add a Command Line Option

- `cl::opt <T> VarName(…);`
  - Arguments:
    1. “flagname”
    2. cl::desc(“my description”)
    3. cl::init(default_val)
- Now, just use `VarName`
- To set flag:

  `-mllvm –flagname <value>`
- See: CommandLine 2.0 Library Manual

DebugInfo

- Defines relationship between source code and generated code (IR)
- Many-to-one mapping
- API just changed
- Get the source line # that generated an IR instruction

Metadata

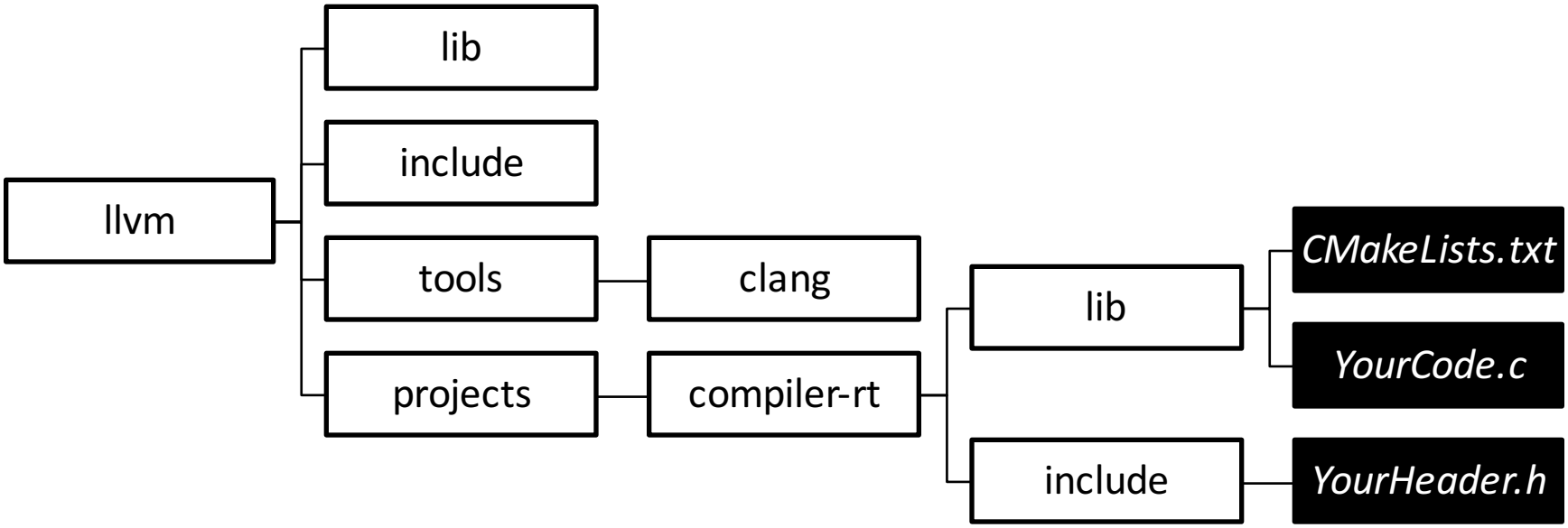
- Information about the IR
- `__attribute__((annotate(…)))` is metadata
- Not available at runtime
- Ex: Type Based Alias Analysis

Your Runtime Support Library

Your Runtime Support Library

- Add new built-in functions
- Insert calls to your functions with `IRBuilder`
- Written in C (for clang)

Runtime Directory Structure



Using Your Runtime

- `Module::getOrInsertFunction(…)`
- `IRB.CreateCall(`

```
yourFn,
{arg1, ... argN});
```

Misc. Tips

Misc. Tips

- Save the IR before and after your pass with `raw_fd_ostream`
- Eventually, run your pass from within clang/LLVM
  - Find a pass that runs at the same stage and copy it
- Debug with LLDB
- Use LTO for whole program analysis

LLDB

- `clang -v` to get the command
- `lldb -- <the command>`
- `X->dump()` while single stepping
- See Holt's website for more

LTO

- Link Time Optimization
- Why we care: whole program analysis
- Building complex software with LTO requires hacking the build
  - Ex: Chrome

Conclusion

Conclusion

- Use LLVM to “do stuff with code”
- Find a pass that does something similar and tweak it
- Community is one of LLVM’s strengths
  - i.e., use Google

Questions?

hexhive

Links

- Adrian Sampson
  - http://adriansampson.net/blog/llvm.html
- Brandon Holt
  - http://homes.cs.washington.edu/~bholt/posts/llvm-debugging.html
- LLVM Weekly (Alex Bradbury)
  - http://llvmweekly.org/
- Tillman Sheller
  - http://blogs.s-osg.org/an-introduction-to-accelerating-your-build-with-clang/
  - http://blogs.s-osg.org/a-conclusion-to-accelerating-your-build-with-clang/
- Dev Meetings
  - http://llvm.org/devmtg/
- Google for the LLVM Docs ;)
- Me: http://scottandrewcarr.com or @ScottCarr

EXTRA SLIDES

Example: Code Pointer Integrity

- Published in OSDI ’14
- Prevents control-flow hijack attacks by ensuring the integrity of function pointers, return addresses, etc.
- Protects a complete OS (FreeBSD)
- Implemented as LLVM pass and runtime

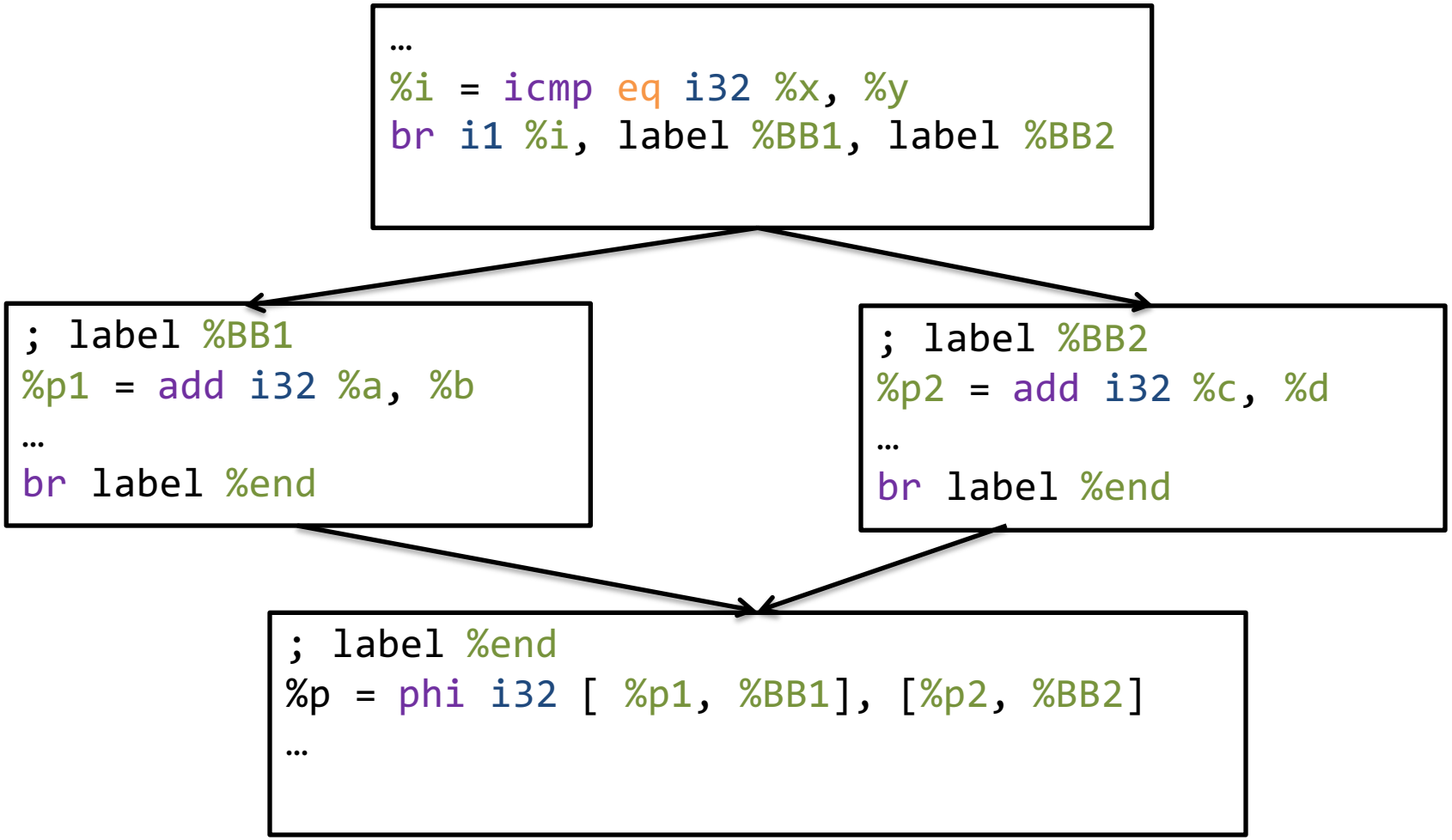
ACM Software System Award 2012

Due to its clean and flexible design and easy to use programming interfaces, **LLVM has quickly replaced GCC** as the infrastructure of choice for doing research on program translation, optimization, and analysis. **Researchers routinely use it for projects as diverse** as building link-time interprocedural optimizers, just-in-time compilers, secure browser extensions, language virtual machines, static analysis tools, automatic vectorization, GPU programming, software verification, hardware synthesis tools, embedded code generators, and numerous language implementations.

Example: Data Confidentiality & Integrity

- Provide strong *(high overhead)* protection for some data and weaker *(lower overhead)* protection for other data
- Pass
  - Identifies **sensitive variables**
  - Inserts a bounds check before each load/store
- Runtime
  - Bounds metadata

PHINode


…
%i = icmp eq i32 %x, %y
br i1 %i, label %BB1, label %BB2
; label %BB1
%p1 = add i32 %a, %b
…
br label %end
; label %BB2
%p2 = add i32 %c, %d
…
br label %end
; label %end
%p = phi i32 [%p1, %BB1], [%p2, %BB2]
…