# Building An Efficient JIT

the best kind of jit

Nate Begeman

Apple Inc

August 1st, 2008

# Overview

- Resources
- JIT Basics
- Clang-based JIT
- An *Efficient* Clang-based JIT

# Resources

## Memory, Cycles, Etc.

*How much are we using?*

# CPU

| | CPU | TOTAL | RPRVT |
|---|---|---|---|
| myjit | 50ms | 100ms | 41928K |

# TOTAL

| | CPU | TOTAL | RPRVT |
|---|---|---|---|
| myjit | 50ms | 100ms | 41928K |

# RPRVT

| | CPU | TOTAL | RPRVT |
|---|---|---|---|
| myjit | 50ms | 100ms | 41928K |

# JIT Basics

## JIT 101

*Common JIT Tasks*

# What does a JIT do?

- Create IR
- Loading Libraries
- Link Modules
- Optimization & Transforms
- Codegen

# Loading Libraries

```
#include “llvm/Bitcode/ReaderWriter.h”

// ParseBitcodeFile - Read the specified bitcode file
// returning the Module.
Module *ParseBitcodeFile(MemoryBuffer *buffer, ...);
```

```
#include “llvm/System/DynamicLibrary.h”

// LoadLibraryPermanently - Load the dynamic library at
// path.  It will be unloaded when the program terminates.
bool LoadLibraryPermanently(const char *path, ...);
```

# Generating IR

```
#include “llvm/Module.h”

// No default constructor, must provide name.
Module(const std::string &ModuleID);
```

You probably also want to...

```
#include “llvm/Module.h”

/// Set the data layout
void setDataLayout(const std::string& DL);
/// Set the target triple.
void setTargetTriple(const std::string &T);
```

# In Your Module...

- Functions
  - Declarations
  - Definitions
- Globals
- Annotations

# Linking

- What?
- Why?

# Linking

```
#include “llvm/Linker.h”

/// LinkModules - The Src module is linked into the Dst module
/// such that types, global variables, functions, etc. are
/// matched and resolved.
static bool LinkModules(Module* Dst, Module* Src, ...);
```

*Destroys Src...but not its memory!*

# Opts & Transforms

- No one correct level of optimization
- Create your own PassManager(s)

# Opts & Transforms

```
#include “llvm/Transforms/Scalar.h”
#include “llvm/Transforms/IPO.h”

std::vector<const char *> exportList;

PassManager Passes;
Passes.add(new TargetData(M));
Passes.add(createInternalizePass(exportList));
Passes.add(createScalarReplAggregatesPass());
Passes.add(createInstructionCombiningPass());
Passes.add(createGlobalOptimizerPass());
Passes.add(createFunctionInliningPass());
```

*All these passes and more are yours for one low price!*

# Inlining

```
#include “llvm/Transforms/Utils/Cloning.h”

/// InlineFunction - Performs one level of inlining.
bool InlineFunction(CallInst *C)
```

```
Constant *Fn = M->getFunction(“inlineMe”);

for (Value::use_iterator ui = Fn->use_begin(),
                         ue = Fn->use_end(); ui != ue; ) {
  if (CallInst *CI = dyn_cast<CallInst>(*ui++))
    InlineFunction(CI);
}
```

# Code Generation

- Get a function pointer from JIT
- Release when finished.

```
#include “llvm/ExecutionEngine/ExecutionEngine.h”

/// getPointerToFunction - This returns the address of the
/// specified function,compiling it if necessary.
void *getPointerToFunction(Function *F);

/// freeMachineCodeForFunction - deallocate memory used to
/// code-generate this Function.
void freeMachineCodeForFunction(Function *F);
```

# Practical Example

## JIT 102

*A Clang-based JIT*

# Assembling the JIT


```
/// c_to_module - entry point into libclang, our shared library
/// that uses clang to turn a C string into an LLVM IR Module.
extern "C" Module *c_to_module(const char *source, char **log);

int main(int argc, char **argv) {
  ...
  const char *source = getFile(path);
  ...
  for(;;)
  {
    // Create a new module from the source string
    Module *M = c_to_module(source, 0);
```

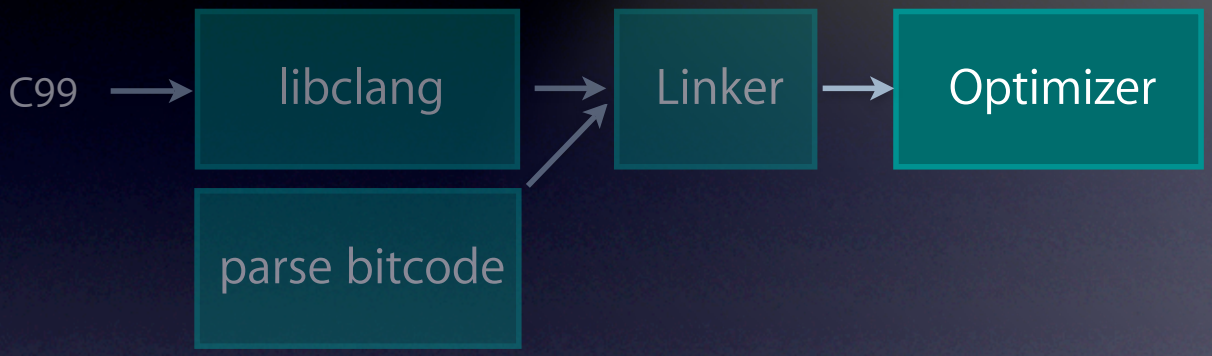
# Assembling the JIT


```
#include “llvm/Support/MemoryBuffer.h”

    // Load my library of fancy extensions to Libc
    MemoryBuffer *buffer = MemoryBuffer::getFile("mylibc.bc");
    Module *Library = ParseBitcodeFile(buffer);
```

# Assembling the JIT

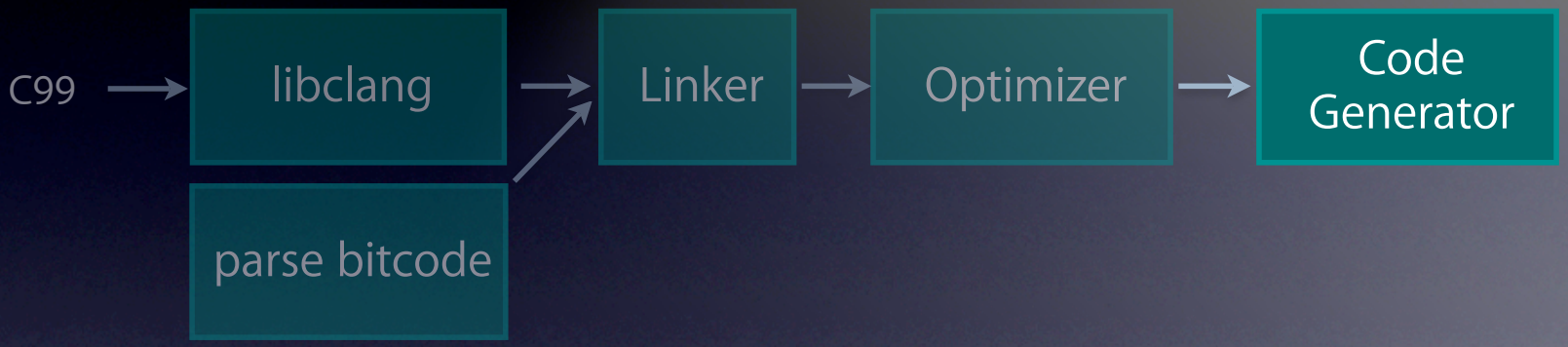

```
// Link the modules together so that we can do inlining.
// After this step, 'M' will contain all the bitcode.
Linker::LinkModules(M, Library, 0 /* error string */);

// Don't forget to delete Library, otherwise we'll leak
// memory.
delete Library;
```

# Assembling the JIT



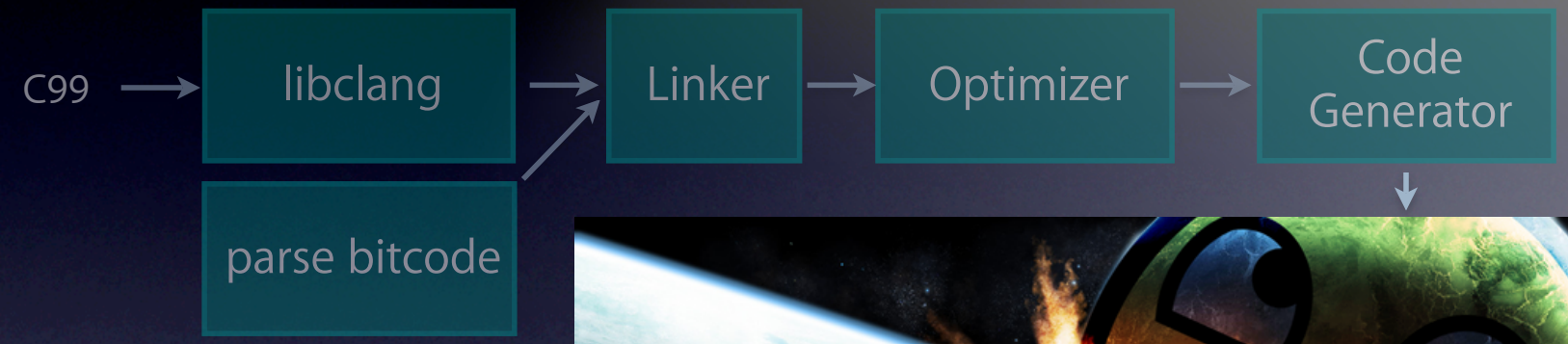

```
// Register some passes with a PassManager, and run them.
PassManager Passes;
Passes.add(new TargetData(M));
Passes.add(createInternalizePass(exportList));
Passes.add(createGlobalDCEPass());
Passes.add(createGlobalOptimizerPass());
Passes.add(createScalarReplAggregatesPass());
Passes.add(createInstructionCombiningPass());
Passes.run(*M);
```

# Assembling the JIT




```
// Create the JIT
ExistingModuleProvider *EMP;
EMP = new ExistingModuleProvider(M);
ExecutionEngine *JIT = ExecutionEngine::create(EMP);
Function *F = M->getFunction("myCosine");

// Cast function pointer to correct type and call.
jitfn fn = (jitfn)JIT->getPointerToFunction(F);
printf("foo! %d\n", fn(5));
```

# Success!


C99
libclang
parse bitcode
Linker
Optimizer
Code Generator


...but at what price?

# Demo

# JIT 102 Results

| | CPU | TOTAL | RPRVT |
|---|---|---|---|
| jit102 | 600ms | 800ms | 31500K |

Slow compile times!

# JIT 102 Results

| | CPU | TOTAL | RPRVT |
|---|---|---|---|
| jit102 | 600ms | 800ms | 31500K |

Too much memory!

# Efficient JITing

## JIT 201

*Don't do what we just taught you in 101!*

# What does a JIT do?

- Loading Libraries
- Create IR
- Link Modules
- Optimization & Transforms
- Codegen

# Areas To Optimize

- Loading Libraries
- Create IR
- Link Modules
- Optimization & Transforms
- Codegen

# Loading Libraries

```
#include “llvm/Bitcode/ReaderWriter.h”

/// getBitcodeModuleProvider - Read the header of the specified
/// bitcode buffer and prepare for lazy deserialization of
/// function bodies.
ModuleProvider *getBitcodeModuleProvider(MemoryBuffer *Buffer);
```

```
#include “llvm/Bitcode/ReaderWriter.h”

/// materializeFunction - make sure the given function is fully
/// read.
bool materializeFunction(Function *F);
```

# Loading Libraries

- What if I don't need to inline?
- Compile your runtime library to a dynamic library!

# Loading Libraries

```
// Load my library of fancy extensions to Libc
MemoryBuffer *buffer = MemoryBuffer::getFile("mylibc.bc");
Module *Library = ParseBitcodeFile(buffer);
```

```
// Create the runtime library module provider, which will
// lazily stream functions out of the module.
MemoryBuffer *buffer = MemoryBuffer::getFile("mylibc.bc");
ModuleProvider *LMP = getBitcodeModuleProvider(buffer);
Module *LM = LMP->getModule();
```

# Linking

```
Module *Library = Provider->theModule();          // 300KB + 20MB
Module *MyFunc = compileWithClang(someSource);  // 50KB

// A common mistake, links Library into MyFunc.
// Materializes & copies 20MB of IR into 50KB module => 40MB!!
LinkModules(MyFunc, Library);

// Much better, link MyFunc into Library, and delete contents
// later.  50KB into 300KB => 350KB.  100x improvement.
LinkModules(Library, MyFunc);
```

# Linking

- What is “GhostLinkage” ?
- Upcoming Improvements.

# Opts & Transforms

- Run Internalize, DCE, and instcombine on clang-generated code.
- Just run IPO opts after link.

# Demo & Bakeoff!

# Final Results

| | CPU | TOTAL | RPRVT |
|---|---|---|---|
| jit201 | 30ms | 40ms | 3900K |

20x faster compile times!

# Final Results

| | CPU | TOTAL | RPRVT |
|---|---|---|---|
| jit201 | 30ms | 40ms | 3900K |

7x less memory!

# Documentation

- llvm / llvm.org
  *llvmdev@cs.uiuc.edu*
- clang / clang.llvm.org
  *cfe-dev@cs.uiuc.edu*
- web archives of lists available

# Questions
# &
# Answers