

## Just-In-Time compilation for C++ codes

Serge Guelton

Juan Manuel Martinez Caamaño (me)

from Quarkslab

# Introduction

- ✓ Library for runtime code generation
- ✗ An omniscient virtual machine
- ✗ Read-Eval-Print Loop
- ✗ Building blocks for a Just-in-Time compiler

# Introduction

- ✓ Library for runtime code generation
- ✓ Type safe C++ Wrapper around the LLVM
- ✓ Easy to understand, predictable abstractions
- ✓ The compiler remains unmodified
- ✓ It's a hobby, so it has to be fun

# Why?

Have you ever used a Just-In-Time compiler in C++?

# Example

# Example

```
for(unsigned i = 0; i != rows-mask_size; ++i) { // scan pixel by pixel
    for(unsigned j = 0; j != cols-mask_size; ++j) {
        for(unsigned ch = 0; ch != channels; ++ch) {

            long out_val = 0; // scan the neighbour pixels
            for(unsigned ii = 0; ii != mask_size; ++ii)
                for(unsigned jj = 0; jj != mask_size; ++jj)
                    out_val += mask[ii*mask_size+jj] * in[((i+ii)*cols+j+jj)*channels+ch];

            out[(i*cols+j+((cols+1)*(mask_size/2+1)))*channels+ch] =
                out_val / mask_area;
        }
    }
}
```

# Example

```
for(unsigned i = 0; i != rows-mask_size; ++i) { // scan pixel by pixel
    for(unsigned j = 0; j != cols-mask_size; ++j) {
        for(unsigned ch = 0; ch != channels; ++ch) {

            long out_val = 0; // scan the neighbour pixels
            for(unsigned ii = 0; ii != mask_size; ++ii)
                for(unsigned jj = 0; jj != mask_size; ++jj)
                    out_val += mask[ii*mask_size+jj] * in[((i+ii)*cols+j+jj)*channels+ch];

            out[(i*cols+j+((cols+1)*(mask_size/2+1)))*channels+ch] =
                out_val / mask_area;
        }
    }
}
```

# Example

```
for(unsigned i = 0; i != rows-mask_size; ++i) { // scan pixel by pixel
    for(unsigned j = 0; j != cols-mask_size; ++j) {
        for(unsigned ch = 0; ch != channels; ++ch) {

            long out_val = 0; // scan the neighbour pixels
            for(unsigned ii = 0; ii != mask_size; ++ii)
                for(unsigned jj = 0; jj != mask_size; ++jj)
                    out_val += mask[ii*mask_size+jj] * in[((i+ii)*cols+j+jj)*channels+ch];

            out[(i*cols+j+((cols+1)*(mask_size/2+1)))*channels+ch] =
                out_val / mask_area;
        }
    }
}
```



# Example

```
void apply_filter(const char *mask,
                  unsigned mask_size, unsigned mask_area,
                  cv::Mat &image, cv::Mat *&out) {

    kernel(mask, mask_size, mask_area,
           image.ptr(0,0), out->ptr(0,0),
           image.rows, image.cols, image.channels());
}
```

# Example

```
#include <functional>

void apply_filter(const char *mask,
                 unsigned mask_size, unsigned mask_area,
                 cv::Mat &image, cv::Mat *&out) {
    using namespace std::placeholders;

    auto callme = std::bind(kernel, mask, mask_size, mask_area,
                           _1, _2, image.rows, image.cols, image.channels());

    callme(image.ptr(0,0), out->ptr(0,0))
}
```

# Example

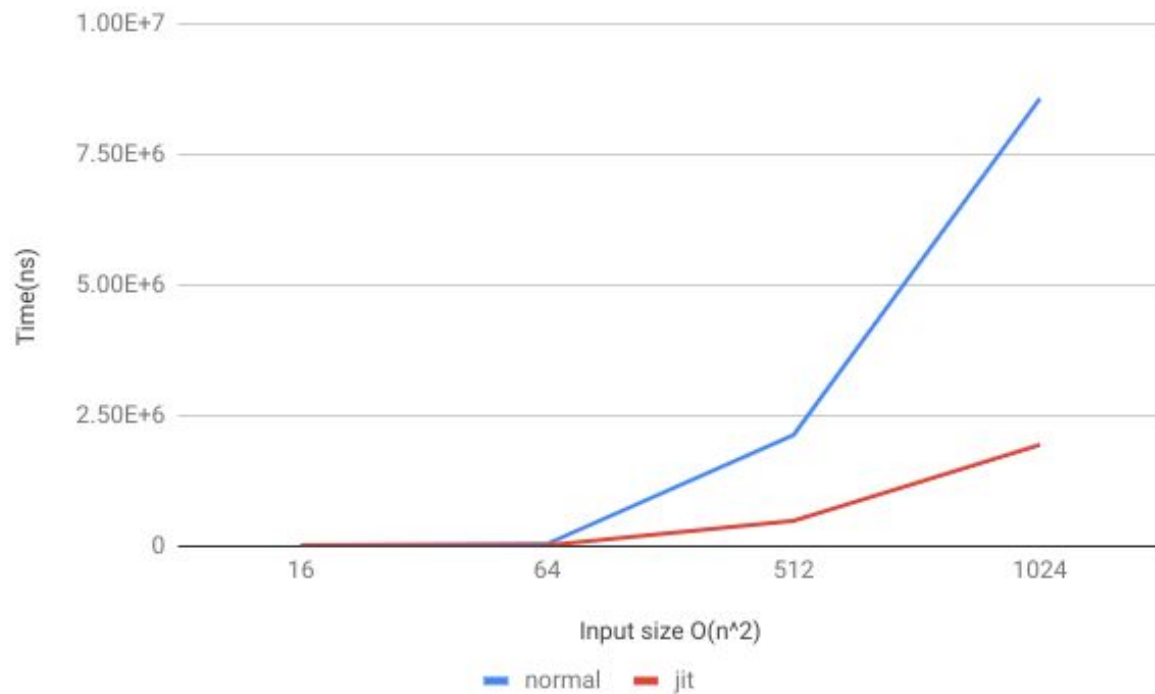
```
#include <easy/jit.h>

void apply_filter(const char *mask,
                  unsigned mask_size, unsigned mask_area,
                  cv::Mat &image, cv::Mat *&out) {
    using namespace std::placeholder;

    auto callme = easy::jit(kernel, mask, mask_size, mask_area,
                            _1, _2, image.rows, image.cols, image.channels());

    callme(image.ptr(0,0), out->ptr(0,0))
}
```

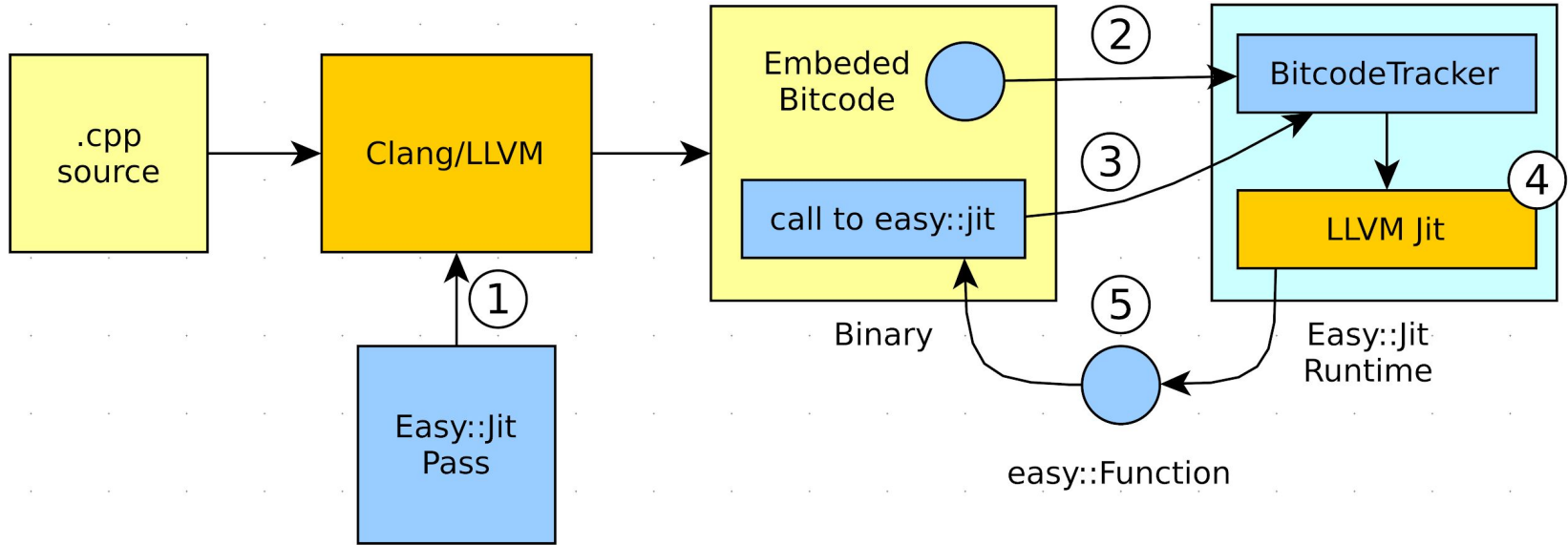
# Example





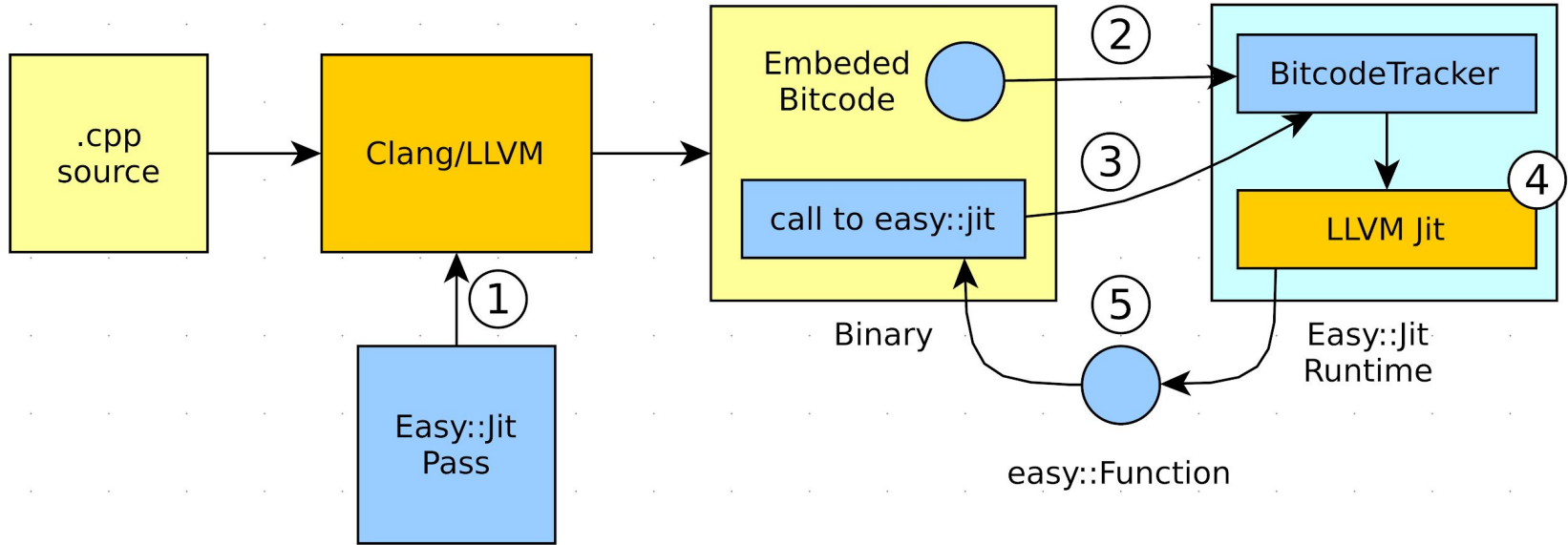
# Overview

# Overview



1. Parse calls to **easy::jit** to discover which functions are used, and embed their bitcode.

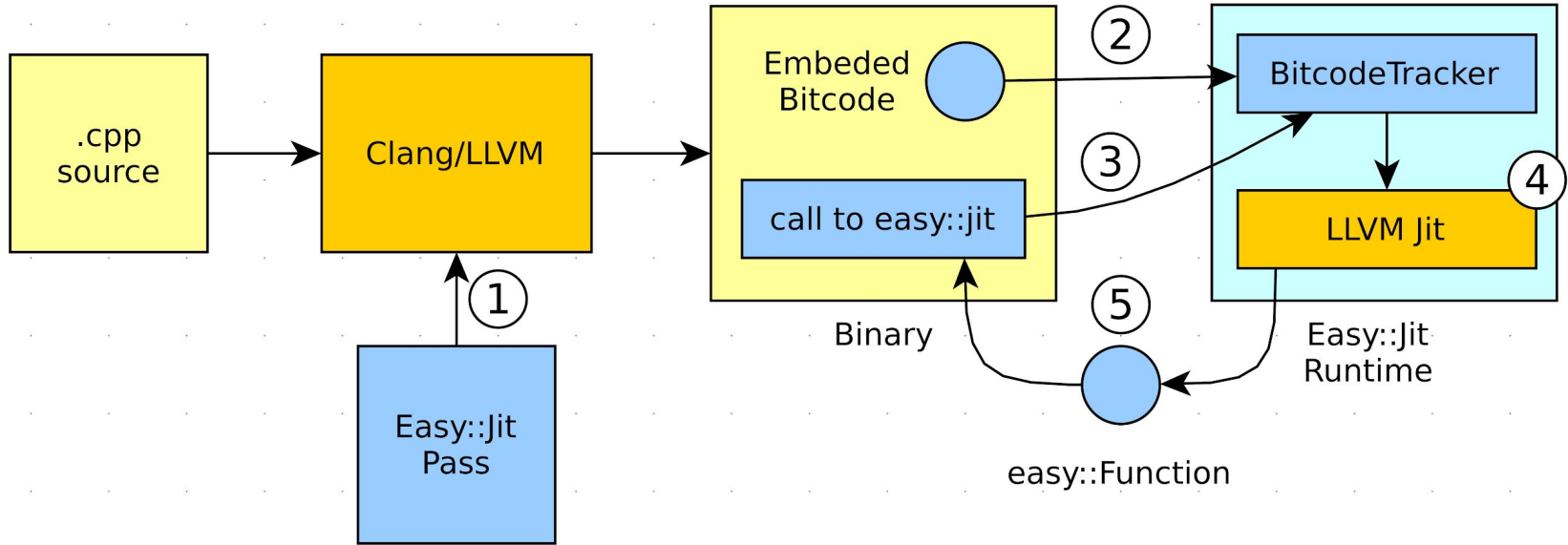
# Overview



2. At startup, register each function pointer, and its associated bitcode to the library's runtime.

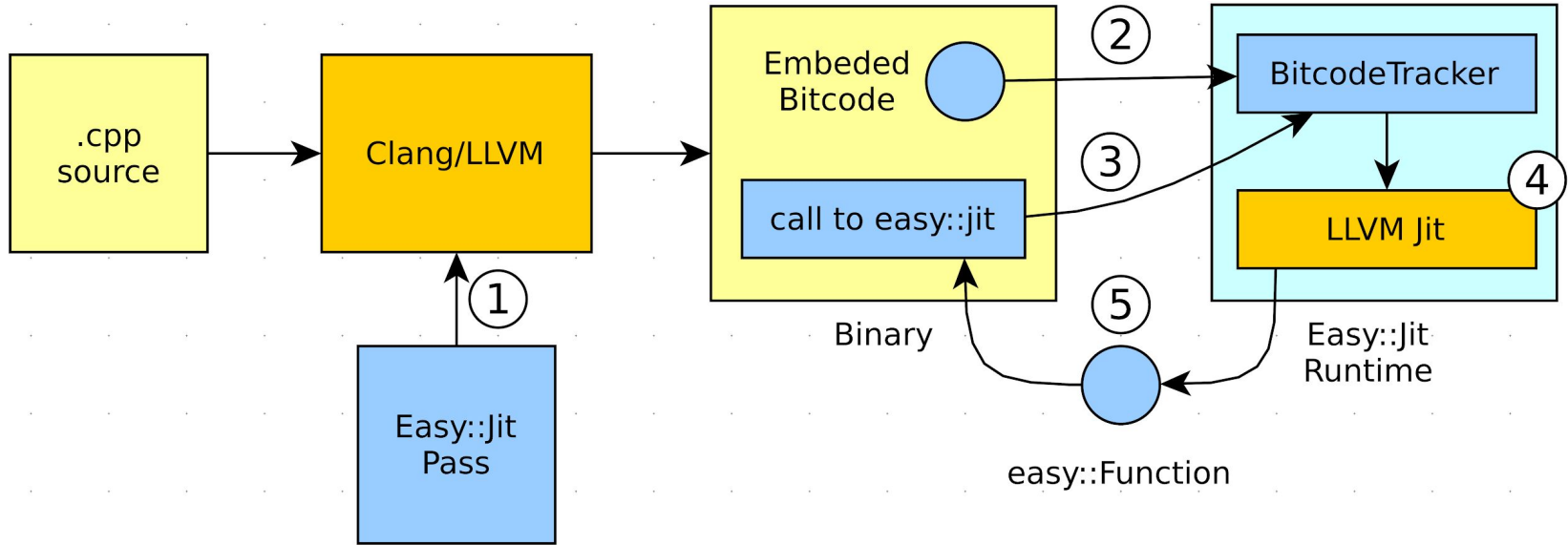


# Overview



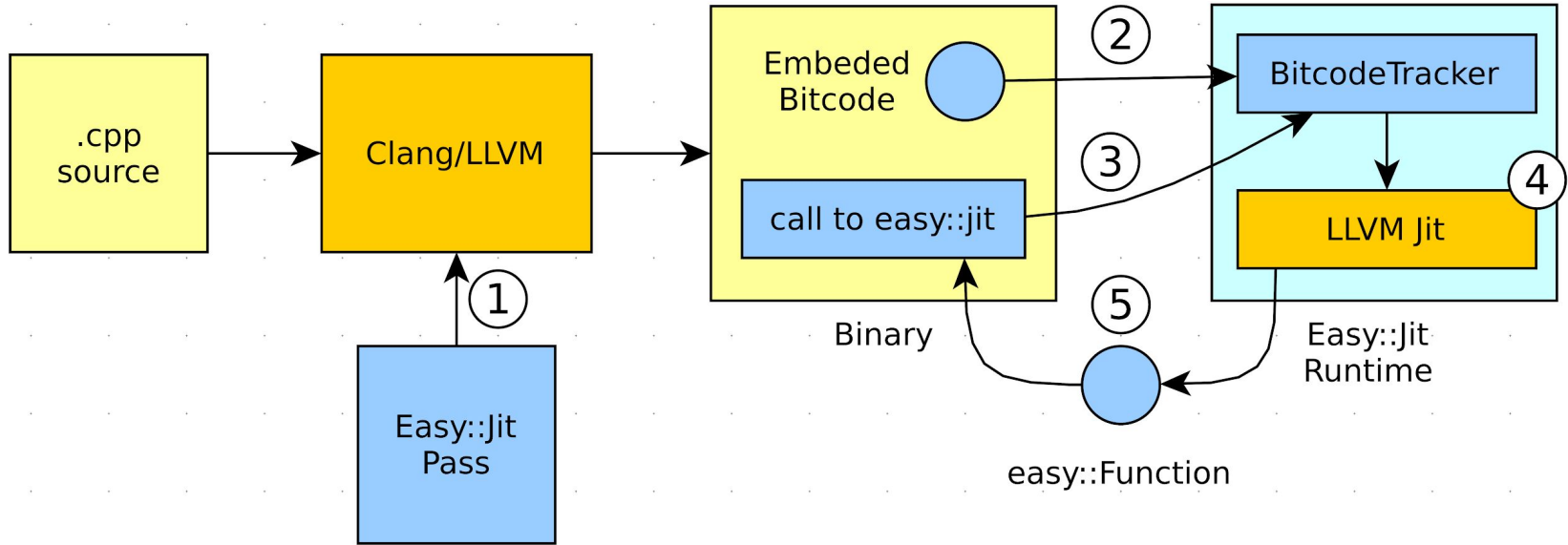
3. a. Using the function pointer, recover its bitcode. Replace the parameters by the known values. Apply optimizations.

# Overview



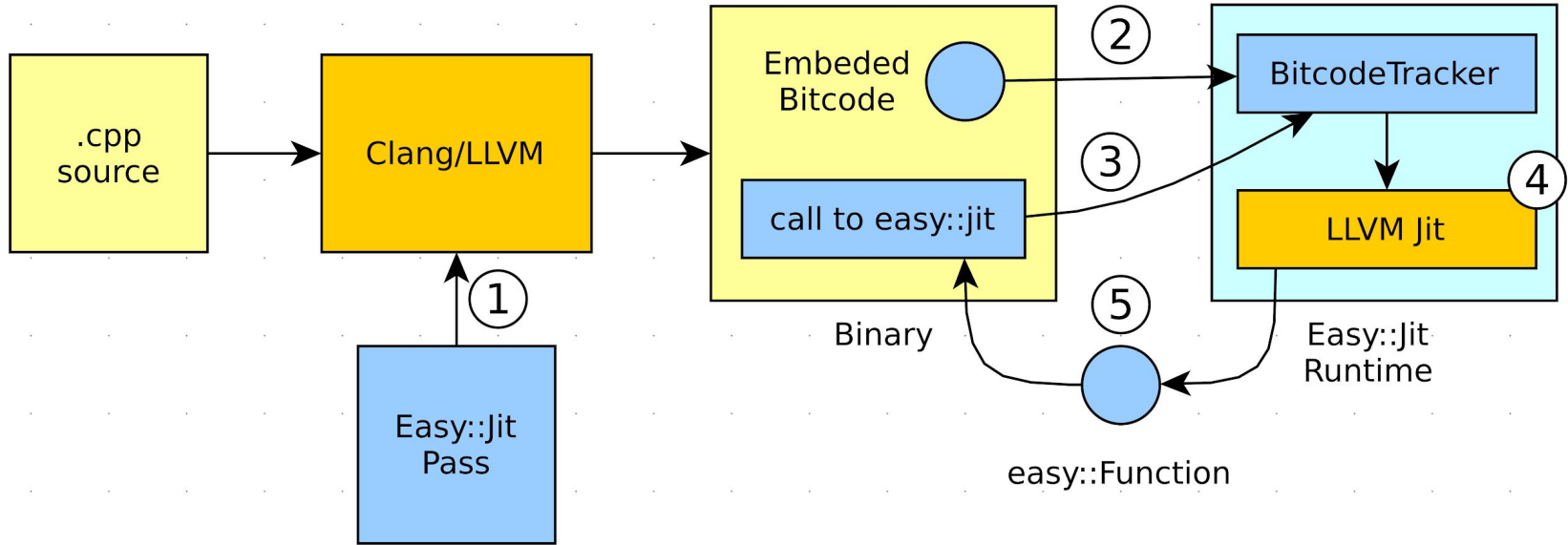
3. b. Special optimization to inline known VTable loads and inline virtual member functions.

# Overview



4. Generate binary code using the LLVM. Link the static global variables.

# Overview



5. Wrap the generated function pointer and the newly created bitcode in an opaque object to return to the user.

# The C++ library

# What happens when `easy::jit`

```
auto callme = easy::jit(kernel, mask, mask_size, mask_area,  
                        _1, _2, image.rows, image.cols, image.channels());
```

# What happens when `easy::jit`

```
template<class T, class ... Args>
auto EASY_JIT_COMPILER_INTERFACE jit(T &&Fun, Args&& ... args) {
    easy::Context C = get_context_for<T, Args...>(std::forward<Args>(args)...);
    return jit_with_context<T, Args...>(C, std::forward<T>(Fun));
}
```

# What happens when `easy::jit`

```
template<class T, class ... Args>
auto EASY_JIT_COMPILER_INTERFACE jit(T &&Fun, Args&& ... args) {
    easy::Context C = get_context_for<T, Args...>(std::forward<Args>(args)...);
    return jit_with_context<T, Args...>(C, std::forward<T>(Fun));
}
```



# What happens when `easy::jit`

```
template<class T, class ... Args>
auto EASY_JIT_COMPILER_INTERFACE jit(T &&Fun, Args&& ... args) {
    easy::Context C = get_context_for<T, Args...>(std::forward<Args>(args)...);
    return jit_with_context<T, Args...>(C, std::forward<T>(Fun));
}

template<class Param, class Arg>
void set_param(Context &C, enable_if_t<is_integral<Param>::value, Arg> &&arg) {
    Param arg_as_param = arg;
    C.setParameterInt(arg_as_param);
}
```

# What happens when `easy::jit`

```
template<class T, class ... Args>
auto EASY_JIT_COMPILER_INTERFACE jit(T &&Fun, Args&& ... args) {
    easy::Context C = get_context_for<T, Args...>(std::forward<Args>(args)...);
    return jit_with_context<T, Args...>(C, std::forward<T>(Fun));
}

template<class _, class Arg>
void set_param(Context &C,
               enable_if_t<is_placeholder<decay_t<Arg>>::value, Arg>) {
    C.setParameterIndex(is_placeholder<decay_t<Arg>::type>::value-1);
}
```

# What happens when `easy::jit`

```
template<class T, class ... Args>
auto EASY_JIT_COMPILER_INTERFACE jit(T &&Fun, Args&& ... args) {
    easy::Context C = get_context_for<T, Args...>(std::forward<Args>(args)...);
    return jit_with_context<T, Args...>(C, std::forward<T>(Fun));
}

template<class Param, class Arg>
void set_param(Context &C,
               enable_if_t<easy::is_function_wrapper<Arg>::value, Arg> &&arg) {
    static_assert(function_wrapper_specialization_is_possible<Param, Arg>::value,
                  "easy::jit composition is not possible. Incompatible types.");
    C.setParameterModule(arg.getFunction());
}
```

# What happens when `easy::jit`

```
template<class T, class ... Args>
auto EASY_JIT_COMPILER_INTERFACE jit(T &&Fun, Args&& ... args) {
    easy::Context C = get_context_for<T, Args...>(std::forward<Args>(args)...);
    return jit_with_context<T, Args...>(C, std::forward<T>(Fun));
}

template<class Param, class Arg>
void set_param(Context &C, enable_if_t<is_class<Param>::value, Arg> &&arg) {
    Param arg_as_param = arg;
    C.setParameterStruct((char*)&Param, sizeof(Param));
}
```

# What happens when `easy::jit`

```
template<class T, class ... Args>
auto EASY_JIT_COMPILER_INTERFACE jit(T &&Fun, Args&& ... args) {
    easy::Context C = get_context_for<T, Args...>(std::forward<Args>(args)...);
    return jit_with_context<T, Args...>(C, std::forward<T>(Fun));
}

template<class Param, class Arg>
void set_param(Context &C, enable_if_t<is_class<Param>::value, Arg> &&arg) {
    Param arg_as_param = arg;
    C.setParameterStruct((char*)&Param, sizeof(Param));
}
```

# How parameters are passed

How are complex data structures passed by value?

```
template<class T> struct Pair {  
    T first;  
    T second;  
};  
template<class T> void foo(T,T);
```

# How parameters are passed

How are complex data structures passed by value?

```
foo(Pair<int>(1,1), Pair<int>(2,2));
```

```
call void @_Z3fooI4PairIiEEvT_(i64 8589934593, i64 17179869186)
```

# How parameters are passed

How are complex data structures passed by value?

```
foo(Pair<double>(1,1), Pair<double>(2,2));
```

```
call void @_Z3fooI4PairIdEEvT_(double 1.00e+00, double 1.00e+00,  
                                double 2.00e+00, double 2.00e+00)
```



# How parameters are passed

How are complex data structures passed by value?

```
foo(Pair<Pair<double>>(Pair<double>(1,1), Pair<double>(2,2)),  
    Pair<Pair<double>>(Pair<double>(3,3), Pair<double>(4,4)));
```

```
call void @_Z3fooI4PairIS0_IdEEEvT_(%Pair.1* byval nonnull align 8 %1,  
                                     %Pair.1* byval nonnull align 8 %2)
```

# How parameters are passed

How are complex data structures passed by value?

```
easy::jit(foo<Pair<double, double>>, _2, _1);
```

# What happens when `easy::jit`

```
template<class T, class ... Args>
auto EASY_JIT_COMPILER_INTERFACE jit(T &&Fun, Args&& ... args) {
    easy::Context C = get_context_for<T, Args...>(std::forward<Args>(args)...);
    return jit_with_context<T, Args...>(C, std::forward<T>(Fun));
}

template<class Param, class Arg>
void set_param(Context &C, enable_if_t<is_class<Param>::value, Arg> &&arg) {
    char* flat = layout::serialize_arg<Param>(arg);
    C.setParameterStruct(flat);
}
```

# What happens when `easy::jit`

This last thing is on-going work...

# Features

```
auto callme = easy::jit(kernel, mask, mask_size, mask_area,  
                        _1, _2, image.rows, image.cols, image.channels());
```

```
callme(image.ptr(0,0), out->ptr(0,0));
```

# Options

```
auto callme = easy::jit(kernel, mask, mask_size, mask_area,  
                        _1, _2, image.rows, image.cols, image.channels(),  
                        options::opt_level(3,0));  
  
callme(image.ptr(0,0), out->ptr(0,0));
```

# Code Cache

```
static easy::Cache<> cache;
```

```
auto const &callme = cache.jit(kernel, mask, mask_size, mask_area,  
                               _1, _2, image.rows, image.cols, image.channels());
```

```
callme(image.ptr(0,0), out->ptr(0,0));
```

# Threading

```
auto callme_future =  
    std::async(std::launch::async, [&]() {  
        return easy::jit(kernel, mask, mask_size, mask_area,  
            _1, _2, image.rows, image.cols, image.channels());  
    });  
  
// do some computations...  
  
auto callme = callme_future.get();  
callme(image.ptr(0,0), out->ptr(0,0));
```



# Serialization

```
auto callme = easy::jit(kernel, mask, mask_size, mask_area,  
                        _1, _2, image.rows, image.cols, image.channels());
```

```
// write bitcode to a stream
```

```
std::stringstream stream;
```

```
callme.serialize(stream);
```

```
// load bitcode from a stream
```

```
auto callme_reload =
```

```
    easy::FunctionWrapper<void(const char*, char*)>::deserialize(stream);
```

# Composition

```
int foo_a_b(int, int);  
void map_vec(std::vector<int>&, int (*)(int));  
  
// generate a function that takes one element  
auto foo_a_a = easy::jit(foo_a_b, _1, _1);  
  
// specialize a version that applies foo_a_a over each element  
auto map_foo_a_a = easy::jit(map_vec, _1, foo_a_a);
```

# Compiler Plugin

# Compiler Plugin

- Implemented as a compiler optimization

```
clang++ -Xclang -load -Xclang EasyJitPass.so ...
```

- Identify functions that are used by Easy::Jit
  - Embed the Bitcode representation
  - Implement `layout::serialize_arg<T>(T)`

# Identify functions

- This includes functions used as parameters
- Does not work across compilation units,
  - manual annotations or a regexp
- Haven't really tried `-fembed-bitcode`.

Where are we going

# Feedback

```
easy::ProfileData profile;  
auto instrumented = easy::jit(kernel, mask, mask_size, mask_area,  
    _1, _2, image.rows, image.cols, image.channels(),  
    options::profile(profile));
```

```
auto optimized = easy::jit(kernel, mask, mask_size, mask_area,  
    _1, _2, image.rows, image.cols, image.channels(),  
    options::optimize(profile));
```

# Feedback

```
auto callme = easy::jit(kernel, mask, mask_size, mask_area,  
                        _1, _2, image.rows, image.cols, image.channels(),  
                        options::auto_tune);  
  
callme(image.ptr(0,0), out->ptr(0,0)); // may profile, or execute optimized
```



# Partial Evaluation

```
void eval(AST* ast, int variables[]);
```

```
auto program = easy::jit(eval, easy::immutable(my_ast), _1);  
program(var_values)
```

# And more...

- Methods
- Function objects
- Cache with threading support
- Cache with Persistence

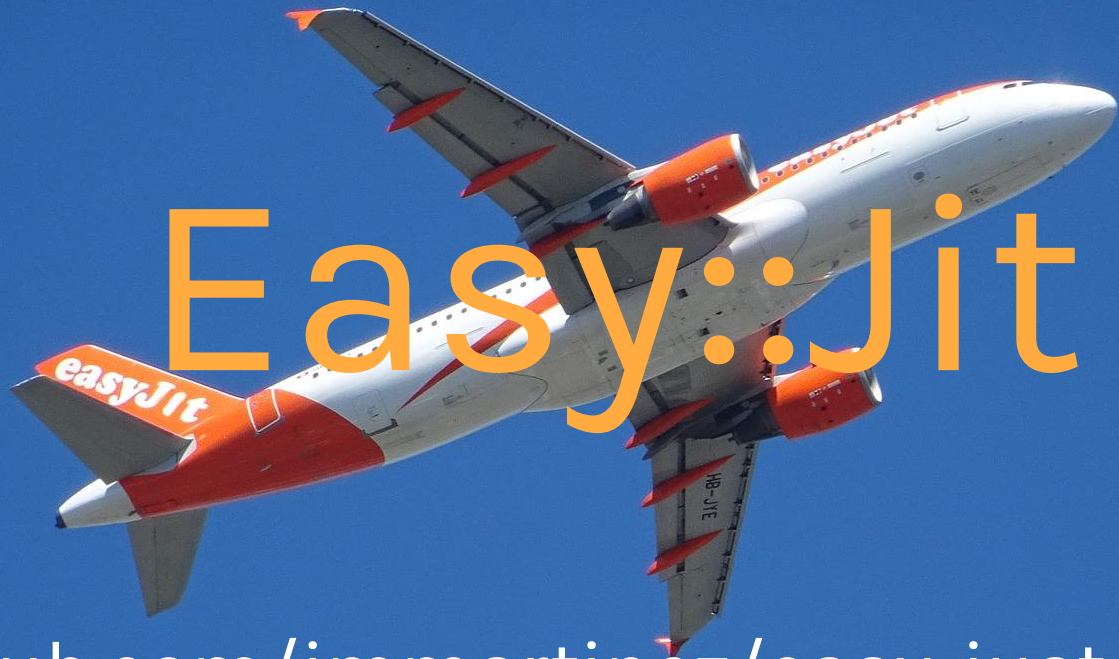
# Spinoff! Yay!

- atJIT: auto-tuner based on `easy::jit`

# Wanna join the fun?

- Do you want to get into the LLVM and don't know how?
  - Contribute!
- Do you like C++ and to tell someone that he is wrong?
  - Contribute!

[github.com/jmmartinez/easy-just-in-time](https://github.com/jmmartinez/easy-just-in-time)



[github.com/jmmartinez/easy-just-in-time](https://github.com/jmmartinez/easy-just-in-time)