Hyper uppercut!(hypercalls in LLVM)

Sumeeth Kyathanahalli

December 14, 2013

1 Introduction

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. The LLVM Core libraries provide a modern source - and target - independent optimizer, along with code generation support for many popular CPUs (as well as some less common ones). These libraries are built around a well specified code representation known as the LLVM intermediate representation ('LLVM IR'). The execution engine implements execution of LLVM IR through an interpreter, JIT dynamic compiler and MCJIT.

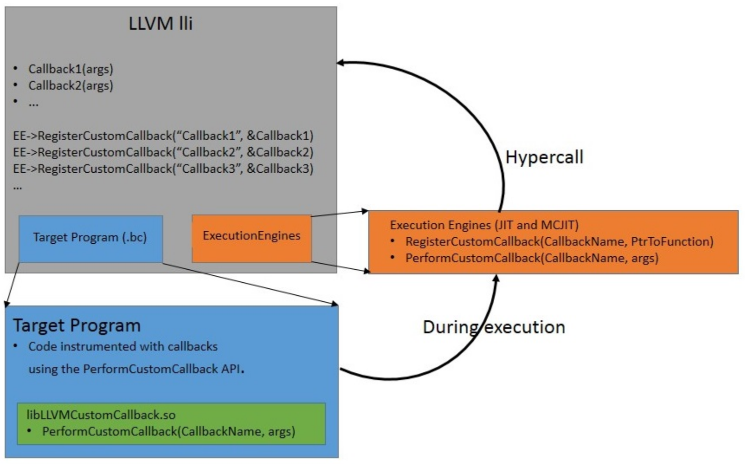
Hypercalls are a way to transfer execution from the program being executed to the LLVM infrastructure to perform re-compilation. But currently, there is no methodology to perform hypercalls into LLVM. The goal of this project is to cleanly integrate an API into the LLVM code/execution engine to create custom callbacks. The callbacks can be used to achieve:

- Hot function replacement - Client uses JIT/MCJIT to optimize frequently executed code.
- Custom memory management to transfer the code memory ownership after compilation.
- Linking of declared function names to specific addresses provided at generation time (e.g. getPointerToNamedFunction).
- Placement of generated code at specific addresses(either via allocation control, or relocation).
- More debugging support and many other requirements.

The 'lli' executable should allow having functions implementing these requirements to be registered as callbacks with the execution engines and the program being compiled by LLVM should be able to make a callbacks into those functions during its execution.

2 Design

Figure 1: Design Overview


LLVM lli
• Callback1(args)
• Callback2(args)
• ...
Hypercall
Target Program (.bc)
ExecutionEngines
Execution Engines (JIT and MCJIT)
• RegisterCustomCallback(CallbackName, PtrToFunction)
• PerformCustomCallback(CallbackName, args)
During execution
Target Program
• Code instrumented with callbacks using the PerformCustomCallback API.
libLLVMCustomCallback.so
• PerformCustomCallback(CallbackName, args)

```
EE->RegisterCustomCallback("Callback1", &Callback1)
EE->RegisterCustomCallback("Callback2", &Callback2)
EE->RegisterCustomCallback("Callback3", &Callback3)
...
```

- The user programs link with a dummy library(libLLVMCustomCallback.so) which provides an API

```
PerformCustomCallback(const char* CallbackName, void *Args)
```

- LLVM ExecutionEngines (JIT and MCJIT) will implement this function to perform the multiplexing of the various callbacks registered with them.
- Whenever the user program calls PerformCustomCallback with appropriate args, lli should experience a call to one of the registered callbacks.

3 Methodology

In JIT, the external symbols are resolved during the call to getPointerToNamedFunction. So, the hypercall resolution can be done here. During code emission when the call sites are being resolved, check whether the function being resolved is PerformCustomCallback and return the pointer to that function. This will store the address of the function PerformCustomCallback in the global table. So the next time a call happens, JIT will get the address directly from the table.

In MCJIT, during finalizeObject(), all the symbol relocations are finalized. In the Memory Managers which are responsible for the momory management of emitted code, the external symbols are resolved. So, the hypercall resolution can be done in LinkingMemoryManager::getSymbolAddress().

In the Execution engines (JIT and MCJIT), the PerformCustomCallback will do the multiplexing of the various callbacks registered based on the callback name passed as the argument. It initiates a call to the appropriate callback function by passing the arguments passed from the client. Once the callback returns with result, the same result is propagated back to the client.

4 Interface

A new API is added to the LLVM ExecutionEngine, **registerCustomCallback** which takes a callback name and two other arguments. The callback name is the name to be registered as the callback with the Execution Engine. The second argument is used to pass arguments to the callback function. The third argument is optional and can be used to pass context information from the Execution Engine to lli. The prototype of the API is as follows.

```
int registerCustomCallback(const char *CallbackName, void* (*)(),
    void *LLVM_args)
```

Whenever a new callback has to be registered, the following things are to be done:

- The new callback has to be defined in lli (a function definition).
- The callback has to be registered with the ExecutionEngine (JIT or MCJIT) using the API registerCustomCallback(). Depending upon the flags set during lli invocation, the call goes to JIT or MCJIT.
- In the user program, a call to PerformCustomCallback() with the name and arguments will execute the registered callback.

5 Apache functions recompilation

The incentive to recompile certain function in Apache[3] is to eliminate repetitive execution of code segment which doesn't do anything significant. For example, a function which always takes the same inputs, and doesn't depend on any global variable, will always give the same output. One approach is to identify the variables which can result in duplicate execution of code segments. Whenever those variables are updated, make them constants and JIT compile the code. This will help in eliminating dead code and also do constant propagation.

We have selected a httpd configuration parameter, **ap_extended_status**. Before executing the program, we find all the places in the code where this configuration parameter has been modified(store instruction) and add a call to **PerformCustomCallback** with a callback handler as the argument right after this store instruction. This will ensure that whenever the value is modified we make a function call to the callback function. This callback handler is defined in lli.

Inside the callback handler, the configuration parameter is made a constant with the value it was just assigned. And all the functions which use this parameter are then recompiled and re-linked into the module. The original copy of these functions is stored during the first execution of the program. This way, we have the unmodified copy of the function. During further recompilation cycles, we use the original function and make a clone of it. During the termination of the program, the extra function clones need to be removed from memory, because otherwise they keep references to pointers which no longer exist in memory, resulting in Segmentation Fault.

6 References

[1] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation.

[2] lli - http://llvm.org/docs/CommandGuide/lli.html

[3] Apache - http://apache.org/