Multi-Entry Functions in LLVM

Advait Bhatwadekar
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
ab8533@cs.rit.edu

Abstract—This project describes the implementation of a novel programming language feature — Multi-Entry Function — in the LLVM compiler backend. A Multi-Entry Function is a generalization of the Function feature of programming languages; unlike a Function, which restricts the program to jump into its body from only a single pre-defined entry point, a Multi-Entry Function allows the program to jump into its body through multiple pre-defined entry points.

Index Terms—LLVM; Language feature; Functions; Multi-entry; compilers

I. INTRODUCTION

In LLVM IR and most other programming languages, the identity of a Function is tightly bound to how it is called. Function’s declaration dictates how other subroutines enter the Function’s body. An implicit assumption is that this is the only way to enter the Function’s body. At the assembly level, however, the code is free to jump anywhere inside the Function’s body as long as it knows its address. Thus, having a single entry point into a Function is really just an artificial barrier set by programming languages. A Multi-Entry Function gets rid of this barrier. At the assembly level, it exposes multiple addresses inside of its body to the outer scope. At the source level, it would have more than one traditional Function declarations for its body. Each declaration would then dictate how to enter the Multi-Entry Function’s body at its corresponding entry point.

Consider the example in Fig.1. It defines a Multi-Entry Function, _add, that has two entry points: increment and plus. Increment sets the values x and y to num, its input argument, and 1 respectively. Plus also implicitly sets variables x and y from its arguments. Both entry points then delegate the addition to the common piece of code, where x and y are added and returned. Although simple, this example demonstrates some key features of Multi-Entry Functions. It shows how similar functionalities (plus and increment) can be grouped together. It also shows how Multi-Entry Functions can be used to implement default parameters. Unlike language constructs that do support default arguments, arguments in Multi-Entry Functions can have multiple default values. The above example can very easily be extended to support one more entry point: add5; where add5 adds 5 to the input argument. Multi-Entry Functions allow powerful ways to initialize variables in a function, and allow separate function calls to share code in a far more efficient manner than what is currently possible.

Multi-Entry Function is a zero-cost abstraction over the Function concept. Assembly code of a Multi-Entry Function with a single entry point gracefully degenerates to the same code that is produced by an equivalent Function. Thus, we do not expect any performance losses when using Multi-Entry Functions. But in general, having the option to enter a Function through multiple entry nodes would lead to several advantages: greater code reuse, more flexible expression of certain algorithms, smaller size of generated binaries, static optimization opportunities such as Call-Pattern Specialization [1], and reduced memory allocations in trampoline-recursive Functions.

A major issue that would need to be addressed while implementing Multi-Entry Functions is the scoping of the local variables of that function. Different entry points will correspond to different scopes, and in general there would be more than one Basic Block that is not dominated by any other
Basic Block. Another set of major hurdles are the problems that will arise due to the coupling of the calling conventions — the contract that determines how to execute a function call — and register allocation. In this paper, however, we will avoid this problem by assuming C calling convention and spilling parameters over the stack when needed.

This paper will discuss the changes to the LLVM IR to accomodate Multi-Entry Functions. We intend to implement the Multi-Entry Function feature in LLVM IR and propagate the new definition through LLVM’s code generation pipeline and generate a x86-64 assembly for it. The following sections are divided as follows: Motivation covers some additional use-cases for Multi-Entry Functions over regular Functions. Background covers the necessary topics required to understand the rest of the paper. Implementation section covers the details and challenges faced in implementing Multi-Entry Functions for LLVM IR and LLVM’s code generation pipeline. Finally, we discuss the Results we acheived for a small test example of a Multi-Entry Function.

II. MOTIVATION

The fundamental benefit of Multi-Entry Functions is code-sharing. In the example in Fig. 2, the common piece of code more than one hundred lines long. To effectively share this code without Multi-Entry Functions, one would have to encapsulate the common-logic into another function, and then call that function in each one of the three entry points: tiger, lion and fish. However, this approach is not equivalent to the Multi-Entry Function solution. In this approach, a call to tiger, lion or fish will also incur the cost of making a call to the common-logic function that registers the mascot with the school. The time to set up and tear down the common-logic function can be saved by using a Multi-Entry Function instead.

Another approach to model this example through Functions is to create a massive Function that takes one argument: what type of animal we want to make the mascot, and then generate a switch-case ladder to select the appropriate Animal constructor. This is still non-ideal in cases where we already know what animal type we are passing in to our Function. Such a static call — in which at compile time, the compiler can deduce the arguments to the Function call — would incur unnecessary runtime costs.

III. BACKGROUND

A. LLVM

LLVM [2] is an open-source project composed of tools for end-to-end compilation of arbitrary programming languages. Frontend compilers that use LLVM exist for several languages: C, C++, C#, Swift, Rust, Haskell, Python. These frontend compilers compile their respective languages to LLVM’s bytecode: LLVM Intermediate Representation (LLVM IR). LLVM IR is the core of the LLVM project. LLVM defines several optimization passes over LLVM IR. This is the first of LLVM’s two defining features: it is capable of accepting arbitrary source codes as input, and apply language agnostic optimizations to it, making it a highly attractive backend for languages that do not have a mature compiler.

The second defining feature of LLVM is its target independent code generation pipeline. LLVM can generate binaries for any target that can implement its Target class interface. Currently, it supports x86, MIPS, XCore, ARM, RISC-V out of the box. LLVM’s code generator is also flexible enough that it can be extended to generate code for arbitrary target, including other programming languages such as C.

Together, these two features allow LLVM to define a complete optimized compilation pipeline that can support many different programming languages on many different CPU architectures. Thus, a new programming language designer only needs to define a frontend to LLVM IR to get the language supported on many different architectures, while reaping the benefits of a mature optimized compiler. On the flip side, a new instruction-set designer only needs to implement methods for LLVM’s Target interface and get multiple high level languages supported on it automatically.

B. Function

This project concerns with replacing the Function feature with its more generalized form: Multi-Entry Function. But before we get into that, its important to understand how an executable code is generated for a Function. In the context of LLVM, this process happens in two phases: source-language Function gets translated to LLVM IR Function, and LLVM IR Function gets translated to a binary file Function.
LLVM IR Function is comprised of Basic Blocks. A Basic Block is a contiguous sequence of non-branching LLVM IR instructions. The last instruction of a Basic Block is an exception to this rule: it must either be some type of a jump instruction — conditional or unconditional jump — to another Basic Block in the Function or a Return instruction. Thus, Basic Blocks of a Function define its control flow graph. From an implementation perspective, an LLVM IR Function maintains an intrusive list of its Basic Blocks. By convention, the head of this list is the entry block of the Function. When the Function is called, execution begins from this block. Naturally, the rest of the LLVM code-base follows this convention, and assumes through various phases of the code generation pipeline that the control flow of a Function starts from the first Basic Block in the Function’s Basic-Block list. Since a Multi-Entry Function would allow for more than one entry block, we will need to keep this in mind when we proceed through the code generation passes.

A binary representation of a Function for any architecture is composed of three parts: a prologue, a body and an epilogue. The body is essentially the generated assembly for each Basic Block in the Function along with a private label. The prologue exposes the entry Basic Block to the outer scope, executes some additional instructions in the Function’s entry Basic Block to initialize the Function’s stack frame, and passes in the Function Parameters to registers if it has not already been done by the caller. It also bears the responsibility to save the call-preserved registers as defined by the Calling Convention of the Function. Call-preserved registers are registers that the caller of the Function expects to be unchanged after a Function Call returns. The epilogue plays the opposite role. It bears the responsibility for restoring the state of call-preserved registers to their original state, and invalidating the Function’s stack frame.

C. Calling Convention

Calling Convention is the contract between the caller and the callee of a Function on how to execute the Function call. It answers several questions regarding the execution of a Function call:

- How should the caller pass the arguments to the function?
- How should the callee return its Return value?
- How is the task of dividing the responsibility of creating and destroying the stack divided between the caller and the callee?
- Which registers are preserved through a Function call, and which ones are clobbered?

D. SSA, φ-functions and Dominance Frontiers

All LLVM IR instructions are represented in Static Single Assignment (SSA) form [3]. This implies that any variables in a valid program of LLVM IR will only be defined once, but can be used an arbitrary number of times. In other words, variables in LLVM IR are single-def. There are no variable reassignments. SSA form simplifies and improves the results of several compiler optimizations such as redundancy analysis [3], liveliness analysis [4] and register coalescing [5]. An example of transformation of regular code into its SSA form is shown in Fig. 4. Notice, how it is very easy to spot the redundant statement \( y = 20 \) in SSA form because variable \( y_1 \) never got used in the program.

Because of the nature of SSA, it might be hard to determine what should renaming of a variable look like if it is part of a diamond-form control flow graph. Consider the example in Fig. 5.

What should the assignment of \( x_5 \) look like? \( x_5 \) could either be \( x_4 \), if the control was transferred from Block #3 or \( x_2 \) if the control was transferred from Block #2. This problem is solved by \( \phi \)-functions in Fig 6.

A \( \phi \)-function contains an associative list of BasicBlock–Variable pairs. It conditionally selects one of the variable values from its pairs depending on which BasicBlock the \( \phi \)-function at runtime. If Block #2 passed the control to Block #4, \( x_5 \) will be set to the value of \( x_2 \), else it will be set to \( x_4 \) in our example. Since \( \phi \)-functions initialize the variables for their blocks, they must be the first instructions in a well-formed BasicBlock, if required.

\( \phi \)-functions in LLVM are implemented by appending copy instructions to the predecessor blocks that copy input SSA

\begin{align*}
\text{Variable assignment} & \quad \text{Variable assignment in SSA} \\
\begin{array}{l}
x = 10 \\
y = 20 \\
y = 40 \\
x = y + x
\end{array} & \quad \begin{array}{l}
x_1 = 10 \\
y_1 = 20 \\
y_2 = 40 \\
x_2 = y_2 + x_1
\end{array}
\end{align*}

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{ssa-transformation}
\caption{Example of SSA transformation}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{ambiguous-value}
\caption{Ambiguous value of \( x_5 \)}
\end{figure}
variables to the same output register (or memory address). This register is then used by the convergent block to access its phi-function assigned variable. Wide-issue machines that might produce multiple inputs to a \( \phi \)-function simultaneously (and as such cannot use the same memory location for storing all the inputs) usually have a selection machine instruction that resolves the \( \phi \)-function at runtime.

The location of all \( \phi \)-nodes that require an input from the current BasicBlock can be found efficiently using a data structure called the Dominance Frontier. The Dominance Frontier of a node \( X \) is a set of nodes where the scope of variables derived from \( X \) becomes conditional.

Formally speaking, the Dominance Frontier of a node, \( D \), is the set of all nodes, \( N \), such that \( D \) dominates an immediate predecessor of \( N \), but \( D \) does not strictly dominate \( N \). In a directed graph with a single root node, a node \( A \) is said to dominate a node \( B \) if every path from the root node to node \( B \) must pass through node \( A \). A node \( A \) is said to strictly dominate node \( B \) if \( A \) dominates \( B \) and \( A \neq B \).

Recurrence relation of dominators of a graph \( Dom \) is defined as follows:

\[
\begin{align*}
Dom(root) &= \{ root \} \quad (1) \\
Dom(n) &= \{ n \} \cup \{ \cap_{p \in \text{predecessor}(n)} Dom(p) \} \quad (2)
\end{align*}
\]

Thus, Dominator Frontier of a node \( X \) is a set of all nodes in the sub-tree of \( X \) that are immediate successors of a node that \( X \) dominates. They would be the first set of nodes from a DFS search from \( X \) where the value of some variable \( t \) derived from \( X \) may become ambiguous, and where we would need \( \phi \)-functions with associative array size of at least 2 to resolve the value of \( t \) at runtime.

IV. IMPLEMENTATION OF MULTI-ENTRY FUNCTIONS

We decided to implement the Multi-Entry Function feature such that it would coexist with the Function feature. We decided to take this route instead of rewriting Function from scratch because we wanted to avoid implementing the support for Multi-Entry Function in each and every LLVM module, as it would hurt our rate of progress. This approach indeed allowed for a faster development cycle, and enabled us to find the minimum number of changes required to get Multi-Entry Function compiled to assembly.

To support multiple entries into a Function, it is important to distinguish the notion of a Function declaration and a Function definition. A Function declaration is an interface that dictates how other subroutines are allowed to interact with the Function. A Function definition is the code that gets executed when the Function is called. Since a Multi-Entry Function exposes multiple addresses to the outer scope, but has a common body, it would have multiple declarations but a single definition.

This distinction is made very clear while implementing Multi-Entry Functions in LLVM IR. The Multi-Entry Function in the IR is then transformed through multiple code generation passes to generate assembly for it. In this project, we only define a barebones proof-of-concept implementation of Multi-Entry Functions. Throughout the implementation, we make the following assumptions:

1) Optimization Level 0: Optimization 0 allowed us to skip register allocation based optimizations, and hence reduce the number of files needed to be changed to get Multi-Entry Functions working.

2) C Calling Convention: C Calling Convention is one of the simplest calling conventions. Registers RAX, RDX and RCX are call-clobbered registers, while all other registers are call-preserved. Return values that fit into 64 bit are returned through RAX. But a more attractive feature of the C calling convention is that all parameters to the Entry block are passed along the stack.

In general, for Multi-Entry Functions, you can form a valid program in which a BasicBlock branches into an Entry Block. If our calling convention was register-based such as the Microsoft x86-64 Calling Convention, it might be a problem because we can no longer define a \( \phi \)-function to resolve our input variables. Since one of the possible options for the SSA variable would be passed in as a parameter, the \( \phi \)-function would be completely oblivious to its existence. A naive implementation of this would not make any sense: if the variable is passed as a parameter through the MEFEntry of that block, it might be stored in RDI, RSI or some other register depending upon what other arguments were present in the call. However, if the variable is passed in from some other register after branching off from another Basic-Block, the successor of this EntryBlock can no longer determine which register to refer to to get the valid instance of the variable.

This problem can be avoided in three ways:

a) Restrict BasicBlocks from jumping into Entry
Blocks directly. This would imply splitting Entry Blocks into two Basic Blocks. The first block would simply pass along the parameters to the second block. The second block — since it is no longer an Entry Block — can now have a \(\phi\)-function to resolve its SSA variable value.

b) Enforce the register allocation of variables that branch into an Entry Block according to the Calling Convention being used. Thus, if according to the Calling Convention, variable \(x\) in the Entry Block would be passed into RDI register, then all Basic Blocks that branch into the Entry Block must also allocate their \(x\) variable in RDI register. This is a much more contrived solution, however, and would require significant changes to the Register Allocation module to compensate for this change.

c) Spilling all variables over the stack. We ended up picking this option because this option would require the least number of changes to the overall codebase, and would be the fastest to implement. This approach assumes that the Entry Blocks would agree on a specific location in the stack to spill their parameters. Moreover, it implies that all variables will be passed through the stack, which is very slow and does not use the mature existing ecosystem around the \(\phi\)-functions at all. In retrospect, this approach is not very reliable as it expects some properties of Entry Points outside of the compiler, and we should have solved this problem with the first approach.

3) Default function and parameter attributes: Function and parameter attributes communicate additional information about the function and its parameters to the compiler. These properties include `nounwind` (Function does not throw an exception), `alignstack` (to force certain stack alignment for the Function), `nofree` (Function does not deallocate memory) and many more. Since some of these properties (`unwind`, `nofree`, `alignstack`) may have different values for different Entry Points in a valid Multi-Entry Function program. Supporting a different attribute for each Entry Point might prove to be non-trivial. Also, since the function and parameter attributes do not affect the correctness of the generated binary, we could safely skip it, assuming defaults, in our proof-of-concept implementation.

A. Implementation in LLVM IR

LLVM IR consists of two major class hierarchies: Value and Type. A Value object is defined as an IR object that can be used as an operand by other Values. Value is subclassed by User, BasicBlock and Argument. User is a Value that uses other Values. Important subclasses of User are Instruction and Constant. Constant is a Value type that can use other values, but can also act as operand for other User types. Moreover, it is an object that is immutable at runtime. The body of a Multi-Entry Function should not change during runtime, and thus, it should subclass Constant. Constant is further subclassed by GlobalValue and then by GlobalObject. A GlobalObject is an IR object visible outside of the local namespace and is either a Function or a GlobalVariable, but not an alias. Conceptually, Multi-Entry Function’s declarations should subclass GlobalObject because they should also be visible outside of the local namespace. Each Value in LLVM IR has a Type. LLVM defines 17 different Types, but the most prominent ones are IntegerType, PointerType, FunctionType, ArrayType and StructType.

We added two new classes to the LLVM IR Value class hierarchy: MEFBody and MEFEntry. MEFBody represents a Multi-Entry Function’s body, and MEFEntry represents a single entry point to a Multi-Entry Function. Thus, a Multi-Entry Function is composed of one instance of MEFBody and one or more instances of MEFEntry. Type of MEFBody is LabelType, so it is easier to identify the Multi-Entry Function in assembly code. The label generated, however, is not visible to the code in the outer scope. Type of MEFEntry is FunctionType. The FunctionType is the signature of the Function: InputType \(\rightarrow\) OutputType. Since an MEFEntry processes input arguments to Multi-Entry Function, it should have its own unique signature. As such, it should have its own FunctionType. A Multi-Entry Function, thus, has more than one FunctionTypes, but a single ReturnType.

The next step was to divide Function’s public interface methods and fields into two groups: definition-specific, and declaration-specific. MEFBody would need to define its own set of definition-specific methods and fields because it pertains to the definition part of its Multi-Entry Func-
tion, and MEFEntry would need to define its own set of declaration-specific methods and fields. MEFBody adopted the BasicBlockList definition-specific field from Function, and MEFEntry adopted ArgumentList declaration-specific field from Function. MEFEntry has another important field — EntryBlock — that identifies the Basic Block to which this MEFEntry’s declaration is tied to.

Another important relationship among LLVM IR objects is the container hierarchy. Module is the top-level container for all LLVM IR objects and it conceptually represents a source file. It contains a list of functions, global variables, and names of imported modules in its corresponding source file. Additionally, it also contains information about the Target platform it is targeting such as DataLayout, TargetTriple etc. Analogous to Functions, the Module should also contain a list of all Multi-Entry Functions in the source file. Moreover, a Function contains a list of its Basic Blocks and a list of its Arguments. Then for a Multi-Entry Function, MEFBody should contain a list of Basic Blocks and MEFEntry should contain a list of Arguments.

The underlying data structure of a container list is an intrusive list. An intrusive list is a doubly linked list without explicit next and previous pointers. The next() and previous() functionality is provided by each element in the list itself. Consider BasicBlock for example. It has an intrusive list of Instruction. A valid BasicBlock, by definition, must have a branching Instruction in its list’s tail. To implement next(), all it needs to do is inspect the destination pointer of its branch Instruction. previous() pointers can be populated by doing a DFS traversal over the Function’s control flow graph.

In LLVM IR, each object that wants to be a member of an intrusive list has to specialize ilist_node template class, and then inherit its public interface. The ilist_node interface also allows the element in the list to access its container element, and subsequently the intrusive list that contains it. And naturally, this reverse relationship is used throughout the codebase to access parent container methods via child elements. For instance, in the instruction selection Function-Pass FastISel, Module’s function getDataLayout() is called via BasicBlock→getParent()→getParent()→getDataLayout() function call.

In LLVM IR, Child–Parent container reverse-relationships are fixed. This made integration of Multi-Entry Functions challenging because in our new model, either MEFBody or Function could be the parent of a BasicBlock. Similarly, either MEFEntry or Function could be the parent of Argument. And because of our goal to preserve the Function feature, we could not simply redefine the Child-Parent relationship in the case of BasicBlock and Argument. As a result, we resorted to add many duplicated MEFBody-specific methods to BasicBlock class and duplicated MEFEntry-specific methods to Argument class. This had a snowballing effect. Everytime we encountered a function that called BasicBlock→(some method, such as getParent(), pertaining to Function), we had to duplicate that function to instead call BasicBlock→(equivalent method, such as getParentMEF(), pertaining to MEFBody).

Final set of changes to integrate MEFBody and MEFEntry into LLVM IR were related to integrating these two classes into LLVM’s custom Runtime Type Information (RTTI) system. LLVM runs its own RTTI to support dynamic_cast and static_cast operators for the Value class without incurring the cost of vtables. In this system, each subclass gets a unique enum value, and runtime casts are done by switching over to the handler for the enum value of the object. To facilitate extensibility of this system, LLVM provides macro expansions such as HANDLE_VALUE in llvm/include/Value.def that automatically generates switch-case ladders and enum names at the preprocessor stage. Enum identifiers for MEFBody and MEFEntry are also generated using this macro, and are instantiated in llvm/llvm-c/Core.h.

B. Implementation in LLVM code generator

Code generation in LLVM transforms the code through six major phases: Instruction Selection, Scheduling and Formation, SSA-based machine optimizations, Register Allocation, Prologue/Epilogue insertion, Late Machine Code Optimizations and Code Emission. This implementation of Multi-Entry Functions passes through Instruction Selection, Scheduling and Formation, Prologue and Epilogue insertion and Code Emission.

In the Instruction Selection phase, LLVM IR instructions are mapped to the target machine’s native instructions. Scheduling and Formation phase imposes an order on the control flow DAG of instructions based on some criteria such as minimizing register pressure or instruction latency. Prologue/Epilogue phase inserts a prologue to each Entry Block in the Multi-Entry Function and an epilogue at the Return Blocks. Code Emission pass prints out the contents of each Basic Block in the Multi-Entry Function along with target-specific gunk on stdout.

Listing 1. “test code for Multi-Entry Function”

```
def add:
    plus(x, y):
        if (y < 1):
            inc(x):
                y = 1;
            return x+y;  # .ret_BBB
```

To identify the smallest set of changes we would need to do to implement Multi-Entry Functions in the backend, we decided to write out a very simple test Multi-Entry Function (Listing. 1) TargetMachine interface defines a function addPassesToEmitFile() that adds passes required for code generation to LLVM’s PassManager. These passes are either Module passes or Function passes. The series of passes generated by this function included 4 Module level passes:

1) Pre-ISel Intrinsic Lowering: This pass lowers all LLVM intrinsic function calls in the Module to regular LLVM instructions before Instruction Selection phase.

2) LLVM IR Function Passes: There are 11 passes in this phase, most pertaining to optimizing the IR before code generation begins. Important passes on the IR are:
a) Removing Redundant Basic Blocks: This pass did a Depth First Search (DFS) traversal on the control flow graph of the Function to identify the Basic Blocks in the Function that could not be reached through the entry node. This DFS, however, assumed that there will always be a single root node. This function was also written generically, and used in other places in the code base on graphs that were not a Functions’ control flow graph. And so it was unreasonable to extend it to support multiple roots for DFS. We added a method getPseudoBasicBlock() to MEFBody class to accommodate the single root DFS code. getPseudoBasicBlock() returns a BasicBlock, which is not a part of MEFBody’s BasicBlockList, that consists of a single switch-case instruction. This instruction has branches to all the entry blocks of the Multi-Entry Function. Thus, a DFS traversal starting from this virtual Basic Block travels to each and every Entry Block in the Multi-Entry Function.

b) Dominator Tree Construction: LLVM’s dominator tree classes also support the construction of PostDominators. PostDominators in general do not have a single root. To accommodate the same Dominator Tree construction algorithm — Semi-NCA — PostDominators in LLVM are created by first constructing a virtual node which has branches to all the root nodes. We chose not to piggyback on this construction as we were not sure if there was any PostDominator-specific code that might hinder dominator tree construction of Multi-Entry Functions. Instead, we used the same approach as in the previous step: begin with Root node as the BasicBlock returned by getPseudoBasicBlock(), and construct the Dominator Tree for the Multi-Entry Function’s control flow graph.

3) Rewrite Symbols: This pass renames Function names with respect to a regular expression rule.

4) Code Generation Function Passes: There were a total of 39 passes in this phase. They cover the code generation pipeline of LLVM discussed earlier. We skipped the passes that had to do with extracting debug information, exception-handling or garbage collection.

Critical passes in this phase are:

a) Dominator Tree Construction: Construction of dominator tree if it was not done in the previous phase.

b) X86 DAG→DAG Instruction Selection: This pass transforms the control flow DAG of LLVM IR Instructions to the control flow DAG of target-specific Instructions. For Functions, this pass lowered the parameters of the first Basic Block in its BasicBlockList. For Multi-Entry Functions, we had to loop over each of the Entry BasicBlock to lower arguments in each one of them individually. Rest of the pass was functionally unchanged from its Function counterpart.

c) Local Stack Slot Allocation: This pass assigns relative stack addresses wherever applicable. This pass did not have a Multi-Entry Function specific change.

d) Two-address Instruction Pass: This pass converts the three address form of LLVM IR instructions to their two-address forms. This pass did not have a Multi-Entry Function specific change.

e) Phi Elimination Pass: This pass gets rid of φ-functions. It appends copy instructions to the predecessor blocks of each φ-node. These copy instructions copy their convergent variables to the same output register (or memory address). This register (or memory address) is then used by the φ-node to resolve the value of its variable. This pass did not have a Multi-Entry Function specific change.

f) Fast Register Allocator: This pass allocates registers on a Basic Block level, and tries to hold all variables in registers. This pass did not have a Multi-Entry Function specific change.

g) Prologue-Epilogue Insertion and Frame Finalization: This pass inserts the prologue for the entry block of a Function and an epilogue to each of its Return blocks. For a Multi-Entry Function, epilogue insertion was straightforward. Surprisingly, extending prologue insertion to multiple entry blocks was also straightforward because the necessary infrastructure to do so was already set up in LLVM to support exception handling.

h) AsmPrinter Pass: This pass prints out the assembly code for the each BasicBlock in the Multi-Entry Function. This pass did not have a Multi-Entry Function specific change.

V. Result

The parts in green correspond correctly to the expected assembly for the test program. It looks like the assembly for the Multi-Entry Function is generated with a lot of seemingly unnecessary mov instructions. However, we do have some explanation for some of the mov instructions. The instructions

\[
\text{movq} \quad -32(\%rsp), \%rdx \\
\text{movq} \quad -40(\%rsp), \%rsi
\]

seem to be fetching the parameters from the stack (as we are using the C Calling Convention). The instruction

\[
\text{mov} \quad \%rdx, \quad -8(\%rsp)
\]

following the jge jump seems to be copying the registers to common memory location for the variable stored in rdx. This could be the result of Phi-Lowering pass.

Rochester Institute of Technology
However, we have no way of explaining the first four instructions under the `plus` label. We are also not yet sure why the assembly code is missing the code generation for the final basic block (labelled `.ret_BB`). Due to time constraints, we were not able to implement the `.globl` headers for the Entry Blocks, and as such they are missing from the generated binary.

There are a couple of reasons that might be responsible for the incorrect output.

1) Missing passes: It is possible that we missed implementing some LLVM passes that were necessary in the correct compilation of the IR code. The LLVM documentation does mention that some passes do exist that do not modify the Function or Module directly but are a prerequisite for some passes that do.

2) Using stack instead of registers for passing variables: As discussed earlier in C Calling Convention subsection, it is likely that the code written with this approach is prone to mistakes, and one of those mistakes might have resulted in a bunch of random `mov` instructions in the generated binary.

VI. STATISTICS

We changed 104 files in the LLVM source code, with 9521 additions and 434 deletions.

1) Link to my LLVM fork: https://github.com/You-NeverKnow/llvm-project/tree/mef-pseudo-block

2) Link to the test code: https://github.com/You-NeverKnow/Multi-Entry-Functions

ACKNOWLEDGMENT

The author would like to thank Prof. Matthew Fluet for introducing this project to him, and his continued guidance throughout the timeline of the project.

REFERENCES


