Generating Stack Machine Code using LLVM

Alan Li
ETC Core
alan.l@etclabs.org, alan.li@me.com

Abstract

This unfinished draft article[1] discusses about the design and implementation of a stack machine codegen in LLVM. LLVM is designed to support code generation for register machines, and this hinders the wide use of LLVM for stack machines. This article shows that it is possible to target LLVM to stack machine platforms with minimal tradeoffs. Such framework could possibly be adapted to various of other stack machine architectures.

This work is funded by ETC Core, the leading Ethereum Classic core development team.

1 Introduction

The Stack Machine Model

Recently, stack machine models are becoming a popular choice for virtual architectures. New virtual ISAs such as WebAssembly, Ethereum VM, and the Telegram VM on TON blockchain all choose stack machine ISAs. The stack machine model is considered a good choice of ISA design in situations where code size and code density is an important factor of performance, such as smart contracts on blockchain, or in-browser applications. Texts discussed the benefits and drawbacks of stack machines include: [Koopman, 1989], [Shi *et al.*, 2008], etc.

Stak Machine Compilers

However, there is no such a general stack machine compiler infrastructure specifically designed for stack machine code generation. Stack machine compiler developers have to either build their own compiler framework or re-purpose existing SSA-based compiler framework to do the job.

A majority of the traditional compiler techniques can be directly applied to build a stack machine compiler, especially in the case of a retargetable compiler infrastructure such as LLVM, where most of its modules are designed to be independent and reused.

Having a dedicated stack machine compiler for a specific stack machine could be beneficial in some aspects when doing codegen, but people generally think it is better to build the stack machine target upon already matured compiler frameworks so we can take advantage of existing compiler analysis framework and optimizations.

The author presents his approach to build a stack machine target (to be more specific, the EthereumVM architecture) in the LLVM framework [Lattner and Adve, 2004] to generate stack machine code in this article.

The EthereumVM Architecture

The EthereumVM [Wood and others, 2014] is a simplistic, Turing-complete virtual machine designed for executing smart contracts on decentralized platforms. By now, EVM is being adopted and used in projects other than the Ethereum blockchain, making it the most popular smart contract engine so far.

EVM is designed to be a 256-bit, deterministic, synchronous, safety-oriented stack virtual machine. The architecture uses no registers, including special registers such as frame pointers. To make things easier, EVM does have a memory space for storing temporary values, along with a storage space which serves as a database so developers can store persistent contract information.

One thing distinguish EVM from other execution engine is that each EVM instruction is fueled with limited supplied "gas". The gas is essentially the underlying currency of the blockchain, called *Ether*. By carefully allocate gas consumption schedule, the blockchain intricately controls computing resource allocation of each of the smart contract executions. The "out of gas" exception forces contract to halt should something out of control happens.

The major optimization goal for EVM target is to reduce the cost of executing a smart contract, along with deployable reduced smart contract size.

Rationales behind EVM Compiler Backend

Currently, EVM developers are limited to only a few languages, each has its own compiler [Dannen, 2017]. Still, people are trying to develop new smart contract languages, and the lack of a general purpose EVM backend makes it difficult to build competitive ones.

The ideas from this article originated from implementing the EVM target in LLVM[2]. The main purpose of building an EVM target on LLVM is not only to benefit the developers

[1] Last edited and compiled on Wednesday 5th February, 2020.

[2] The code is hosted at https://github.com/etclabscore/evm_llvm

by expanding the EVM ecosystem, but also to have long term compiler support of the EVM target.

2 Code Generation

2.1 Codegen Pipeline

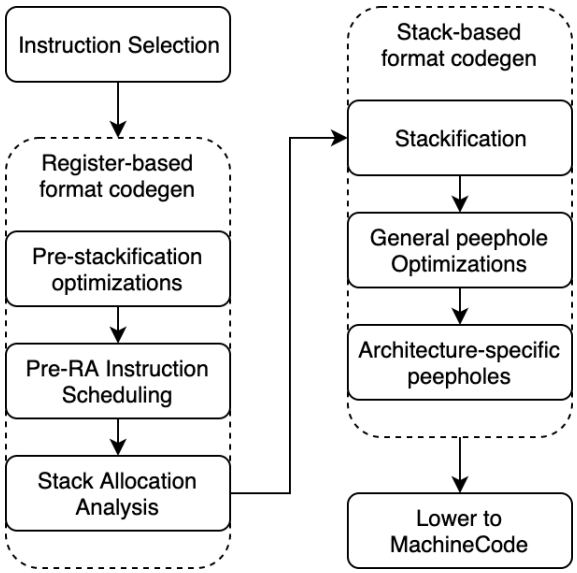

Figure 1: Illustration of pipeline

Figure.1 shows the flowchart showing the codegen workflow for a stack machine. The pipeline used to generate stack machine code is different than that of a register machine codegen pipeline. The followings are discussions on the design of the pipeline:

No register allocation. General purpose physical registers are non-existent in stack machines, there is no point to allocate physical registers for a stack machine. All the registerrelated operations will be performed on virtual register format. The physical resource allocation process is instead done in the stack allocation pass.

Two sets of machine opcodes are defined. For each stack machine instruction, we define a register-based flavor and a stackbased flavor in the backend. Figure.2 is a side-by-side comparison of the two formats, where register-based instructions are suffixed with `_r` to distinguish themselves from stack-based ones. There is also a mapping function defined to retrieve a stack instruction given by its register counterpart. The register instructions are defined in simple RISC format, where all instructions take register operands except the `PUSH_r`, which is specifically used for instantiating a register with constant value.

Such idea originated from the WebAssembly LLVM backend [Haas *et al.*, 2017]. The EVM backend reuses this technique because of its engineering simplicity.

Codegen Passes are split into two categories. The *Stackification* pass is used to convert register-based instructions into stack-based flavor. Passes after it will operate on stack instructions.

```
%0 = PUSH_r 0x40          PUSH 0x40
%1 = LOAD_r %1            LOAD
%2 = PUSH_r 0x20          PUSH 0x20
%3 = ADD_r %1, %2         ADD
%4 = PUSH_r 0x40          PUSH 0x40
STORE_r %4, %3            STORE
```

Figure 2: Instructions in register and stack representations

Stack Allocation

The stack allocation pass tries to allocate a virtual register on stack space or spill them onto memory space if the situation is not permitted. The goal of stack allocation is to reduce overall memory spillings.

[Shannon and Bailey, 2006] described a global stack allocation method which works in place of global register allocation. The method divides the stack into several spaces, including local regions and cross-BB regions. The algorithm will assign a region to each of the variables.

This pass is irrelevant to our local (intra-basicblock) instruction scheduler. A local scheduler only alters an instruction's position within a basicblock, hence can not alter a variable's position within an assigned region. Such property enables us to simplify stack machine codegen.

The Stackification Pass

The tasks of stackification pass is to map instructions to stack instructions, then inserts stack manipulation instructions so the stack operands are correctly retrieved. It respects the orders of input instructions and block layout, and only inserts stack manipulation instructions when it sees fit. Notice that to correctly arrange stack manipulation instructions, the pass needs to know stack allocation information. If an operand was assigned to a spilled memory location, the pass have to insert instructions to bring it back to stack.

To make the engineering effort manageable, the pass is designed to be agnostic of previous optimizations, such as instruction scheduling. It does not change or remove the order of original instructions.

Figure.3 shows a case where we have to insert an additional `SWAP` instruction to make sure the order of operands are correct on top of stack. In reality, the inserted `SWAP` instruction can be eliminated by instruction scheduling.

3 Global stack allocation

Ethereum VM's memory access is expensive in that to access a memory element, more than a few instructions are needed to retrieve the value from memory. In this case, putting variables on stack instead of memory is profitable.

3.1 Stack Regions

We divide the stack space into two major regions for allocation. The local stack region (*L region*) stores variables that are local to a basic block, and the transfer stack region (*X region*). We also utilize memory space for cases that cannot be

```
%0 = PUSH_r 0x00          PUSH 0x00
%1 = LOAD_r %0            LOAD
%2 = PUSH_r 0x20          PUSH 0x20
%3 = LOAD_r %2            LOAD
%4 = SUB_r %1, %2         SWAP
%5 = PUSH_r 0x40          SUB
STORE_r %5, %4            PUSH 0x40
                          STORE
```

Figure 3: Example: `SWAP` is inserted to ensure correctness of operand order on stack

allocated on stack. Figure 4 shows how the local and transfer regions are arranged in the stack space, where the stack grows upwards. Note that function's formal arguments are instantiated using a pseudo instruction at the beginning of the function, so there are no cross function boundary dependencies for variables.

The analysis pass will assign to each of the variables with one of the following four allocation decisions.

- **Allocate on *L region*.** For a variable with a live range of a single basic block, assign it to *L region*. Notice that at the beginning and at the end of a basic block, the *L region* is empty.
- **Allocate on *X region*.** Some variables with cross basic block access patterns can be assigned to the transfer stack region so as to avoid memory accesses. The *X region* carries cross block dependent information. We will discuss the details in Section 3.2.
- **Allocate on memory.** Variables with complicated access patterns will be assigned to a memory slot. This is the fallback option. Our goal is to minimize it as much as possible.
- **No allocation.** If a variable does not have any use after define, do not allocate stack slot for it. In the stackification pass, we will insert a `POP` instruction immediate after its define to pop it out of the stack.

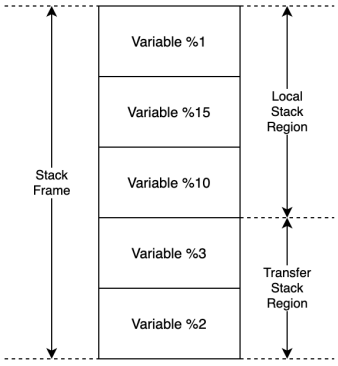

Figure 4: Illustration of stack region arrangement

3.2 SSA-based Algorithm

It is fairly easy to distinguish *L-region*-assigned variables, so we will focus on other cases. [Shannon and Bailey, 2006] recognizes that when flow control changes, the program have to make sure the shape of the stack must be in a fixed and known state to successors. When multiple predecessors or successors exist, predecessors must have exact the same stack at the time of branching.

Shannon's original dissertation describes the algorithm using edge-sets to determine the group of control flow branches that should have the same stack shape. Basic blocks in a same edge-set have to ensure its entry or exit stack statuses are identical to each other. For example, the scenario of `%X` shown in the left side of Case 1 in Figure 5 is a no for *X region* allocation. The reason is that the lower left basic block does not have uses of `%X` — resulting in imbalanced stack if the program takes the left path.

3.3 Optimizing Stack Allocation

A prepossessing pass consists of program transformations is used to improve the performance of stack allocation. The pass is designed to be fully decoupled with stack allocation, and they both are agnostic of each other. The transformations in the pass enables more *X region* allocations rather than memory allocations. The ideas behind this optimization can be described by the following bullet points:

- Splitting register live ranges. Register coalescing tries to combine variables to eliminate copy instructions. The idea works for register machines but does not work on stack machines, as stack operands are ephemeral. On a stack machine, the process of coalescing lengthens register liveness range, potentially invalidates stack allocation opportunities. The Case 2 shown in Figure 5 is a conversion to split a register range so both *%x* and *%y* can be allocated on *X region*.
- Fill in empty blocks in a same edge-set, so that all the basic blocks in a same edge-set produce or consume same stack elements. This idea is shown in the example of Case 1 in Figure 5.

Case 3 in Figure 5 is a comprehension example showing how to stack allocate *%x*.

4 Codegen Optimizations

This section describes codegen optimizations that *could be* applied to a stack machine backend. [3]

4.1 Instruction Scheduling

Instructions such as `DUP`, `SWAP`, `DROP`, that do not contribute directly to the computation but are necessary to ensure operand order are stack manipulation instructions. Instruction scheduler on stack machine is used to remove as much stack manipulation instructions as possible. Many literature,

[3] As of year end 2019, the EVM target has not yet implemented all of the optimizations. Those unimplemented optimizations will be our future works. Experimental results will be later reflected in Section 7.

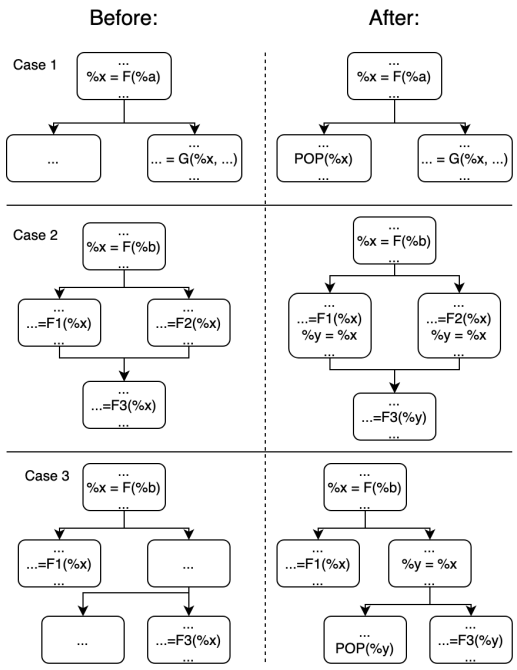

Figure 5: Examples of optimization cases

[Maierhofer and Ertl, 1998], [Koopman, 1992] suggests local instruction scheduling algorithms produce excellent results. In our compiler framework, we choose to use local scheduler. The scheduler is orthogonal to other optimizations, and works only on variables allocated on *L region*.

4.2 Peephole optimizations

Some preliminary general optimization ideas for stack machine code generation are described in [Koopman, 1992] and [Knaggs, 1993].

Architecture-specific peephole optimizations are an important part and is executed at the end of the pipeline. For example, in EVM backend the constant materialization instruction `PUSH` have 32 flavors indicating the constant length. A `PUSH32 0x01` instruction will take up 33 bytes of space, so it is profitable to convert it to `PUSH1 0x01` to improve code density. Similarly, materializing a small negative value can be implemented as a `PUSH` and along with a `SIGNEXTEND` to do sign extension.

5 The Plan for EVM Backend [4]

The EVM backend is still undergoing development. Here is a short summary of its engineering efforts:

Basic codegen completed, most optimizations unfinished. Including:

- Instruction scheduler
- Pre-stackifying optimizations
- Global stack allocation

Local stack allocation algorithm is used to support stackification pass.

Other necessary backend plugs: such as 256-bit support; simple EVM-specific object file emitting, including metadata information; EVM-specific intrinsics.

EVM LLVM project is a candidate for Ethereum Community Fund supports. An alpha version is released in January 2020.

6 A General Purpose Stack Machine Codegen Framework

It has come to our attention that the majority part of the codegen framework we implemented can be reused across different stack machine targets. Or at least, the methodologies exercised can be summarized and used as guidelines for another stack machine target. This section contains discussions on building a general purpose stack codegen framework on LLVM.

6.1 Target-generic Components

The following is a list of codegen components that can be implemented in a target-agnostic way:

A symbolic execution engine simulates the stack status must be used to determine the variable locations on the stack. Such engine must be used in the stackification pass, and can also be used in the instruction scheduler and peephole optimizations pass.

A stack allocator along with an optimization pass to analyze the program and efficiently allocate stack space for variables.

A register/stack instruction defining mechanism allows users to quickly define both register-based instructions and stack-based instructions, along with a mapping mechanism between both categories.

An instruction scheduler can also be implemented to be agnostic of target information. One simple approach is to implement a simple depth-first scheduler.

A general-purpose stack code optimizer which implements optimization ideas presented in various of texts such as [Koopman, 1992] and [Maierhofer and Ertl, 1998] can be useful.

6.2 Target-specific Components

Target-specific implementation inside the general framework are done through virtual function hooks. Operations such as *load from memory* or *store to memory* must be implemented by target backend because it involves machine instructions.

Other than that, we collected a list of target-specific information the backend needs to fill in:

Stack operations that tells the codegen what tools can be used in generating optimal code for a specific scene. For example, some target have restrictions on stack operations: in EVM the computer can retrieve a variable on stack up to stack depth of 16. Should a variable reside deeper in the stack, compiler have to spill some variables to make the

[4] This section is completely sidenote and is not related to technical discussions. The materials presented in this section are for the purpose of completeness and information.

desired stack variable depth less or equal than 16. On the other hand, WebAssembly provides convenient approaches to retrieve variables deep in the stack such as `put_local` and `get_local`.

Stack frame definitions that explains the structure of a stack frame so the framework can generate stack loads and stores properly. Target's calling conventions also partly attribute to the stack frame definitions.

7 Performance and Evaluations

We are still undergoing corresponding evaluations and will update this section as the development goes on.

8 Future Works

EVM is the most widely accepted smart contract engine on blockchains. Its performance matters very much, because every instruction is associated with a monetary cost. So it is very tempting for developers to reduce the cost of smart contract execution.

By utilizing LLVM infrastructure, the EVM LLVM backend gives a platform for various of opportunities of stack machine code optimizations. On the other hand, such compiler optimizing and analysis platform can help improving EVM ISA design in a next version iteration. The "softwarevirtual ISAruntime" codesign approach could further improve smart contract performance.

References

[Dannen, 2017] Chris Dannen. *Introducing Ethereum and Solidity*. Springer, 2017.

[Haas *et al.*, 2017] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *ACM SIGPLAN Notices*, volume 52, pages 185–200. ACM, 2017.

[Knaggs, 1993] Peter J Knaggs. *Practical and Theoretical Aspects of Forth Software Development*. PhD thesis, University of Teesside, 1993.

[Koopman, 1989] Philip Koopman. *Stack computers: the new wave*. figshare, 1989.

[Koopman, 1992] Philip Koopman. Preliminary exploration of optimized stack code generation. In *Rochester Forth Conference*, pages 111–111. J FORTH APPLICATION AND RESEARCH, 1992.

[Lattner and Adve, 2004] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[Maierhofer and Ertl, 1998] Martin Maierhofer and M Anton Ertl. Local stack allocation. In *International Conference on Compiler Construction*, pages 189–203. Springer, 1998.

[Shannon and Bailey, 2006] Mark Shannon and Chris Bailey. Global stack allocation–. In *22nd EuroForth Conference*, page 13. Citeseer, 2006.

[Shi *et al.*, 2008] Yunhe Shi, Kevin Casey, M Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(4):2, 2008.

[Wood and others, 2014] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.