Smalltalk JIT Compilation: LLVM Experimentation

Janat Baig[1,†], Dave Mason[2,*,†]

Toronto Metropolitan University, Toronto, Canada

Abstract

This paper discusses the ongoing development of the Zag Smalltalk LLVM JIT Compiler project. The project is aimed at enhancing the performance of dynamic languages through JIT compilation using LLVM. We highlight the project's rationale, emphasizing LLVM's optimization capabilities and architectural support. Our methodology focuses on generating LLVM IR code and exploring optimization strategies to improve execution efficiency.

Keywords

LLVM, JIT code generation, Smalltalk

1. Introduction and Motivation

Zag Smalltalk is a from-scratch implementation of a Smalltalk virtual machine whose low-level is implemented in Zig. The project aims to implement a higher-performance Smalltalk as well as to inspire and support existing OpenSmalltalk systems. To further advance Zag's performance, the integration of a Just-In-Time (JIT) compiler using LLVM[1] is underway. This paper explores the initial efforts to develop the JIT compiler, underlining its potential to improve speed and performance of execution by leveraging LLVM's extensive architecture support and optimization features that are applied to it's intermediate representation (IR).

Code generation models cover a spectrum ranging from interpreters handling intermediate bytecodes to compilers producing native machine code. Currently, Zag Smalltalk is using an intermediate strategy known as threaded execution, which falls between these two extremes.[2]

Although that has good performance, we also want a JIT to maximize performance. For a variety of reasons we believe LLVM is the most viable path to generating a high-quality JIT as quickly as possible.

We will use the `fibonacci` code in listing 1 for examples in this paper.

```
fibonacci
    self <= 2
        ifTrue: [ ^ 1 ].
    ^ (self-1) fibonacci + (self-2) fibonacci
```

Listing 1: Smalltalk Source for fibonacci

2. Zag execution model

Zag uses a continuation-passing-style of execution. A send saves return addresses in a Context and then passes control through a tail-call to the new method.

A return retrieves the return addresses from the Context and then passes control again through a tail-call to the ReturnBlock described below.

```
pub fn pushLiteral0(pc: PC, sp: SP, process: *Process, context: *Context, sig: MethodSignature)
    callconv(stdCall) SP {
```

IWST 2024: International Workshop on Smalltalk Technologies, July 9–11, 2024, Lille, France

*Corresponding author.

†These authors contributed equally.

janatbaig2@gmail.com (J. Baig); dmason@torontomu.ca (D. Mason)

https://github.com/JanBaig/ (J. Baig); https://sarg.torontomu.ca/dmason (D. Mason)

0000-0002-2688-7856 (D. Mason)

```
    const newSp = sp.push(Object.from(0));
    return @call(tailCall, pc.prim(),           .{ pc.next(), newSp, process, context,
        undefined });
}
```

Listing 2: Zag calling convention

Listing 2 shows a very simple function (in Zig) showing the values passed through the tail-call mechanism. `pc` points to a threaded program counter. `sp` points to the top of stack. `process` points to the private data of this process (which includes the stack). `context` points to the context of the calling method. `sig` has the selector and class of the target method.

3. Methodology

As this is exploratory work, we've opted for a from-scratch implementation of a textual LLVM IR generator rather than invoking the LLVM IR Builder via it's C API through a Foreign Function Interface (FFI). Several factors influenced this decision:

1. textual IR is easy to verify by eye, making it straightforward to ensure we are generating the correct code;
2. debugging textual IR is simpler and avoids the potential segmentation faults associated with debugging in-memory LLVM IR produced by the IR Builder, providing a smoother debugging experience;
3. LLVM has tools that can transform textual IR through various optimizations to machine-dependent object code, allowing us to evaluate the quality of our code generation and our choice of optimizations effectively.

Given the need for rapid iteration in exploratory work, generating textual IR offers a practical and efficient approach. Additionally, since Zag Smalltalk does not yet support a development environment, we are initially developing our textual LLVM code generator on Pharo[3].

3.1. Zag Code Generation

Zag Smalltalk stores all methods as Abstract Syntax Trees (ASTs). When a method is first required to be executed, it must be converted from an AST to either a threaded method or a JITed method. It is beyond the scope of this paper to discuss the decisions about when a JITed version is required.

3.1.1. Abstract Syntax Tree

The AST has Method and Block objects, which both have bodies composed of a sequence of expressions, optionally terminated with a return. Expressions include: variable Assign, Array, Send/Cascade, Literal, variable Reference, Self/Super, and ThisContext. This is a simplified version of the Pharo AST, and can easily be generated from Smalltalk source, or by walking a Pharo AST.

Regardless of whether generating for a threaded or native target, the AST compiler works in two stages: conversion to basic blocks, followed by target code generation.

3.1.2. AST to Basic Blocks

The first stage converts the AST into a series of basic blocks, made up of a sequence of stack operations (sends, pushes, pops, returns, etc.). Each basic block ends with a send, return, branch, or class switch. Part of this stage is an optional target-independent inlining of methods and blocks. As part of inlining, many of the sends may be replaced with switches based on the class of the receiver.[4]

The first basic block is a MethodBlock. A MethodBlock sets up a compile-time stack that represents the `self`, parameters, and locals of the method (or BlockClosure). The basic conversion outputs a sequence

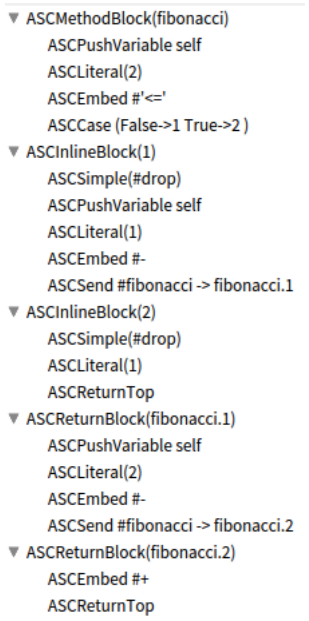

Figure 1: Inlined basic blocks for `fibonacci`

of pushes and pops, mimicking the runtime stack as well as adding the corresponding operations to the current basic block. When a send or class switch is required it forces the creation of a new basic block.

If this was a send then the new basic block will be a ReturnBlock. Because the send will execute code we know nothing about, any and all values must be represented on the stack or the heap. This is the exact semantics we expect from the stack operations, but as we will see below this requires particular attention when generating LLVM code. A ReturnBlock will be initialized with a stack as it would be on return from the message send, that is with the stack as it was just before the send, but with the receiver and parameters of the send replaced by a result value. Then we resume the basic conversion.

If this was a class switch the new basic blocks will be an InlineBlocks. Each of the new basic blocks will have a stack initialized from the stack at the point of the creation of the class switch. Because of the structural nature of the replacement of the send → class switch, all of the various paths will eventually come to a common InlineBlock which will have a stack the same as if the send was still in place, that is with the receiver and parameters replaced by a result. A path that is a sequence of InlineBlocks is highly advantageous for a target like LLVM because they will all be part of a single function and values don't need to be actually saved to memory, as we'll see below. If, along any of the paths an actual send is required, because there is no safe inlining that can be done, a ReturnBlock may be required, in which case the common block will be converted to a ReturnBlock, at some performance cost for a target such as LLVM.

Figure 1 shows a version of `fibonacci` converted to basic blocks and fully inlined. Note there are still 2 sends because the calls are recursive.

After all the basic blocks have been generated, the data-flow graph is updated for the basic blocks, if it is required by the target code generator. This integrates the *require*/*provides* values of the various blocks. We iterate over all the blocks, taking the values that are required for a given basic block and telling all of the blocks that transfer to it that it needs those values. This may make those basic blocks now require the values, so we continue the iteration until we reach a fix-point, where nothing is changing. This analysis can reduce the memory traffic for native targets such as LLVM.

3.1.3. Basic Blocks to Target Code

The final stage is either generation of threaded code or JIT code. Threaded code generation is very straight-forward, and will not be discussed here.

Since LLVM uses a register-transfer model, the first stage of LLVM generation is to convert the stack operations into Single Static Assignment (SSA) form. Static Single Assignment (SSA) form is a property of the LLVM IR in which each variable is assigned exactly once and is defined before it is used. This simplifies optimization processes in the compiler by providing a clear and unambiguous mapping of variables to their values. In our textual IR generation, we use sequence numbers for variables to conform to SSA form. Each variable is assigned a unique sequence number upon assignment, ensuring that it is defined exactly once throughout the IR. The values are retrieved from the stack on return from a message send and need to be reflected to the stack before a subsequent message send. Between sends, values live in LLVM variables (mostly registers).

Listing 3 shows an extract of the LLVM code for fibonacci which shows retrieving the value of `self` from the stack. It is given the name `%.self` because it will be accessed from all the InlineBlocks of the `@fibonacci` or `@fibonacci.1` functions, such as the blocks `%1` and `%2`. Other values that "change" are passed along via the `phi` chains.

```
%zag.process.Process = type { [909 x i64], ptr, i64, ptr, ptr, ptr, ptr, ptr, ptr, i64, [3637 x
    i64], [3637 x i64] }
%zag.context.Context = type { i64, ptr, %zag.execute.PC, ptr, ptr, i64, [0 x i64] }
%zag.execute.PC = type { ptr }
%zag.execute.Code = type { <{ ptr, [8 x i8] }> }
%zag.execute.Stack = type { i64, i64, i64, i64 }
%zag.execute.CompiledMethod = type { i64, i64, %zag.execute.MethodSignature, ptr, [0 x %zag.
    execute.Code] }
%zag.execute.MethodSignature = type { i32, i16, [2 x i8] }
@fibonacci.CM = global %struct.CompiledMethod { i64 0, %union.Object { i64 0 }, %struct.
    MethodSignature 0, ptr verifyMethod, [3 x %union.Code] [%union.Code { ptr @fibonacci }, %union
    .Code { ptr @fibonacci.1 }, %union.Code { ptr @fibonacci.2 }]}, align 8
define ptr @fibonacci(ptr noundef %pc, ptr noundef %sp, ptr noundef %process, ptr noundef %context
    , i64 %signature) #1 {
  %.self = load i64, ptr %sp, align 8
  %3 = i64 -1125899906842621 ; pushLocal 2
  %4 = icmp ule i64 %.self, %1 ; embed: <=
  br i1 %4, label %2, label %1 ; case False->%1 True->%2
%1: ...

%2: ...
}
define ptr @fibonacci.1(ptr noundef %pc, ptr noundef %sp, ptr noundef %process, ptr noundef %
    context, i64 %signature) #1 {
  %.self = load i64, ptr %sp, align 8
  ...
```

Listing 3: Extracct of LLVM code for fibonacci

To generate the LLVM code, we go through each basic block, mimicking the operations that will occur at runtime system. If we push something on the stack, there is no memory operation, but simply assigning the value to a newly named LLVM "register" such as `%3` then we push that register name on the compile-time stack. Then to access the top of stack, we pop the register name off the compile time stack and output a reference to that register. We also don't modify the stack pointer, but rather maintain at compile time where the stack pointer is. If a value is required in a subsequent InlineBlock, that block will start with a `phi` expression such as `%14 = phi ptr [ %1, %5 ], [ %39, %37 ]` that specifies for each source block the name that the value had in that block. In this example, it's saying to combine the value `%1` from block `%5` and the value `%39` from block `%37` into a new register `%14`. LLVM does register allocation so as to minimize the amount of register shuffling required. Any given register, like `%14`, will only be assigned in one place.

In this way, all of the operations within a MethodBlock or a ReturnBlock may be able to be just register operations. However, when we are going to other code, either with a return or a send, the register values will be spilled to the stack.

3.2. LLVM Compilation Process

Once we've output the textual LLVM IR, we use the command line to invoke the LLVM compiler (LLC) which can perform various optimization passes on the IR. LLVM has a spectrum of built-in optimizations that can be performed in any order. Beyond these optimization options that are pre-defined, LLVM also provides a framework for creating custom optimization passes. As our code generation is very idiosyncratic we will also explore these in an attempt to further enhance performance by allowing us to tailor the optimization process to our specific needs. After performing optimizations, LLC then outputs an object file (.o) which can be linked with the Zag runtime environment from which we can execute benchmarks to evaluate performance.

4. Conclusion

By generating textual LLVM IR code, we can explore appropriate code generation and LLVM optimizations which allows us to validate the viability of using LLVM to produce our JITed code.

LLVM code generation fits very nicely into our code generation model. Combined with our aggressive inlining, we anticipate signicicant performance boost.

Future Work

If the performance improvements are compelling, we will connect the LLVM IR Builder into the Zag runtime and generate the JITed code on the fly in Zag. As the project advances, further details and results will be available to substantiate the anticipated benefits of this innovative approach to improving dynamic language performance.

References

[1] C. Lattner, V. Adve, Llvm: A compilation framework for lifelong program analysis & transformation, in: International symposium on code generation and optimization, 2004. CGO 2004., IEEE, San Jose, CA, USA, 2004, pp. 75–86.

[2] D. Mason, Threaded execution as a dual to native code, in: Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming, Association for Computing Machinery, 2023, p. 7–11. URL: https://doi.org/10.1145/3594671.3594673.

[3] Pharo Development Team, Pharo: A pure object-oriented programming language and a powerful environment, http://pharo.org/, 2008.

[4] D. Franklin, D. Mason, Inlined code generation for smalltalk, in: International Workshop on Smalltalk Technologies, 2024.