Computer Architecture Honors Contract in Just-In-Time Compilers

Kartavya Vashishtha

Introduction

Just-In-Time (JIT) compilers are programs that simultaneously translate source code to target code while executing target code. This can be understood in contrast with Ahead-Of-Time (AOT) compilers that translate source code to target code and produce an artifact which may be executed later.

This report discusses the implementation of `k-lox-jit`, a JIT compiler implemented in Rust that translates from an invented language called "Lox (Kartavya's Version)" (see Appendix A for details) to 64-bit ARMv8 A64 assembly.

Utility

JIT compilers are utilized in areas such as kernel observability (eBPF in Linux), JavaScript implementations (in all major browsers), and Hotspot VM (a Java implementation).

JIT compilers are useful in areas where:

- shipping machine code to end-users is impractical (due to security concerns, inability to provide builds for all architectures, size constraints)
- minimizing time-to-run (time from source file to result) is important

The second requirement disadvantages interpreters since they're slower than executing machine code. While AOT compilers offer good runtime performance, they lack knowledge about "hot" (frequently executed) code paths, and thys may waste compile time optimizing rarely-used code.

A prime example of where JIT compilation shines is in web browsers executing JavaScript:

- Letting websites execute arbitrary assembly code on user machines can lead to security risks even with strong sandboxing. The JIT compiler can be trusted to produce non-malicious assembly from JavaScript.
- Assembly code is frequently much larger in size than the equivalent JavaScript.
- It would be inconvenient and difficult for JavaScript providers (websites) to provide builds for every computer architecture that may attempt to run their programs. Having the user device perform compilation from JavaScript allows websites to ship software in a cross-platform compatible manner.
- JavaScript is a high-level language that is easy to write and read, but slow to execute. JIT compilers can optimize the execution of JavaScript code to run faster than an interpreter.

In summary, JIT compilers provide security, portability, and optimized total execution time.

Implementation Overview

For the discussion below, we assume our target language to be machine code.

Given a source language, we can compile basic blocks to assembly, execute that assembly, then compile the next basic block that must be run. This forms a compile-evaluate loop. This ensures we only compile blocks that are going to be executed (acting as a dynamic version of dead code elimination), and we can use run-time information to optimize further compilation.

For `k-lox-jit`, functions serve as basic blocks; compilation occurs on the first call to a function, and generated assembly is cached for future calls. Compared with an AOT compiler, `k-lox-jit` can save

on compilation time in cases where there may be large functions in source code that are never called at runtime.

```
fn rec_fact (n) {
  if (n < 1) {
    return 1;
  } else {
    return n * rec_fact(n - 1);
  };
}

fn main () {
  return rec_fact(5);
}
```

Listing 1: Example `k-jit-lox` program

Frontend Overview

`lox-jit` performs AOT parsing, and translates source code into stack-based bytecode.

Parsing implementation

We use the `chumsky v0.9.3` parser combinator library to implement parsing. There is no separate lexing step, and all keywords are *contextual*: if they occur in non-ambiguous places, they can function as valid identifiers.

Example: `var while = 2; return while;` parses correctly.

Stack-based bytecode implementation

Source Function

```
fn rec_fact (n) {
  if (n < 1) {
    return 1;
  } else {
    return n * rec_fact(n - 1);
  };
}
```

Generated Bytecode Block

```
ByteCodeChunk {
  in_arg: 1, // n is pushed to top of stack
  code: [
    LoadVar { stack_idx: 0 },
    Constant { val: 1 },
    LessThan, // n < 1
    JumpIfZero { label_id: 0 }, // if (_) {
    Constant { val: 1 }, // return 1
    Return,
    Jump { label_id: 1 }, // } else {
    JumpLabel { label_id: 0 },
    LoadVar { stack_idx: 0 }, // n *
    LoadVar { stack_idx: 0 },
    Constant { val: 1 },
    Sub, // (n - 1)
    // rec_fact(n - 1)
    Call { fn_idx: 0, word_argc: 1 },
    Mul, // n * rec_fact(n - 1)
    Return, // return n * rec_fact(n - 1);
    JumpLabel { label_id: 1 }, // }
    // function cleanup, unused in this case
    Pop { count: 1 },
    Constant { val: 0 },
    Return,
  ],
  out_args: 0,
}
```

Listing 2: Generated bytecode for a recursive factorial function

Backend Overview

The stack-based bytecode is translated to ARMv8 assembly. The bytecode stack is mapped to the ARM stack pointer (`sp`), and generated assembly instructions operate on the stack. In particular, we maintain a grow-down stack where `sp` points to the top-most valid element. Note that stack indexing and loading variables from the stack is done using the base pointer.

For example, `LoadVar {stack_idx: 1}` is translated to loading the value two slots below the frame pointer.

Bytecode Block

```
Constant { val: 2 },
Constant { val: 3 },
LoadVar { stack_idx: 0 },
LoadVar { stack_idx: 1 },
Add,
Return
```

Generated ARM Assembly

```
; Prologue (setting up frame pointer)
stp x29, x30, [sp, #-0x10]!
mov x29, sp
mov x0, #2 ; Constant { val: 2 }
str x0, [sp, #-0x10]!
mov x0, #3 ; Constant { val: 3 }
str x0, [sp, #-0x10]!
sub x0, x29, #0x10 ; LoadVar { stack_idx: 0 }
ldr x0, [x0]
str x0, [sp, #-0x10]!
sub x0, x29, #0x20 ; LoadVar { stack_idx: 1 }
ldr x0, [x0]
str x0, [sp, #-0x10]!
ldr x0, [sp], #0x10 ; Add
ldr x1, [sp], #0x10
add x0, x0, x1
str x0, [sp, #-0x10]!
ldr x0, [sp], #0x10 ; Return
mov sp, x29
ldp x29, x30, [sp], #0x10
ret
```

Listing 3: Generated bytecode for a recursive factorial function

Function calls translate to calls to `call_fn` with the index number of the callee function.

Pseudocode for the `call_fn` procedure:

```
fn call_fn(function_idx: u32, argv: *const i64, argc: u32) -> i64 {
  if !cache.contains(function_idx) {
    cache[function_idx].set(compile(function_idx))
  }

  let compiled = cache.get(function_idx);

  return compiled(argv);
}
```

`call_fn` is passed the function index, a pointer to the top of the stack and the number of passed arguments, which is used to copy over the arguments (stack's top values) into the new function's stack.

The `Op::Call { fn_idx: idx, word_argc }` bytecode is translated to:

```
ldr x0, ->cbc_ptr
mov w1, fn_index
mov x2, sp ; argv is current stack pointer
mov x3, word_argc ; number of arguments being passed
ldr x4, ->call_fn
blr x4 ; call function
```

```
add sp, sp, #(word_argc * 16) ; pop args
str x0, [sp, #-16]! ; store return value on stack
```

Future Improvements

- Reduce basic-block size from functions to syntactical blocks (such as if-statements, while-loops, etc.)
- On compilation of function, rewrite calling code to directly call the generated assembly for future calls instead of looking up compiled functions in cache.
- Add types.
- Utilize run-time information to specialize compiled functions to specific call-site. For example, if a loop calls a function with integer arguments, we can specialize the generated assembly to assume integer arguments and improve performance. In the case when we are passed a non-integer argument, we despecialize to a generic compilation that can handle all types.
- Implement register allocation to massively reduce stack manipulation overhead. There is a prototype present in the repository's main branch, but it cannot handle branches right now (and is generally unreliable).
- Introduce a register-based (possibly SSA) IR to optimize register allocation.

Acknowledgements

Jinglei Cheng mentored me and Harry Zheng in building our implementations.

Appendix A: Lox (Kartavya's Version) Specification

Syntax in BNF form

```
ident ::= [a-zA-Z] [a-zA-Z0-9_]*

num ::= [0-9]+

atom ::= ident
       | num
       | '(' atom ')'

arg_list ::= expr (',' expr)*

call ::= ident '(' arg_list? ')'

unary ::= '-' unary
        | call
        | atom

product ::= unary '*' unary
          | unary '/' unary
          | unary

sum ::= product '+' product
      | product '-' product
      | product

expr ::= sum '>=' sum
       | sum '<=' sum
       | sum '<' sum
       | sum '>' sum
       | sum

var_set ::= ident '=' expr
var_decl ::= 'var' ident '=' expr
return ::= 'return' expr

if ::= 'if' '(' expr ')' stmt_block ( 'else' stmt_block )?
while ::= 'while' '(' expr ')' stmt_block

stmt ::= (return | if | while | var_decl | var_set | expr) ';'
stmt_block ::= '{' stmt+ '}'

param_list ::= ident (',' ident)*
fn ::= 'fn' ident '(' param_list? ')' stmt_block

program ::= fn+
```

Listing 5: Brackus Naur Grammar for Lox

Example Programs

```
fn rec_fact (n) {
  if (n < 1) {
    return 1;
  } else {
    return n * rec_fact(n - 1);
  };
}

fn fact (n) {
  var acc = 1;
  var i = 1;
  while (i <= n) {
    acc = acc * i;
    i = i + 1;
  };
  return acc;
}

fn main () {
  return fact(5);
}
```

Listing 6: Calculating the factorial in k-lox, recursively and using a while-loop