A Surprisingly Simple Lua Compiler – Extended Version

Hugo Musso Gualandi
hgualandi@inf.puc-rio.br
PUC-Rio
Rio de Janeiro, Brazil

Roberto Ierusalimschy
roberto@inf.puc-rio.br
PUC-Rio
Rio de Janeiro, Brazil

ABSTRACT

Dynamically-typed programming languages are often implemented using interpreters, which offer several advantages in terms of portability and flexibility of the implementation. However, as a language matures and its programs get bigger, programmers may seek compilers, to avoid the interpretation overhead.

In this study, we present LuaAOT, a simple ahead-of-time compiler for Lua derived from the reference Lua interpreter. We describe two alternative compilation strategies. The first one exemplifies an old idea of using partial evaluation to produce a compiler based on an existing interpreter. Its contribution is to apply this idea to a well-established programming language. We show that with a quite modest effort it is possible to implement an efficient compiler that covers the entirety of Lua, including coroutines and tail calls. The whole implementation required less than 500 lines of new code. For this effort, we reduced the running time of our benchmarks from 20% to 60%.

The second compilation strategy is based on function "outlining", where each bytecode is implemented by a small subroutine. This strategy reduces executable sizes and compilation times, at the cost of not speeding up the running times as much as the first compilation method.

CCS CONCEPTS

• **Software and its engineering** → **Interpreters**; **Just-in-time compilers**.

KEYWORDS

dynamic languages, interpreters, partial evaluation, compilers, just-in-time compilers

1 INTRODUCTION

Dynamic programming languages are popular for many applications, including scripting. They are often implemented using an interpreter, which makes it easier to load code fragments at run-time and enables a fast test-change-recompile development loop. However, interpreted programs are often slower than compiled programs, due to the interpretation overhead.

The conventional wisdom is that the most efficient implementations for dynamic languages are just-in-time (JIT) compilers, which can take advantage of run-time information to perform speculative optimizations. Ahead-of-time compilers (AOT) have a steeper hill to climb, because they must rely on clever static analysis or type inference to inform their optimizations. Nevertheless, in this paper we show that a very simple compiler, focused exclusively on reducing the interpretation overhead, can deliver respectable improvements for a modest implementation effort. We also argue that such a simple compiler can provide useful insight about the performance of the interpreter that it is based on.

Our motivation for this paper was our previous work on Lua compilers, in particular the Pallene language [11]. Pallene is superficially similar to a typed dialect of Lua, where the type information allows the compiler to perform significant optimizations to the code. Because the type information is not speculative, Pallene's compiler can work ahead of time and be simpler than a JIT compiler. However, a relevant question is how much of this improvement is due to the types and how much of it is due to just using a compiler instead of an interpreter. To help answer this question, we developed LuaAOT, a simple ahead-of-time compiler for Lua which does not perform any type-based optimizations.

The inspiration for the architecture of LuaAOT is an old idea of producing a compiler from an existing interpreter by unrolling and specializing the core interpreter loop [8, 20]. Our contribution is to show that this idea can be successfully applied to an established language. Using less than 500 lines of new code, our compiler implements the entirety of Lua, including coroutines and tail calls.

One thing that contributes to the simplicity of LuaAOT is that we can delegate a significant part of the work to a C compiler. We get several optimizations "for free", including constant propagation and dead code elimination. This allows us to remove much of the interpreter overhead while still emitting straightforward code that is mostly copied from the existing interpreter.

Before going on, we should emphasize that LuaAOT catches the low-hanging-fruits of compiler optimizations for dynamic languages. Its performance is not competitive with a reasonable JIT compiler. Its selling point is that it achieves a decent performance boost for a surprisingly low cost.

The next section has a brief discussion about partial evaluation of virtual machines. Section 3 describes LuaAOT, our take on that idea. Next, we evaluate our artifact in Section 4. Finally, we discuss related work in Section 5 and draw some conclusions in Sections 6 and 7.

This paper is an extended and expanded version of a previous paper [12] published at the Brazilian Symposium of Programming Languages. The main changes in this expanded version are the addition of more benchmarks and a compilation method that uses function "outlining" to reduce the size and compilation time of the generated code.

2 VIRTUAL MACHINES AND PARTIAL EVALUATION

One of the most popular ways to implement an interpreter for a dynamically typed programming language is via a virtual machine. The virtual machine defines an intermediate language of portable instructions, also called *bytecodes*. This approach is illustrated in Figure 1, which shows a small Lua function and the corresponding

```
function foo(a, b, c)
    local d = 17
    while true
        a = b + c
        b = b + d
    end
end


Instruction foo[] = {
    { LOADI, 3, 17 },
    { ADD, 0, 1, 2 },
    { ADD, 1, 1, 3 },
    { JUMP, -3 }
};
```

Figure 1: A Lua function and its bytecode.

portable instructions for the Lua virtual machine. The first instruction, LOADI 3 17, loads the integer constant 17 into the register R[3] (the address of the local variable 'd'). Then, the next instruction ADD 0 1 2 adds the content of the R[1] and R[2] registers (variables 'b' and 'c'), storing the result in R[0] (variable 'a'). The next ADD is similar. Finally, the JUMP instruction jumps three instructions back in the code, repeating the loop. Note that Lua optimized the test for the while condition, given that it was a constant. To ease the presentation, we represented these instructions using records. The actual Lua interpreter encodes the instruction components as bit-fields of a 32-bit integer [13].

Figure 2 shows a typical inner loop of a virtual machine. It executes the portable instructions, one by one, like a conventional CPU. The interpreter maintains a stack, which is where the local variables are stored. The program counter points to the current instruction and guides the control flow. Data operations, such as LOADI and ADD, manipulate the values in the stack. Control-flow operations, such as JUMP, modify the program counter. In this example, `DoAdd` is a macro that does the actual work, including checking the types of the arguments.

The virtual machine in our example is register-based, similarly to the Lua virtual machine [13]. The defining characteristic of a register-based virtual machine is that the data-manipulation instructions can read from and write to any position in the stack. The other common way to design a virtual machine is in a stack-based discipline, where the data manipulation instructions always push values to the top of the stack and pop results from its top. We believe that the technique we describe in this paper should also apply to stack-based virtual machines. Although stack and register VMs differ in how they encode the opcodes, the basic dispatch mechanism and switch/case is similar.

In this paper we are interested in the interpretation overhead that is associated with decoding and dispatching virtual-machine instructions. The decoding overhead comes from fetching the next virtual instruction from memory and computing the values of its parameters; in the example, those would be the `tag` and `arg` fields. The dispatch overhead happens as the interpreter transfers the control to the appropriate instruction handler. The most basic form is a while-switch loop, but some interpreters might use more advanced

```
void execute(Instruction prog[], Value stack[])
{
    int pc = 0;
    while (1) {
        Instruction instr = prog[pc++];
        switch (instr.tag) {
            case LOADI: {
                int dst = instr.arg1;
                int val = instr.arg2;
                stack[dst] = IntValue(val);
                break;
            }
            case ADD: {
                int dst  = instr.arg1;
                int src1 = instr.arg2;
                int src2 = instr.arg3;
                stack[dst] = DoAdd(stack[src1],
                                   stack[src2]);
                break;
            }
            case JUMP: {
                pc += instr.arg1 - 1;
                break;
            }
        }
    }
}
```

Figure 2: A virtual machine / interpreter.

dispatch techniques such as "threaded code" [7]. The Lua 5.4 interpreter can be configured to use either a portable while-switch loop or a dispatch table that takes advantage of the computed-goto GCC extension.

To minimize the decoding and dispatching overheads, LuaAOT produces a modified version of the inner interpreter loop that is specialized to run a given function. Figure 3 provides an example of this idea. The instructions become compile-time constants and the jumps become goto statements. The `execute_foo` function can be seen as a partial evaluation of the `execute` function, where the `prog` argument is fixed to be the `foo` array from Figure 1. This method to produce a compiler from an interpreter is sometimes called a Futamura Projection [8].

This simple compilation strategy does not optimize all the things that an advanced Lua compiler can try to optimize. For example, there is no attempt to store Lua variables in CPU registers. Similarly to the Lua interpreter, LuaAOT stores all local variables in the Lua stack. However, the simple compilation strategy does provide an idea of what can be achieved by optimizing the low-hanging fruit. In particular, it can tell us about the interpretation overhead of the original interpreter. Since the bytecode instructions are compile-time constants, the C compiler can use constant propagation to remove most of the operations for decoding the instructions. Similarly, because the Lua jumps are converted to C gotos, we avoid indirect jumps and dispatch tables. The control flow graph also is exposed to the C compiler, possibly allowing further optimizations.

```
void execute_foo(Value stack[])
{
    L0: {
        Instruction instr = { LOADI, 3, 17 };
        int dst = instr.arg1;
        int val = instr.arg2;
        stack[dst] = IntValue(val);
    }
    L1: {
        Instruction instr = { ADD, 0, 1, 2 };
        int dst  = instr.arg1;
        int src1 = instr.arg2;
        int src2 = instr.arg3;
        stack[dst] = DoAdd(stack[src1], stack[src2]);
    }
    L2: {
        Instruction instr = { ADD, 1, 1, 3 };
        int dst  = instr.arg1;
        int src1 = instr.arg2;
        int src2 = instr.arg3;
        stack[dst] = DoAdd(stack[src1], stack[src2]);
    }
    L3: {
        // JUMP
        goto L1;
    }
}
```

Figure 3: Specializing the interpreter to a particular function.

3 THE LuaAOT COMPILER

In this section we describe how we built the LuaAOT compiler and which challenges we encountered in the process. The compiler is free software and the source code is publicly available [10].

Lua, like many scripting languages, can be extended by binary modules written in other languages (such as C). This was a natural mechanism for integrating LuaAOT with Lua. The core of the LuaAOT compiler is the `luaot` executable, which takes Lua source code as input and outputs such a binary extension module. For the most part these compiled modules can be loaded by the Lua interpreter and runtime just like any other binary module. The only difference is that we had to patch the interpreter to tell it how to call our compiled Lua functions.

A possible alternative implementation would be for the LuaAOT compiler to generate a standalone executable. The standard procedure for this would be to bundle the interpreter inside the executable, because the compiled code still needs the Lua runtime. However, creating standalone programs for Lua in this manner is not typical. It is more common to install and distribute Lua applications separately from the interpreter.

3.1 The Interpreter

The Lua interpreter plays a central role in our system, which comprises of both a compiler and a slightly modified interpreter. There are multiple reasons for this. The first is that programs can contain both compiled and non-compiled sections and the interpreter is necessary to run the non-compiled parts. Moreover, our compiled code still requires the interpreter: because we partially evaluate the inner interpreter loop, the compiled code calls several subroutines from the interpreter. Furthermore, the interpreter codebase also houses the Lua runtime and garbage collector, which are used by both the compiled and the non-compiled code.

The custom interpreter has few changes compared to the original Lua interpreter. The first modification was to add an additional field to the data structure that represents Lua functions. This field refers to the compiled code for that function, if it exists. The C extension modules that we generate include initialization code that associates the Lua functions with their compiled C implementation.

After this, we told the interpreter how to use these compiled functions. At the start of the `execute` subroutine (the one that Figure 2 talks about), we add a check just before the inner loop. If the function has a compiled version we transfer the control to that, instead of continuing to the usual interpreter loop.

The next change we made is related to the public interface that is exposed to C extension modules. Our partial-evaluation generates code that calls many internal functions from the Lua interpreter, which in normal circumstances are not exposed to extension modules. To allow our generated code to use these internal functions, we modified the interpreter to make all those internal names public.

Finally, we proceeded to implement the code generator, examining the bytecodes one by one. Most had their code directly pasted into the compiler; some required modifications to the generated code, but there was one case that also required modifications to the interpreter: the bytecodes for function calls (CALL, TAILCALL, and RETURN). The Lua 5.4 interpreter has an optimization where Lua-to-Lua calls reuse the same execution frame for the `execute` function. Like a conventional CPU, the Lua interpreter implements these function calls by updating the `prog` argument and the program counter, so that the same interpreter loop naturally runs the called function. Unfortunately, this implementation is incompatible with our compiled code, where each `execute` function is specialized to a particular Lua function. Our solution was to disable this optimization. We believe that with additional work it might have been possible to keep it. However, disabling it was certainly simpler.

We should stress that this change did not harm the tail-recursive functions, which are guaranteed to use O(1) stack space. In the generated C code for the TAILCALL instruction, the crucial function call appears in a tail position, so that a good C compiler can perform the required tail-call optimization.

As important as the changes we made are the many things we did not need to modify. Other than adding a single field to the function objects, we made no other changes to the internal Lua data types. We also did not modify the Lua runtime system, the garbage collector, or the Lua standard library.

3.2 The Code Generator

The code generator receives a Lua module and converts it to C. To do this it calls the Lua bytecode compiler and then it converts the bytecode to C. For each function in the module, the code generator produces an appropriate function header and then it iterates over the function's bytecodes, outputting a block of C code for each

instruction. For the most part, these blocks of C code are copied verbatim from the original Lua interpreter. However, we had to adapt a few bytecodes. The main categories were bytecodes that modify the program counter, bytecodes using C gotos, and bytecodes related to function calls.

In the original interpreter, jumps are implemented by assigning to the interpreter variable representing the program counter. In our compiler, we replaced each of these jumps by a C goto. This required making the appropriate changes to the generated code for the JUMP instruction, as well as to the instructions that implement for-loops. We also had to change the instructions for binary operations, because of how Lua implements operator overloading. In Lua, every binary operation is followed by a special instruction (MMBIN), which handles overloading. When the operands have the expected types (e.g., numbers for the ADD operation), the binary operation increments the program counter to skip this next instruction. This means that all binary operations contain an implicit jump, which our compiler must also replace by a goto.

The next category of instructions we had to adapt were the ones that use goto statements in their original implementation. One example is the FORCALL instruction, which is always followed by a FORLOOP. As an optimization, the Lua interpreter uses a goto to jump straight to the handler for the FORLOOP, bypassing the usual dispatch logic. Since our compiler is already removing the run-time instruction dispatching, we simply removed this optimization from our generated code.

The instructions for function calls also had the issue of gotos in the original implementation, as we discussed in the previous section. However, in this case we had to apply our changes both to the interpreter and to the generated code, so that uncompiled Lua functions could properly call the compiled ones.

3.3 Coroutines

Lua's coroutines [5, 6] are a powerful control-flow mechanism, useful for asynchronous programming. They operate in a similar space to generators and delimited continuations. Lua implements coroutines by maintaining a separate call stack for each coroutine. When a coroutine yields, the interpreter saves the current program counter and exits from the interpreter loop (via a longjmp). When the coroutine is resumed, the interpreter must continue the execution loop from where it left.

To make our compiler compatible with coroutines, we had to teach it how to restart the execution from the point where the coroutine was interrupted. To do this, we need to jump to the appropriate location in the code, according to the saved value of the program counter. Figure 4 illustrates how we do this. At the start of the function, we insert a switch-case that jumps to the location indicated by the saved program counter. The rest of the compiled function, including the jump labels, is the same as the version without coroutine support, shown in Figure 3. The switch case is used only once, at the start of the function. After this, all the other jumps happen as previously described, with gotos.

3.4 Alternative Compilation Without Gotos

A fragile aspect of LuaAOT is the optimization of replacing Lua jumps by C gotos. While the implementation of the Lua interpreter

```
void execute_foo(Coroutine *f, Value stack[])
{
    switch (f->savedpc) {
        case 0: goto L0;
        case 1: goto L1;
        case 2: goto L2;
        case 3: goto L3;
    }
    L0: { /* ... */ } // LOAD
    L1: { /* ... */ } // ADD
    L2: { /* ... */ } // ADD
    L3: { /* ... */ } // JUMP
}
```

Figure 4: Support for Lua coroutines.

gave us the opportunity to use gotos without too much trouble, doing this to a different interpreter might have been more difficult. For example, the Lua interpreter is written in C, which is a language with gotos. Had it been written in a different language, it might have been harder to use gotos in the partial evaluation process. Another important aspect is that Lua's core interpreter loop is all in a single function. Had it been broken into smaller subroutines, it might have been more difficult to use gotos because one subroutine would not be able to use goto to jump to the other.

In response to this, we developed an alternative design for jumps, this time using a trampoline-like pattern instead of gotos. As we can see in Figure 5, there is still a switch-case. However, it dispatches based on the program counter instead of based on the instruction tag. For instructions that do not jump, the handler falls through to the handler for the next instruction. For the instructions that do modify the program counter, the handler jumps back to the start of the switch-case, which then re-dispatches to the desired jump target. The trampoline analogy comes from this two-step jumping scheme: the break statement drops back to the start of the loop and then the switch-case bounces us back to the destination label.

In terms of implementation effort the trampoline approach is even simpler than the goto-based one. Most of the manual modifications that we described in Section 3.2 involved replacing Lua jumps with C gotos. In the trampoline implementation, the vast majority of those sections can be copy-pasted without any changes at all. This design also simplifies the support for coroutines, because the trampoline already includes the switch-case that the coroutines need. The only difference is the initial value of the program counter: instead of always starting from zero (first instruction, as is done in Figure 5), it should start from the program counter value that the coroutine saved (as is done in Figure 4).

The obvious downside of trampolines is that they preserve some of the dispatch-related overhead from the interpreter. We will evaluate this overhead in Section 4.

3.5 Alternative compilation with opcode subroutines

As we will discuss in more detail in Section 4, we observed that LuaAOT can generate fairly large binaries due to how it effectively "inlines" and copy-pastes the implementation of every VM

```
void execute_foo(Value stack[])
{
    int pc = 0;
    while (1) {
        switch (pc) {
            case 0: {
                Instruction instr = { LOADI, 3, 17 };
                int dst = instr.arg1;
                int val = instr.arg2;
                stack[dst] = IntValue(val);
                // fallthrough
            }
            case 1: {
                Instruction instr = { ADD, 0, 1, 2 };
                int dst  = instr.arg1;
                int src1 = instr.arg2;
                int src2 = instr.arg3;
                stack[dst] = DoAdd(stack[src1],
                                   stack[src2]);
                // fallthrough
            }
            case 2: {
                Instruction instr = { ADD, 1, 1, 3 };
                int dst  = instr.arg1;
                int src1 = instr.arg2;
                int src2 = instr.arg3;
                stack[dst] = DoAdd(stack[src1],
                                   stack[src2]);
                // fallthrough
            }
            case 3: {
                Instruction instr = { JUMP, -3 };
                pc += instr.arg1 - 1;
                break;
            }
        }
    }
}
```

Figure 5: Compilation without gotos, using a trampoline.

```
typedef struct {
    Instruction *prog;
    Value *stack;
    int pc;
} State;

void do_LOADI(State *st, int dst, int val) {
    st->stack[dst] = IntValue(val);
}

void do_ADD(State *st, int dst, int src1, int src2) {
    st->stack[dst] = DoAdd(st->stack[src1],
                           st->stack[src2]);
}

void execute_foo(State *st)
{
    L0: do_LOADI(st, 3, 17);
    L1: do_ADD(st, 0, 1, 2);
    L2: do_ADD(st, 1, 1, 3);
    L3: goto L1;
}
```

Figure 6: Extracting regular opcodes into functions.

```
// Generated code for a TFORPREP,
// which ends with an unconditional jump
op_TFORPREP(st, 0);
goto L09;

// Generated code for a TFORLOOP,
// which ends with a conditional jump
if (op_TFORLOOP(st, 0))
    goto L06;
```

Figure 7: Extracting control-flow opcodes into functions

instruction. With this problem in mind, we designed an alternative implementation in which each opcode is split off into a separate subroutine.

This technique is illustrated in Figures 6 and 7. The first thing we did was refactor the local variables that represent the interpreter state (stack, program counter, etc.), placing that data in a struct that can be passed by reference. Then, for each instruction we created a helper function that receives the instruction parameters as arguments. Keep in mind that while the helper functions in Figure 6 only contain a single line of code, in the real interpreter they are typically longer than that.

The body of the compiled Lua function consists of a series of function calls to the helper functions, one per instruction. Our compilation still gets rid of the run-time instruction decoding and dispatching, as those instruction parameters are encoded as constants passed as function arguments.

Similarly to before, control-flow instructions that modify the program counter are treated specially. For unconditional jump instructions, we put the goto statement right after the call to the opcode subroutine. For conditional jump instructions, the subroutine returns a boolean saying whether to make the jump. Finally, for the JMP opcode, which is just a single goto, we did not create a helper function and just inlined its definition as before.

Another change we made was related to common opcode pairs. There are a couple of places where Lua encodes an operation as a sequence of two opcodes, because they would not fit in a single instruction. The most common case where this happens are the MMBIN instructions that follow every binary arithmetic instruction, which we discussed in Section 3.2. LuaAOT does not share the same limitation of limited space to encode instructions, so what we did was merge the MMBIN instructions into the preceding instruction.

We hypothesized that this function-based compilation would produce a smaller amount of C code, which could lead to better compilation times and smaller executables. For running time, it was harder to guess. In one direction, having smaller executables could lead to better performance due to improved instruction cache usage. However, refactoring the local interpreter variables into a struct and introducing function call overhead could potentially harm the performance.

4 EVALUATION

To evaluate LuaAOT, we studied its performance and we measured the complexity of its implementation. We also made a qualitative analysis of what are the requirements that an interpreter must fulfill to allow a compiler in the style of LuaAOT. The source code for the benchmarks and the related scripts are available online, in the same repository as the compiler [10].

4.1 Correctness

Our main tool for assessing the correctness of LuaAOT was the test suite from the reference Lua interpreter [15]. Although tests cannot prove the absence of bugs, they certainly help. We ensured that LuaAOT could run the full Lua test suite without crashing and that it produced the same result as the reference interpreter. During this process we found and fixed several bugs, including some tricky ones involving recursive functions and tail calls.

4.2 Running time

To study the performance of LuaAOT, we compared the running time of the compiled programs with the running time of the interpreter. To provide a baseline, we also compared the results with LuaJIT [21], an advanced just-in-time compiler for Lua.

The benchmarks we used come from the Computer Language Benchmarks Game [9] and Are We Fast Yet [17]. From the former we excluded three benchmarks: pydigits, regex-redux, and reverse-complement. The first two require external libraries that are not part of the Lua standard library. The latter is bottlenecked by the string library, which is implemented in C, therefore making it unsuitable for evaluating the performance of the interpreter.

We carried out the measurements on a laptop with an Intel i7-10510U CPU, running Fedora Linux 35. We used Lua version 5.4.3 and LuaJIT version 2.1.0-beta3. For the C compiler we used GCC version 11.2.1, with the -O2 optimization level. For each benchmark we picked an input size large enough to ensure that the fastest implementation took at least one second to run. We ran each benchmark 20 times.

The complete performance data is listed in in Table 1, which displays the average running time as well as the encountered variation. The error intervals refer to the difference between the average and the maximum or the minimum time, whichever was greater. In this table, the LuaAOT column is the default LuaAOT compiler; Trampoline refers to the compilation strategy without goto; Functions is the compilation strategy with one function call per instruction; Struct is a variant of the Function strategy without the separate functions, only refactoring the interpreter state into a struct (in order to measure the cost of that refactoring).

Before we get into the alternative compilation strategies, let's start by looking at the default LuaAOT compiler. In Figure 8 we compare the performance of LuaAOT against the reference Lua interpreter and LuaJIT, with the times being normalized by the average time of the reference interpreter. In all benchmarks, LuaAOT was faster than the Lua interpreter, but slower than LuaJIT. In the microbenchmarks, the reduction in running time compared to interpreted Lua ranged from approximately 20%, in the K-Nucleotide benchmark, to approximately 60%, in the Mandelbrot benchmark. Among the bigger benchmarks (CD, Deltablue, Havlak, JSON), there was a smaller time improvement in the order of 5 to 10%.

The speed of the trampoline version of LuaAOT fell between the speed of Lua and the speed of the default version of LuaAOT (using gotos). This is what we expected, because the main difference in the trampoline design is that it has a bit more dispatch overhead. Other than the dispatching, the generated code is the same as default LuaAOT.

In all benchmarks, the version that generates function calls for each instruction was slower than default LuaAOT. The worst case being the Mandelbrot benchmark with a 66% difference. Most of this can be attributed to the refactoring which moved the interpreter state from local variables to a struct object, which was necessary to be able to split the interpreter loop into multiple subroutines. We tested this by running a version of the compiler that only did that refactoring. As we can see in Figure 9, the performance was about the same as the version that also split into multiple subroutines. This suggests that the biggest culprit was having less efficient access to the interpreter state. One question that might be interesting for future research is whether we can avoid this problem by using a custom calling convention that preserves the important interpreter state in machine registers.

4.3 Pipelining

We were curious whether the better performance of LuaAOT compared to Lua was because it ran less CPU instructions or because it could run more instructions per second. To answer this question, we reran the benchmarks using Linux's `perf` tool, which can measure the number of CPU instructions and CPU cycles used by each program. The results are listed in Table 2. They suggest that, at least for this CPU model, the largest factor behind the improved speeds is a reduction in the number of CPU instructions. It appears that the instruction-per-cycle statistic is actually slightly worse for LuaAOT. For most benchmarks the reduction in instruction count is larger than the reduction in time (CPU cycles). We hypothesize that the biggest speedup comes from the compiler removing some of the instructions responsible for bytecode decoding and dispatching. As most of these instructions are cheap (e.g., shifts and masks for decoding), the reduction in the number of instructions is larger than the reduction in cycles, therefore reducing the instructions per cycle.

In theory, removing the bytecode dispatching (and the associated branches) has the potential to improve performance by avoiding costly branch mispredictions. However, in our benchmarks this was not a big factor, because the CPU already did a good job of predicting the branches in the interpreted version. In almost all benchmarks, the branch miss rates for both Lua and LuaAOT were

Benchmark	Lua	Trampoline	LuaAOT	Function	Struct	LuaJIT
Binary Trees	3.49 ± 0.06	3.08 ± 0.14	2.87 ± 0.07	3.15 ± 0.69	2.92 ± 0.11	1.24 ± 0.50
Fannkuch	43.97 ± 2.43	25.87 ± 2.76	21.79 ± 0.88	33.94 ± 2.40	32.90 ± 0.62	7.38 ± 0.25
Fasta	4.84 ± 0.11	4.05 ± 0.33	3.74 ± 0.17	4.46 ± 0.37	4.43 ± 0.09	1.24 ± 0.07
K-Nucleotide	3.91 ± 0.14	3.35 ± 0.10	3.21 ± 0.17	3.58 ± 0.14	3.64 ± 0.74	0.95 ± 0.07
Mandelbrot	14.27 ± 1.63	10.49 ± 0.71	6.80 ± 0.11	11.36 ± 1.14	10.94 ± 0.27	1.63 ± 0.05
N-Body	17.85 ± 1.45	12.26 ± 1.21	10.49 ± 0.36	14.15 ± 0.69	14.59 ± 0.80	1.10 ± 0.04
Spectral Norm	34.64 ± 1.12	28.00 ± 12.64	18.76 ± 0.56	28.28 ± 0.71	28.38 ± 2.27	1.21 ± 0.02
CD	1.89 ± 0.04	1.87 ± 0.10	1.68 ± 0.10	1.80 ± 0.06	1.79 ± 0.07	0.90 ± 0.15
Deltablue	2.08 ± 0.20	2.00 ± 0.20	1.89 ± 0.12	2.06 ± 0.09	2.09 ± 0.15	1.05 ± 0.17
Havlak	7.63 ± 0.32	7.41 ± 0.18	7.20 ± 0.12	7.59 ± 0.19	7.59 ± 0.15	4.36 ± 0.17
JSON	5.06 ± 0.11	4.89 ± 0.19	4.52 ± 0.42	4.88 ± 0.11	4.87 ± 0.15	1.03 ± 0.07
List	2.58 ± 0.11	2.01 ± 0.15	1.76 ± 0.54	2.23 ± 0.11	2.25 ± 0.12	1.02 ± 0.05
Permute	2.64 ± 0.08	2.00 ± 0.17	1.93 ± 0.15	2.58 ± 0.55	2.52 ± 0.19	0.09 ± 0.04
Richards	3.61 ± 0.13	3.09 ± 0.09	2.86 ± 0.12	3.34 ± 0.16	3.32 ± 0.15	0.89 ± 0.08
Hanoi Towers	4.16 ± 0.12	3.40 ± 0.15	3.22 ± 0.47	4.13 ± 0.38	4.11 ± 0.13	0.33 ± 0.09

Table 1: Running times, in seconds.

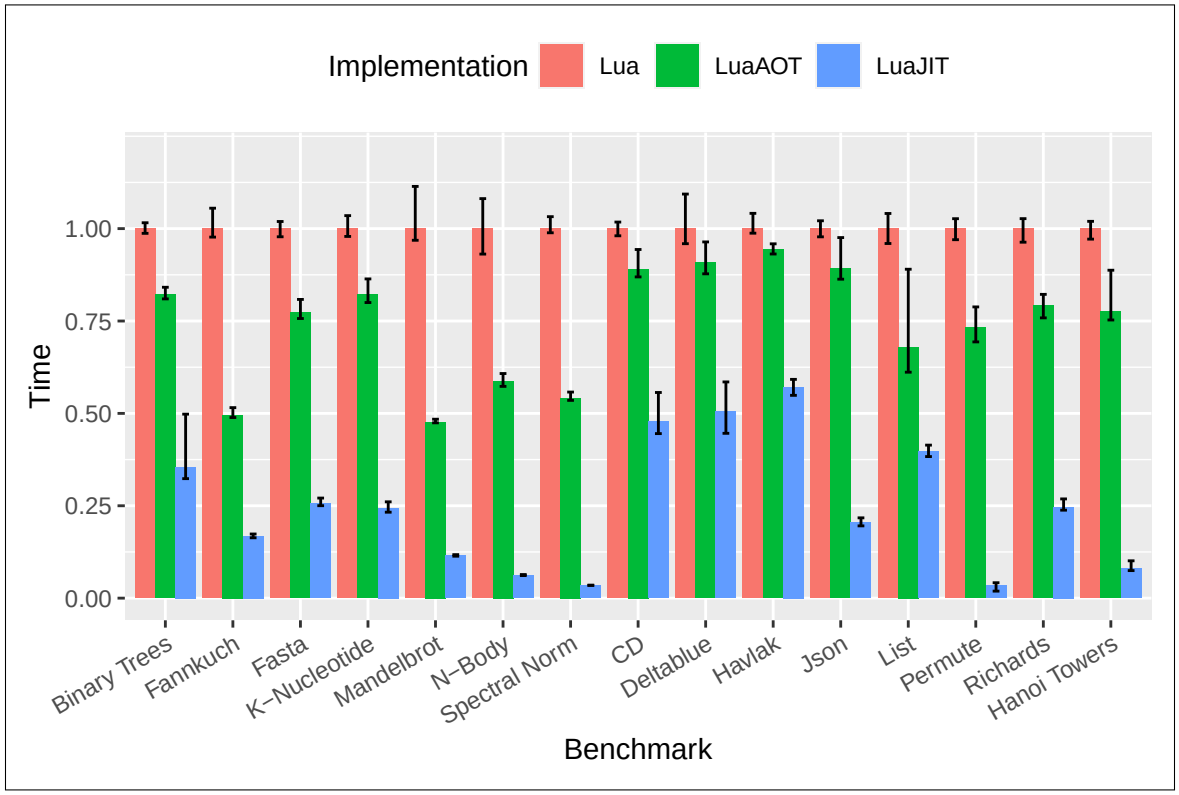

Figure 8: LuaAOT and LuaJIT running times, normalized by reference interpreter.

under 1%; the sole exception is the the N-Body benchmark for Lua, with a branch miss rate of 1.2%. With these low rates of branch miss, we do not think it is meaningful to compare the absolute numbers. It is also hard to tell whether the compilation is helping the branch-miss rates.

4.4 Code size

One trade-off of LuaAOT compared to the interpreter is that the generated binaries are larger than the corresponding bytecode. To evaluate this aspect of the compiler we measured the sizes of the Lua bytecode and of the AOT-compiled executables (stripped,

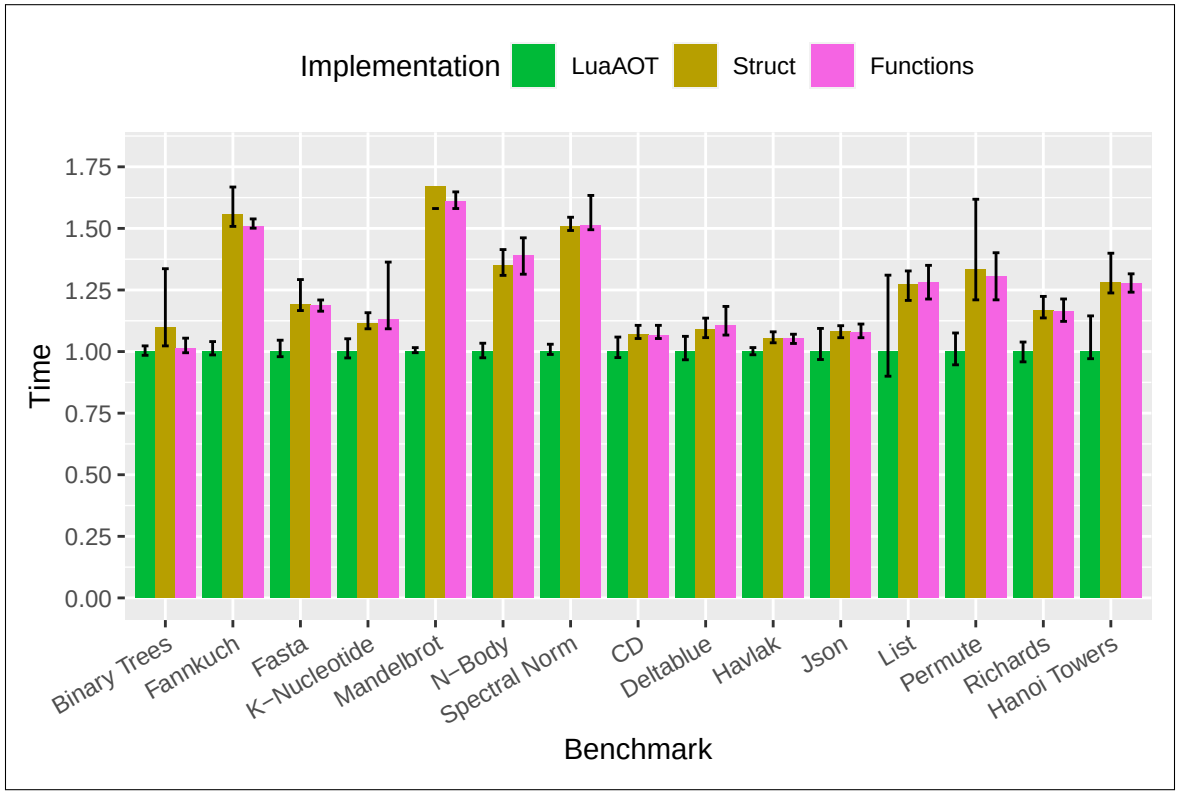

Figure 9: Running time for function call backend, normalized by reference interpreter.

Benchmark	Instrs (%)	Time (%)
Binary Trees	82.3	81.1
Fannkuch	44.3	50.4
Fasta	70.5	75.9
K-Nucleotide	75.8	81.9
Mandelbrot	32.7	46.5
N-Body	57.1	59.9
Spectral Norm	51.4	56.3
CD	81.5	89.1
Deltablue	83.6	91.1
Havlak	89.1	92.7
JSON	82.4	87.5
List	60.1	63.7
Permute	69.3	73.1
Richards	76.1	80.3
Hanoi Towers	72.1	80.8

Table 2: CPU instruction count for LuaAOT, relative to Lua.

Benchmark	Bytecode	AOT	FUN
Empty File	0.07	15	15
Binary Trees	1.4	35	39
Fannkuch	1.1	39	39
Fasta	3.0	71	71
K-Nucleotide	2.2	51	51
Mandelbrot	0.9	35	35
N-Body	3.2	83	67
Spectral Norm	1.5	39	39
CD	26.0	495	447
Deltablue	26.0	451	415
Havlak	24.0	411	379
JSON	49.0	435	407
List	13.0	243	215
Permute	12.0	231	207
Richards	22.0	371	343
Hanoi Towers	13.0	243	215

Table 3: Size of compiled modules, in KB.

without debug information). The results are listed in Table 3. In this table, the Bytecode column refers to interpreted Lua, the AOT column to the default compilation strategy with gotos, and FUN refers to the alternative compilation strategy with subroutines. We can see that the compiled executables are significantly larger than the corresponding Lua bytecode, albeit not so large that it becomes prohibitive.

For the larger benchmarks from Are We Fast Yet, the function-based compilation strategy did generate smaller executables as

intended. There was a size reduction of approximately 10% when compared to default LuaAOT. However, we did not observe this in the smaller benchmarks from the Benchmarks Game; in those cases the executable size is essentially the same. This might be because those programs have small bytecode size, meaning that the executable size is dominated by fixed overheads instead of by bytecode compilation results.

Due to the large size of the compiled executables, programmers may want to consider compiling only some of their modules. These decisions are commonplace for scripting languages, where we often write parts of the program in a compiled system language to achieve better performance. At the end of the day, compiling a Lua module with LuaAOT is a speed vs size tradeoff. On one axis we we have the number of bytecode instructions that are executed; modules with hot loops stand to benefit the most from compilation. On another axis we have the size of the files; larger Lua programs lead to larger executables.

4.5 Compilation times

In addition to reducing executable sizes, the alternative function-based compilation strategy can also reduce compilation times, when compared to the default goto-based compilation strategy. These numbers can be found in Table 4. Most benchmarks had a reduction of compilation times of over 15% and for the larger benchmarks the reduction was around 20% to 25%.

Although compilation times are not as important for ahead-of-time compilers as they are for just-in-time compilers, long compilation times can add friction to the development process. And since LuaAOT generates fairly large executables, the compilation times are quite noticeable in practice.

Our compilation time experiments suggest that the function-based compilation strategy has a bigger impact for larger programs. Out of curiosity, we tested what happens to the largest Lua file we could get our hands on: the Teal compiler [19]. It consists of a single file with over 9500 lines of Lua code. Compiling it with LuaAOT at -O2 optimization level took a leisurely 430 seconds. The function-based compilation strategy brought that down to 160 seconds, which is less than half.

4.6 Complexity of the implementation

A selling point of the partial-evaluation strategy that we used is its extreme simplicity. To measure this, we counted the lines of code of our code generator, as a proxy for implementation complexity.

We built the code generator by hand, using generous amounts of copy-pasting of code from the Lua interpreter loop and from the Lua bytecode compiler. Because we copied some subroutines from the Lua bytecode compiler, we chose to write the code generator in C. Out of the total of 1600 lines of code in the generator, 450 lines can be attributed to those subroutines from the bytecode compiler, which are responsible for traversing and printing bytecodes. Code templates derived from the core interpreter loop account for over half of the code generator, about 850 lines. The rest of the code, which we wrote from scratch, fits in less than 500 lines. It consists of miscellaneous things such as comments, command-line option handling, and the initialization routines for the generated extension modules.

Benchmark	AOT	FUN	Δ
Binary Trees	1.92	1.98	+3%
Fannkuch	2.24	1.97	-12%
Fasta	4.26	3.90	-8%
K-Nucleotide	2.53	2.52	0%
Mandelbrot	1.89	1.44	-24%
N-Body	5.84	3.22	-45%
Spectral Norm	1.61	2.00	+24%
CD	38.84	29.53	-24%
Deltablue	32.32	25.93	-20%
Havlak	31.05	24.24	-22%
JSON	27.64	24.93	-10%
List	15.20	12.01	-21%
Permute	14.08	11.88	-16%
Richards	20.15	20.34	+1%
Hanoi Towers	14.93	12.12	-19%

Table 4: Compilation time of AOT executables, in seconds

For comparison, the reference Lua interpreter contains 28 thousand lines of C code [14] and the LuaJIT just-in-time compiler has 80 thousand lines of C and 35 thousand lines of platform-specific assembly language [21]. Another thing that we can point out is that while LuaAOT required us to be familiar with the internals of the Lua interpreter, the final product did not require complex analysis algorithms or optimization passes.

We also looked at the implementation complexity of the alternative compilation strategies that we described: the trampoline strategy and the function-based strategy. The implementation for the trampoline version is about the same size as default LuaAOT, the main difference being that the code templates required less manual tweaking. The function-based implementation, on the other hand, required more manual tweaking. Namely, we had to refactor local variables into struct fields and create the helper functions (which required manually choosing the proper argument types and return values for each function).

One limitation of our manual process for creating LuaAOT is that we must repeat it if we want to update LuaAOT to a new version of the upstream Lua. In theory it ought to be possible to automate at least some of this work. Create text manipulation scripts that go through the Lua codebase, copy over the opcode handlers from the interpreter loop, put them inside printf calls, substitute goto statements in place of the program counter assignments, etc. The main catch is that without collaboration from the upstream interpreter, it is hard to guarante that these text manipulation scripts will still work in a future version of the interpreter. It is likely that at least some degree of manual intervention would still be necessary. Exploring this sort of automation might be an interesting avenue for future work.

4.7 Applicability of the technique

While the work we have presented is specific to the reference Lua interpreter, we think that the technique is simple enough to be applicable to other dynamic language interpreters. In this section, we discuss what were the aspects of Lua that our interpreter relied

on, and what conditions are necessary to apply this to another interpreter. Of course, the performance improvements will depend on the specifics of the interpreter, in particular what percentage of time is attributable to the core interpreter loop.

The first important thing is that our technique would not work as easily for AST-based interpreters. While it is also possible to use partial evaluation for an AST-based interpreter, it is more complicated than for a bytecode-based one because of the frequent presence of recursion in the main interpreter loop.

Since our technique is based on partial evaluation, the language used to implement the original interpreter is important. C, which is a popular language for writing interpreters, worked well for several reasons: the presence of a goto statement, the availability of optimizing compilers, and the existence of preprocessor macros.

When we compile jump instructions in the bytecode, we want a similar jump operation in our target language. In C we can use goto statements for this purpose, provided that the control flow in the original interpreter is all inside a single interpreter function. If the target language does not have goto statements, it is harder to compile the unstructured jumps in the bytecode.

Using C as the target language allowed us to take advantage of several optimizations from the C compiler, including constant propagation for the bytecode instructions. The partial evaluator has the luxury of emitting code that is almost identical to the code used by the original interpreter. This would be harder to do if, for example, the original interpreter were implemented in hand-written assembly language. In that case we would likely have to implement the constant propagation ourselves.

Albeit not a fundamental requirement, the C preprocessor was a convenient feature. The LuaAOT code generator is essentially a text-based code transformer and in that context it helps to have a text-based macro system built into the target language. Unlike inline functions, macros can jump to other parts of the program and assign to surrounding local variables.

To summarize, we believe that our technique is a good fit for languages that are implemented by a bytecode interpreter written in C or C++. Some popular scripting languages that fit this criteria include Python, Ruby, JavaScript, Perl, and PHP.

5 RELATED WORK

Although our work is inspired by partial evaluation, it is not a partial evaluation system. There is a rich literature on these partial evaluation systems and their application to interpreters [1, 16]. However, one difference between our work and these partial evaluation systems is that they usually require that the input interpreter must be in some specific format that the partial evaluator can work with. LuaAOT is a case study in doing this partial evaluation in an ad-hoc manner, on an existing interpreter.

A relevant example of partial evaluation for interpreters is the Truffle framework [23, 24]. Truffle allows a language implementer to create an efficient just-in-time compiler based on an AST interpreter. The implementer can write the AST interpreter and provide hints that tell Truffle which run-time type information should be collected and where to use the partial evaluation. As we just mentioned, one important difference compared to our work is that Truffle requires that the interpreter be written in Java, using the Truffle framework.

Another area we touch is the study of interpretation overhead. One way that this has been studied is by profiling the interpreter while it is running, to measure how much of the execution time can be assigned to bytecode decoding and dispatching [18]. However, if the motivation for the question is to compare the performance of an interpreter versus a compiler, it is useful to measure the result of an actual compiler. In addition to removing the decoding and dispatching, the compiler can also perform optimizations that are natural to implement in a compiler. One such compiler is Barany's pylibjit [2]. Barany implemented a just-in-time compiler for Python using the GNU LibJIT [22] library. Similarly to LuaAOT, his compiler also works at the bytecode level, converting each Python bytecode instruction into a machine code sequence. Barany measured the effect of enabling and disabling various optimizations passes of his compiler, to estimate how much of an impact these aspects have in the performance of Python programs [3]. Some of the optimizations that he implemented were removal of redundant reference counting, static dispatch of arithmetic operations, unboxing of number and container objects, and call-stack frame removal. These Python-specific optimizations allowed Barany to investigate the performance impact of more features of the interpreter other than just the bytecode handling. However, his compiler is more complex than ours; he had to reimplement most of the bytecodes using the LibJIT framework.

In addition to the basic form of LuaAOT, this paper also presented two alternative compilation strategies. The trampoline-based strategy is based on a novel idea. The function-based strategy is a trick that has been rediscovered by many compiler writers (for instance, PHC [4] and likely several others). In the context of LuaAOT, one of the most interesting aspects of the function-based strategy is that it shines a light at the code size and compilation times.

Finally, we want to mention that our work does not consider optimizations of the interpreter itself, including superinstructions or threaded dispatch [7]. We also did not study the effect of type inference. Our focus was in reducing the direct interpretation overhead.

6 THREATS TO VALIDITY

The benchmarks we used for this paper are microbenchmarks, each designed to take at least one second to run when executed by the Lua interpreter. It is conceivable that the performance results could be different for larger programs or programs that are not as computationally intensive as our benchmarks.

Another thing to consider is this work evaluates a single compiler, in the context of Lua. While we believe these techniques should also applicable to other languages, that has not been evaluated yet.

7 CONCLUSION

Dynamic programming languages are often implemented using interpreters, which spend some portion of the running time on interpretation overhead. Compilers can avoid this, but they may be complex to implement.

In this paper we have presented LuaAOT, a simple ahead-of-time compiler for Lua. Using less than 500 lines of new code in a total of 1600 lines, we were able to compile the entirety of Lua, including features such as coroutines and tail calls.

In return, we achieved a reduction in running times between 20% and 60%. While these numbers cannot compete head-to-head with a good JIT compiler, they demonstrate a noticeable performance boost, for a tiny fraction of the implementation cost. These numbers also offer a contribution to studies about interpretation overheads.

One novelty presented in this paper was the trampoline-based compilation strategy, which does not require modifications to the jump instructions. While it is not as fast as the goto-based strategy, it is even simpler to implement. It also supports coroutines with very little work. That is something that is tricky to do when compiling to C, because C itself does not support coroutines.

We believe that the technique we used to implement LuaAOT may be of interest to other programming languages. In particular, the basic ideas of our work may also be applicable to other interpreters that use a bytecode-based virtual machine written in C.

ACKNOWLEDGMENTS

This work has been funded in part by CNPQ, grants 153918/2015-2 and 305001/2017-5; and by CAPES, grant number 001.

A EXAMPLE OF GENERATED CODE

In Section 2, our examples featured a simplified interpreter. In this appendix, we show some real code produced by LuaAOT. It is the result of compiling the `foo` function from Figure 1. To make it fit, we made minor stylistic edits and replaced some sections by `/*...*/` comments. The code starts with some boilerplate initialization code, followed by the coroutine dispatch table, and finally the handlers for each bytecode instruction. In this code, we can see some of the preprocessor macro tricks. The `vmfetch` macro initializes the `i` variable to a compile-time constant, allowing the C compiler to constant fold the `GETARG_sBx` macro. The `LUAOT_SKIP1` macro allows instructions like `op_arith` to skip over the following MMBIN instruction; in the original interpreter, they increment the program counter (`pc++`) while in LuaAOT we replace that by `goto LUAOT_SKIP1`.

```
static void execute_foo(lua_State *L, CallInfo *ci)
{
  int trap = L->hookmask;
  LClosure *cl = clLvalue(s2v(ci->func));
  TValue *k = cl->p->k;
  const Instruction *pc = ci->u.l.savedpc;
  if (trap) { /*...*/ }
  StkId base = ci->func + 1;

  Instruction *code = cl->p->code;
  Instruction i;
  StkId ra;

  switch (pc - code) {
    case 0: goto label_00;
    /*...*/
    case 6: goto label_06;
  }

  // 0 - LOADI 3 17
  #undef  LUAOT_SKIP1
  #define LUAOT_SKIP1 label_02
  label_00: {
    aot_vmfetch(0x80080181);
    lua_Integer b = GETARG_sBx(i);
    setivalue(s2v(ra), b);
  }

  // 1 - ADD 0 1 2
  #undef  LUAOT_SKIP1
  #define LUAOT_SKIP1 label_03
  label_01: {
    aot_vmfetch(0x02010022);
    op_arith(L, l_addi, luai_numadd);
  }

  /*...*/
  // 6 - RETURN0
  label_06: {
    aot_vmfetch(0x00010247);
    if (L->hookmask) {
      L->top = ra;
      savepc(ci);
      luaD_poscall(L, ci, 0);
      trap = 1;
    } else {
      L->ci = ci->previous;
      L->top = base - 1;
      for (int nres = ci->nresults; nres > 0; nres--)
        setnilvalue(s2v(L->top++));
    }
    return;
}
```

Figure 10: The actual code generated by the compiler.

REFERENCES

[1] Lars Ole Andersen. 1992. Partial evaluation of C and automatic compiler generation. In *Compiler Construction*, Uwe Kastens and Peter Pfahler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 251–257.

[2] Gergö Barany. 2014. pylibjit: A JIT Compiler Library for Python.. In *Software Engineering (Workshops)*. 213–224.

[3] Gergö Barany. 2014. Python Interpreter Performance Deconstructed. In *Proceedings of the Workshop on Dynamic Languages and Applications (Dyla'14)*. 5:1–5:9. https://doi.org/10.1145/2617548.2617552

[4] Paul Biggar. 2010. *Design and Implementation of an Ahead-of-Time Compiler for PHP*. phdthesis. Trinity College Dublin. https://paulbiggar.com/research/#phd-dissertation

[5] Ana Lúcia de Moura. 2004. *Revisitando co-rotinas*. Ph.D. Dissertation. PUC-Rio.

[6] Ana Lúcia de Moura, Noemi Rodriguez, and Roberto Ierusalimschy. 2004. Coroutines in Lua. *Journal Universal Computer Science* 10, 7 (July 2004), 910–925. https://doi.org/10.3217/jucs-010-07-0910

[7] Anton M. Ertl and David Gregg. 2003. The Structure and Performance of Efficient Interpreters. *Journal of Instruction-Level Parallelism* 5 (Nov. 2003). https://www.jilp.org/vol5/index.html

[8] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391. https://doi.org/10.1023/A:1010095604496

[9] Isaac Gouy. 2013. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame

[10] Hugo Gualandi. 2021. LuaAOT 5.4 source code repository. https://github.com/hugomg/lua-aot-5.4

[11] Hugo Gualandi and Roberto Ierusalimschy. 2020. Pallene: A companion language for Lua. *Science of Computer Programming* 189 (2020), 102393. https://doi.org/10.1016/j.scico.2020.102393

[12] Hugo Musso Gualandi and Roberto Ierusalimschy. 2021. A Surprisingly Simple Lua Compiler. In *Proceedings of the 25th Brazilian Symposium on Programming*

Languages, SBLP 2021, Joinville, Brazil, September 27-October 1, 2021. https://doi.org/10.1145/3475061.3475077

[13] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2005. The Implementation of Lua 5.0. *Journal Universal Computer Science* 11, 7 (July 2005), 1159–1176. https://doi.org/10.3217/jucs-011-07-1159

[14] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2021. Lua 5.4 interpreter source code. https://www.lua.org/versions.html

[15] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2022. Lua test suite. https://www.lua.org/tests/

[16] Neil D. Jones. 2004. Transformation by interpreter specialisation. *Science of Computer Programming* 52, 1 (2004), 307–339. https://doi.org/10.1016/j.scico.2004.03.010 Special Issue on Program Transformation.

[17] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-language Compiler Benchmarking: Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages* (Amsterdam, Netherlands) *(DLS 2016)*. 120–131. https://doi.org/10.1145/2989225.2989232

[18] Nagy Mostafa, Chandra Krintz, Calin Cascaval, David Edelson, Priya Nagpurkar, and Peng Wu. 2010. *Understanding the Potential of Interpreter-based Optimizations for Python.* Technical Report. University of California, Santa Barbara. https://www.cs.ucsb.edu/sites/cs.ucsb.edu/files/docs/reports/2010-14.pdf

[19] Hisham Muhamma. 2019. The Teal Compiler. Teal source code repository. https://github.com/teal-language/tl/

[20] Peter Sestoft Neil D. Jones, Carsten K. Gomard. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice Hall.

[21] Mike Pall. 2005. LuaJIT, a Just-In-Time Compiler for Lua. http://luajit.org/luajit.html http://luajit.org/luajit.html.

[22] Rhys Weatherley. 2004. GNU LibJIT. The GNU LibJIT library.

[23] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, 662–676. https://doi.org/10.1145/3062341.3062381

[24] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. 187–204. https://doi.org/10.1145/2509578.2509581