「こんなきれいな星も、やっぱりここまで来てから、見れたのだと思うから。だから・・もっと遠くへ・・」

Building the fastest Lua interpreter.. automatically!

2022-11-22 ()

> This is Part 1 of a series of posts.
> Part 2 is available here: Building a baseline JIT for Lua automatically (/2023/05/12/2023-05-12/)

It is well-known that writing a good VM for a dynamic language is never an easy job. High-performance interpreters, such as the JavaScript interpreter in Safari (https://webkit.org/blog/10308/speculation-in-javascriptcore/), or the Lua interpreter in LuaJIT (http://lua-users.org/lists/lua-l/2011-02/msg00742.html), are often hand-coded in assembly. If you want a JIT compiler for better performance, well, you've got some more assembly to write. And if you want the best possible performance with multiple-tier JIT compilation… Well, that's assembly all the way down.

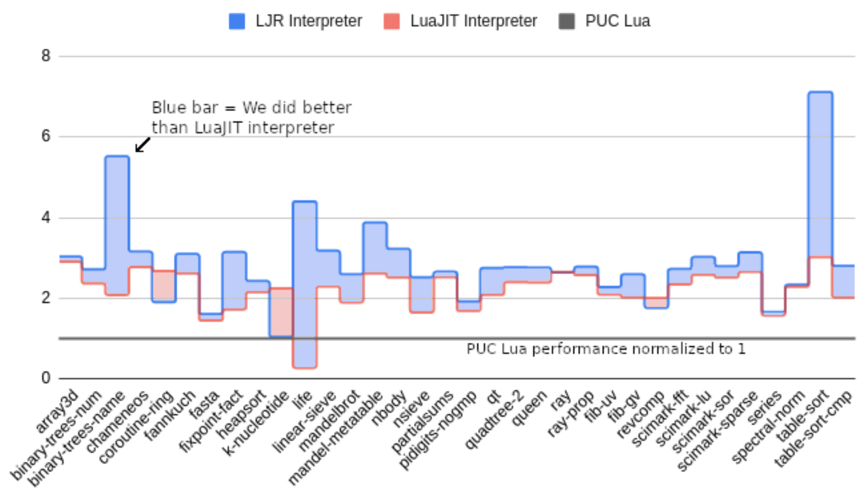
I have been working on a research project to make writing VMs easier. The idea arises from the following observation: writing a naive interpreter is not hard (just write a big switch-case), but writing a good interpreter (or JIT compiler) is hard, as it unavoidably involves hand-coding assembly. So why can't we implement a special compiler to automatically *generate* a high-performance interpreter (and even the JIT) from "the big switch-case", or more formally, a semantical description of what each bytecode does?

The LuaJIT Remake Project

I chose Lua (https://www.lua.org/) as the experiment target for my idea, mainly because Lua is concise yet supports almost every language feature one can find in dynamic languages, including exotic ones like stackful coroutines. I named my project LuaJIT Remake (https://github.com/luajit-remake/luajit-remake) (LJR) because in the long term, it will be a multi-tier method-based JIT compiler for Lua.

After months of work on the project, I've finally got some early results to share. LJR now has a feature-complete Lua 5.1 interpreter that is automatically generated at build time using a meta-compiler called `Deegen` (for "Dynamic language Execution Engine Generator"). More importantly, it is the world's fastest Lua interpreter to date, outperforming LuaJIT's interpreter by 28% and the official Lua interpreter by 171% on average on a variety of benchmarks[1].

The figure below illustrates the performance of our interpreter, the LuaJIT interpreter, and the official PUC Lua interpreter. PUC Lua's performance is normalized to 1 as a baseline.



LJR interpreter vs LuaJIT interpreter performance, normalized against PUC Lua (higher is better)

(https://sillycross.github.io/images/2022-11-22/interpreter-perf-comparison-2.png)

As the figure shows, our interpreter performs better than LuaJIT's hand-coded-in-assembly interpreter on 31 out of the 34 benchmarks[2], and on geometric average, we run 28% faster than LuaJIT interpreter, and almost 3x the speed of official PUC Lua.

Enough of the numbers, now I will dive a bit into how my approach works.

Why Assembly After All?

To explain how I built the fastest Lua interpreter, one needs to understand why (previously) the best interpreters have been hand-coded in assembly. This section is all about background. If you are already familiar with interpreters, feel free to skip to the next section.

Mike Pall, the author of LuaJIT, has explained this matter clearly in this great email thread (http://lua-users.org/lists/lua-l/2011-02/msg00742.html) back in 2011. The problem with the "big switch-case" approach is that C/C++ compilers simply cannot handle such code well. Although eleven years have passed, the situtation didn't change much. Based on my experience, even if a function only has one fast path and one cold path, and the cold path has been nicely annotated with `unlikely`, LLVM backend will still pour a bunch of unnecessary register moves and stack spills into the fast path[3]. And for the "big switch-case" interpreter loop with hundreds of fast-paths and cold-paths, it's unsurprising that compilers fail to work well.

Tail call (https://en.wikipedia.org/wiki/Tail_call), also known as continuation-passing style (https://dl.acm.org/doi/10.1145/800179.810196), is an alternative to switch-case-based interpreter loop. Basically each bytecode gets its own function that does the job, and when the job is done, control is transferred to the next function via a tail call dispatch (i.e., a jump instruction at machine code level). So despite that conceptually, the bytecode functions are calling each other, they are really jumping to each other at machine code level, and there will be no unbounded stack growth. An alternate way to look at it is that each "case" clause in the switch-case interpreter loop becomes a function. The "switch" will jump (i.e., tail call) to the corresponding "case" clause, and at the end of the case a jump (i.e., tail call) is executed to jump back to the switch dispatcher[4].

With the tail-call approach, each bytecode now gets its own function, and the pathological case for the C/C++ compiler is gone. And as shown by the experience (https://blog.reverberate.org/2021/04/21/musttail-efficient-interpreters.html) of the Google protobuf developers, the tail-call approach can indeed be used to build very good interpreters. But can it push to the limit of hand-written assembly interpreters? Unfortunately, the answer is still no, at least at its current state.

The main blockade to the tail-call approach is the callee-saved registers. Since each bytecode function is still a function, it is required to abide to the calling convention, specifically, every callee-saved register must retain its old value at function exit. So if a bytecode function needs to use a callee-saved register, it needs to save the old value on the stack and restore it at the end[5]. The only way to solve this problem is to use a calling convention with no callee-saved registers. Unfortunately, Clang is (to-date) the only compiler that offers guaranteed-tail-call intrinsic (`[[clang::musttail]]` annotation), but it has no such user-exposed calling convention with no callee-saved registers. So you lose 6 (or 8, depending on cconv) of the 15 registers for no reason on x86-64, which is clearly bad.

Another blockade to the tail-call approach is, again, the calling convention. No unbounded stack growth is a requirement, but tricky problems can arise when the caller and callee function prototype does not match, and some parameters are being passed in the stack. So Clang makes the compromise and requires the caller and callee to have *identical* function prototypes if `musttail` is used. This is extremely annoying in practice once you have tried to write anything serious under such limitation (for POC purpose I had hand-written a naive Lua interpreter using `musttail`, so I have first-hand experience on how annoying it is).

Generating the Interpreter: Another Level Of Indirection Solves Everything

As you might have seen, the root of all the difficulties is that our tool (C/C++) is not ideal for the problem we want to solve. So what's the solution?

Of course, throwing the tool away and resort to sheer force (hand-coding assembly) is one solution, but doing so also results in high engineering cost. Can we do it more swiftly?

It is well-known (https://en.wikipedia.org/wiki/Fundamental_theorem_of_software_engineering) that all problems in computer science can be solved by another level of indirection. In our case, C/C++ is a very good tool to describe the semantics of each bytecode (i.e., what each bytecode should do), but C/C++ is not a good tool to write the most efficient interpreter. So what if we add one level of indirection: we write the bytecode semantical description in C++, *compile it to LLVM IR*, and feed the IR into a special-purpose compiler. The special-purpose compiler will take care of all the dirty work, doing proper transformation to the IR and finally generate a nice tail-call-based interpreter.

For example, at LLVM IR level, it is trivial to make a function use `GHC` calling convention (a convention with no callee-saved registers) and properly transform the function to unify all the function prototype, thus solving the two major problems with `musttail` tail calls that is unsolvable at C/C++ level. In fact, `Deegen` (our meta-compiler that generates the interpreter) does a *lot* more than producing the tail calls, which we will cover in the rest of this post.

Hide All the Ugliness Behind Nice APIs

In Deegen framework, the semantics of each bytecode is described by a C++ function. One of the most important design philosophy of Deegen is to abstract away all the nasty parts of an interpreter. I will demonstrate with a simplified example for the `Add` bytecode:

```
void Add(TValue lhs, TValue rhs) {
  if (!lhs.Is<tDouble>() || !rhs.Is<tDouble>()) {
    ThrowError("Can't add!");
  } else {
    double res = lhs.As<tDouble>() + rhs.As<tDouble>();
    Return(TValue::Create<tDouble>(res));
  }
}
```

The function `Add` takes two boxed values (a value along with its type) `lhs` and `rhs` as input. It first checks if both `lhs` and `rhs` are `double` (the `Is<tDouble>()` check). If not, we throw out an error. Otherwise, we add them together by casting the two boxed value to its actual type (`double`) and do a normal `double` addition. Finally, we create a new boxed value of `double` type using `TValue::Create<tDouble>()` , return it as the result of the bytecode and dispatch to the next bytecode, through the `Return()` API call (note that this is not the C keyword `return`).

Notice how much nasty work we have abstracted away: decoding the bytecode, loading the operands and constants, throwing out errors, storing results to the stack frame, and dispatching to the next bytecode. All of these interpreter details either happen automatically, or happen with a simple API call (e.g., `ThrowError` or `Return`).

Now let's extend our `Add` to add support for the Lua `__add` metamethod semantics:

```
void AddContinuation(TValue /*lhs*/, TValue /*rhs*/) {
  Return(GetReturnValueAtOrd(0));
}
void Add(TValue lhs, TValue rhs) {
  if (!lhs.Is<tDouble>() || !rhs.Is<tDouble>()) {
    /* we want to call metamethod now */
    HeapPtr<FunctionObject> mm = GetMMForAdd(lhs, rhs);
    MakeCall(mm, lhs, rhs, AddContinuation);
    /* MakeCall never returns */
  } else {
    double res = lhs.As<tDouble>() + rhs.As<tDouble>();
    Return(TValue::Create<tDouble>(res));
  }
}
```

The `GetMMForAdd` is some arbitrary runtime function call that gets the metamethod. Deegen does not care about its implementation: the bytecode semantic description is just a normal C++ function, so it can do anything allowed by C++, of course including calling other C++ functions. The interesting part is the `MakeCall` API. It allows you to call other Lua functions with the specified parameters, and most importantly, a *return continuation*. The `MakeCall` API does not return. Instead, when the called function returns, control will be returned to the return continuation (the `AddContinuation` function). The return continuation function is similar to the bytecode function: it has access to all the bytecode operands, and additionally, it has access to all the values returned from the call. In our case, the semantics for Lua `__add` is to simply return the first value returned by the call as the result of the bytecode, so we use `GetReturnValueAtOrd(0)` to get that value, and use the `Return` API we have covered earlier to complete the `Add` bytecode and dispatch to the next bytecode.

Again, notice how much nasty work that we have abstracted away: all the details of creating the new Lua frame, adjusting the parameters and return values (overflowing arguments needs to go to variadic arg if callee accepts it, insufficient arguments need to get `nil`), transferring control to the callee functions, etc., are all hidden by a mere `MakeCall` API. Furthermore, all of these are language-neutral: if we were to target some other languages (e.g., Python), most of the Deegen code that implements the `MakeCall` could be reused.

The use of return continuation is designed to support Lua coroutines. Since Lua coroutines are stackful, and `yield` can happen anywhere (as `yield` is not a Lua keyword, but a library function), we need to make sure that the C stack is empty at any bytecode boundary, so we can simply tail call to the other continuation to accomplish a coroutine switch. This design also has a few advantages compared with PUC Lua's coroutine implementation:

1. We have no fragile `longjmp` s.
2. We can easily make any library function that calls into VM yieldable using this mechanism. In fact, the error message `cannot yield across C call frames` is gone completely in LJR: all Lua standard library functions, including exotic ones like `table.sort` , are redesigned to be yieldable using this mechanism.

Automation, Automation, and More Automation!

The bytecode semantic function specifies the execution semantics of the bytecode, but one still needs to specify the definition of the bytecode. For example, one needs to know that `AddVN` takes two operands where LHS is a bytecode slot and RHS is a number value in the constant table, and that `AddVN` returns one value, and that it always fallthroughs to the next bytecode and cannot branch to anywhere else. In Deegen, this is achieved by a *bytecode specification language*.

Again, let's use the `Add` as the example:

```
DEEGEN_DEFINE_BYTECODE(Add) {
  Operands(
    BytecodeSlotOrConstant("lhs"),
    BytecodeSlotOrConstant("rhs")
  );
  Result(BytecodeValue);
  Implementation(Add);
  Variant(
    Op("lhs").IsBytecodeSlot(),
    Op("rhs").IsBytecodeSlot()
  );
  Variant(
    Op("lhs").IsConstant<tDoubleNotNaN>(),
    Op("rhs").IsBytecodeSlot()
  );
  Variant(
    Op("lhs").IsBytecodeSlot(),
    Op("rhs").IsConstant<tDoubleNotNaN>()
  );
}
```

There are a few things going on here so we will go through them one by one. First of all, the `DEEGEN_DEFINE_BYTECODE` is a macro that tells us that you are defining a bytecode.

The `Operands(...)` API call tells us that the bytecode has two operands, with each can be either a bytecode slot (a slot in the call frame) or a constant in the constant table. Besides `BytecodeSlotOrConstant` , one can also use `Literal` to define literal operands, and `BytecodeRange` to define a range of bytecode values in the call frame.

The `Result(BytecodeValue)` API call tells us that the bytecode returns one value and does not branch. The enum key `BytecodeValue` means the bytecode returns one `TValue` . One can also use enum key `CondBr` to specify that the bytecode can branch, or just no argument to specify that the bytecode doesn't return anything.

The `Implementation(...)` API specifies the execution semantics of the bytecode, which is the `Add` function we just covered.

The interesting part is the `Variant` API calls. It allows one to create different variants of the bytecode. For example, in Lua, we have the `AddVV` bytecode to add two bytecode values, or the `AddVN` bytecode to add a bytecode value with a constant `double` , or the `ADDNV` bytecode to add a constant `double` with a bytecode value. In a traditional interpreter implementation, the implementation of all of these bytecodes must be written by hand, which is not only laborious, but also error prone. However, in Deegen's framework, all you need to do is to specify them as `Variant` s, and we will do all the work for you!

The `IsConstant` API allows optionally further specifying the type of the constant, as shown in the `IsConstant<tDoubleNotNaN>()` usage in the snippet. Deegen implemented special LLVM optimization pass to simplify the execution semantics function based on the known and speculated type information of the operands. For example, for the bytecode variant where `rhs` is marked as `IsConstant<tDoubleNotNaN>()` , Deegen will realize that the `rhs.Is<tDouble>()` check in the bytecode function must be `true` , and optimize it out. This allows us to automatically generate efficient specialized bytecode implementation, without adding engineering cost to the user. (And by the way, the `tDouble` and `tDoubleNotNaN` things, or more formally, the type lattice of the language, is also user-defined. Deegen is designed to be a generic meta-compiler: it is not welded to Lua).

Finally, Deegen will generate a user-friendly `CreateAdd` function for the user frontend parser to emit a `Add` bytecode. For example, the frontend parser can write the following code to generate an `Add` bytecode that adds bytecode slot `1` with constant `123.4` , and stores the output into slot `2` :

```
bytecodeBuilder.CreateAdd({
  .lhs = Local(1),
  .rhs = Cst<tDouble>(123.4),
  .output = Local(2)
});
```

The implementation of `CreateAdd` will automatically insert constants into the constant table, select the most suitable variant based in the input types (or throwing out an error if no satisfying variant can be found), and append the bytecode into the bytecode stream. The concrete layout of the bytecode in the bytecode stream is fully hidden from the user. This provides a maximally user-friendly and robust API for the user parser logic to build the bytecode stream.

This link (https://github.com/luajit-remake/luajit-remake/blob/f8fb972ec91c28b849bd263f164832f0ff434d1f/annotated/bytecodes/arithmetic_bytecodes.cpp) is the real implementation of all the Lua arithemtic bytecodes in LuaJIT Remake. It used a few features that we haven't covered yet: the `DEEGEN_DEFINE_BYTECODE_TEMPLATE` macro allows defining a template of bytecodes, so `Add` , `Sub` , `Mul` , etc., can all be defined at once, minimizing engineering cost. The `EnableHotColdSplitting` API allows automatically hot-cold-splitting based on speculated and proven input operand types, and splits out the cold path into a dedicated function, which improves the final code quality (recall the earlier discussion on the importance of hot-cold code splitting?).

And below is the actual disassembly of the interpreter generated by Deegen for Lua's `AddVV` bytecode. Comments are manually added by me for exposition purposes:

```
__deegen_interpreter_op_Add_0:
    # decode 'lhs' from bytecode stream
    movzwl      2(%r12), %eax
    # decode 'rhs' from bytecode stream
    movzwl      4(%r12), %ecx
    # load the bytecode value at slot 'lhs'
    movsd       (%rbp,%rax,8), %xmm1
    # load the bytecode value at slot 'rhs'
    movsd       (%rbp,%rcx,8), %xmm2
    # check if either value is NaN
    # Note that due to our boxing scheme,
    # non-double value will exhibit as NaN when viewed as double
    # so this checks if input has double NaN or non-double value
    ucomisd     %xmm2, %xmm1
    # branch if input has double NaN or non-double values
    jp          .LBB0_1
    # decode the destination slot from bytecode stream
    movzwl      6(%r12), %eax
    # execute the add
    addsd       %xmm2, %xmm1
    # store result to destination slot
    movsd       %xmm1, (%rbp,%rax,8)
    # decode next bytecode opcode
    movzwl      8(%r12), %eax
    # advance bytecode pointer to next bytecode
    addq        $8, %r12
    # load the interpreter function for next bytecode
    movq        __deegen_interpreter_dispatch_table(,%rax,8), %rax
    # dispatch to next bytecode
    jmpq        *%rax
.LBB0_1:
    # branch to automatically generated slowpath (omitted)
    jmp         __deegen_interpreter_op_Add_0_quickening_slowpath
```

As one can see, thanks to all of our optimizations, the quality of the assembly generated by Deegen has no problem rivalling hand-written assembly.

Inline Caching API: The Tricks of the Trade

A lot of LJR's speedup over LuaJIT interpreter comes from our support of inline caching. We have rewritten the Lua runtime from scratch. In LJR, table objects are not stored as a plain hash table with an array part. Instead, our table implementation employed hidden classes, using a design mostly mirroring the hidden class design in JavaScriptCore (https://webkit.org/blog/10308/speculation-in-javascriptcore/).

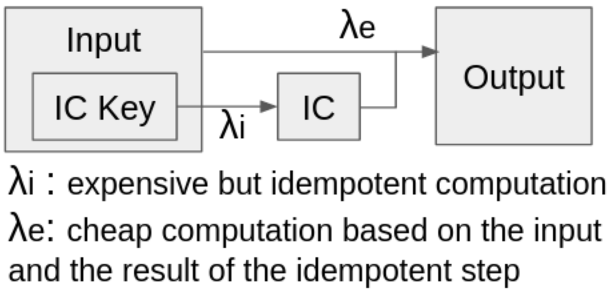
Hidden class allows efficient *inline caching*, a technique that drastically speeds up table operations. Briefly speaking, one can think of a hidden class as a hash-consed metadata object that describes the layout of a table object, or (simplified for the purpose of exposition), a hash map from string key to the storage slot in the table storing the value of this string key.

Let's use the `TableGetById` bytecode (aka, `TGETS` in LuaJIT) as example. `TableGetById` takes a table `T` and a fixed constant string `k` as input, and outputs `T[k]` .

Due to the natural use case of dynamic languages, for a fixed `TableGetById` bytecode, the tables it operates on are likely to have the same hidden class, or only a few different kinds of hidden classes. So `TableGetById` will cache the most recent hidden class `H` it saw, as well as `H[k]` , the storage slot in the table for the constant string key `k` . When `TableGetById` is executed on input `T` , it first check if the hidden class of `T` is just its cached hidden class `H` . If so (which is likely), it knows that the result must be stored in slot `H[k]` of `T` , so the expensive hash-lookup work (which queries hidden class `H` to obtain `H[k]`) can be elided.

In general, one can characterize the inline caching optimization as the following: there are some generic computation `λ : input -> output` that can be split into two steps:

1. An expensive but idempotent step `λ_i : icKey -> ic` where `icKey` is a subset of the `input` data, and `ic` is an opaque result.
2. A cheap but effectful step `λ_e : <input, ic> -> output`, that takes the `input` and the idempotent result `ic` for `input` in step 1, and outputs the final output.



Input
IC Key
λi
IC
λe
Output
λi : expensive but idempotent computation
λe: cheap computation based on the input and the result of the idempotent step

(https://sillycross.github.io/images/2022-11-22/ic.png)

Computation eligible for inline caching can be characterized as above.

If the computation satisfies such constraint, then one can cache `icKey` and the corresponding `ic`. Then on new inputs, if the `icKey` matches, the expensive idempotent step of computing `ic` can be safely elided.

Deegen provided *generic inline caching APIs* to allow easy employment of inline caching optimization. Specifically:

1. The full computation `λ` is specified as a C++ lambda (called the `body` lambda).
2. The effectful step `λ_e` is specified as C++ lambdas defined inside the `body` lambda (called the `effect` lambdas).

We allow specifying multiple possible `effect` lambdas in the `body` lambda, since the `λ_e` to execute can often be dependent on the outcome of the idempotent step. However, we require that at most one `effect` lambda can be executed in each run of the `body` lambda.

For example, for `TableGetById`, the code that employs inline caching would look like the following (simplified for the purpose of exposition):

```
void TableGetById(TValue tab, TValue key) {
  // Let's assume 'tab' is indeed a table for simplicity.
  HeapPtr<TableObject> t = tab.As<tTable>();
  // And we know 'key' must be string since the index of
  // TableGetById is required to be a constant string
  HeapPtr<String> k = key.As<tString>();
  // Call API to create an inline cache
  ICHandler* ic = MakeInlineCache();
  HiddenClassPtr hc = t.m_hiddenClass;
  // Make the IC cache on key 'hc'
  ic->Key(hc);
  // Specify the IC body (the function 'λ')
  Return(ic->Body([=] {
    // Query hidden class to get value slot in the table
    // This step is idempotent due to the design of hidden class
    int32_t slot = hc->Query(k);
    // Specify the effectful step (the function 'λ_e')
    if (slot == -1) {     // not found
      return ic->Effect([] { return NilValue(); }
    } else {
      return ic->Effect([=] { return t->storage[slot]; });
    }
  });
}
```

The precise semantic of the inline caching APIs is the following:

1. When `ic->Body()` executes for the first time, it will honestly execute the `body` lambda. However, during the execution, when a `ic->Effect` API call is executed, it will create an inline cache[6] for this bytecode that records the IC key (defined by the `ic->Key()` API), as well as all captures of this `effect` lambda that are *defined within* the `body` lambda. These variables are treated as constants (the `ic` state).
2. Next time the `ic->Body` executes, compare the cached key against the actual key.
3. If the key matches, it will directly execute the previously recorded `effect` lambda. For each capture of the `effect` lambda, if the capture is defined inside the `body` lambda, it will see the cached value recorded in step 1. Otherwise (i.e., the capture is defined as a capture of the `body` lambda), it will see the fresh value.
4. If the key does not match, just execute step 1.

The precise semantic might look a bit bewildering at first glance. A more intuitive way to understand is that one is only allowed to do idempotent computation inside the `body` lambda (idempotent is with respect to the cached key and other values known to be constants to this bytecode). All the non-idempotent computations must go to the `effect` lambda. As long as this rule is followed, Deegen will automatically generate correct implementation that employs the inline caching optimization.

Deegen also performs exotic optimizations that fuses the ordinal of the `effect` lambda into the opcode, to save an expensive indirect branch that branches to the correct `effect` implementation when the inline cache hits. Such optimizations would have required a lot of engineering efforts in a hand-written interpreter. But in Deegen, it is enabled by merely one line: `ic->FuseICIntoInterpreterOpcode()`.

Below is the actual disassembly of the interpreter generated by Deegen, for `TableGetById` bytecode. The assembly is for a "fused-IC" quickened variant (see above) where the table is known to have no metatable, and the property exists in the inline storage of the table. As before, comments are manually added by me for exposition purposes.

```
__deegen_interpreter_op_TableGetById_0_fused_ic_1:
    pushq         %rax
    # decode bytecode slot for the 'table' operand
    movzwl        2(%r12), %eax
    # decode bytecode slot for the 'index' operand
    movswq        4(%r12), %rcx
    # load the bytecode value of 'table'
    movq          (%rbp,%rax,8), %r9
    # load the constant value of 'index' (must be string)
    movq          (%rbx,%rcx,8), %rsi
    # check that 'table' is a heap object value (a pointer)
    cmpq          %r15, %r9
    # if not, branch to slow path (omitted)
    jbe           .LBB5_8
    # decode the destination slot from the bytecode
    movzwl        6(%r12), %r10d
    # load the metadata struct offset for this bytecode
    # the contents of the inline cache is stored there
    movl          8(%r12), %edi
    # compute the pointer to the metadata struct
    addq          %rbx, %rdi
    # load the hidden class of the heap object
    movl          %gs:(%r9), %ecx
    # check if hidden class matches
    cmpl          %ecx, (%rdi)
    # branch to code that calls the IC body if the hidden
    # class does not match (omitted)
    jne           .LBB5_4
    # The hidden class matched, we know the heap object is
    # a table without metatable, and the value for 'index'
    # is stored in the inline storage of the table, with the
    # slot ordinal recorded in the inline cache
    # Load the slot ordinal from the inline cache
    movslq        5(%rdi), %rax
    # load the value from the inline storage of the table
    movq          %gs:16(%r9,%rax,8), %rax
.LBB5_3:
    # store the value into the destination slot
    movq          %rax, (%rbp,%r10,8)
    # decode next bytecode and dispatch
    movzwl        12(%r12), %eax
    addq          $12, %r12
    movq        __deegen_interpreter_dispatch_table(,%rax,8), %rax
    popq          %rcx
    jmpq          *%rax
```

As one can see, in the good case of an IC hit, a `TableGetById` is executed with a mere 2 branches (one that checks the operand is a heap object, and one that checks the hidden class of the heap object matches the inline-cached value).

LuaJIT's hand-written assembly interpreter is highly optimized already. Our interpreter generated by Deegen is also highly optimized, and in many cases, slightly better-optimized than LuaJIT. However, the gain from those low-level optimizations are simply not enough to beat LuaJIT by a significant margin, especially on a modern CPU with very good instruction-level parallelism, where having a few more instructions, a few longer instructions, or even a few more L1-hitting loads have negligible impact on performance. The support of inline caching is one of the most important high-level optimizations we employed that contributes to our performance advantage over LuaJIT.

Conclusion Thoughts and Future Works

In this post, we demonstrated how we built the fastest interpreter for Lua (to date) through a novel meta-compiler framework.

However, automatically generating the fastest Lua interpreter is only the beginning of our story. LuaJIT Remake (https://github.com/luajit-remake/luajit-remake) is designed to be a multi-tier method-based JIT compiler generated by the Deegen framework, and we will generate the baseline JIT, the optimizing JIT, the tiering-up/OSR-exit logic, and even a fourth-tier heavyweight optimizing JIT in the future.

Finally, Deegen is never designed to be welded to Lua, and maybe in the very far future, we can employ Deegen to generate high-performance VMs at a low engineering cost for other languages as well.

Footnotes

1. The benchmarks are run on my laptop with Intel i7-12700H CPU and 32GB DDR4 memory. All benchmarks are repeated 5 times and the average performance is recorded. ↩
2. As a side note, two of the three benchmarks where we lost to LuaJIT are string processing benchmarks. LuaJIT seems to have some advanced string-handling strategy, yielding the speedup. However, the strategy is not perfect: it failed badly on the `life` benchmark, and as a result, LuaJIT got stomped 3.6x by PUC Lua (and 16x by us) on that benchmark. ↩
3. Some of these poor code come from insufficiently-optimized calling convention handling logic (e.g., LLVM often just pours all the spills at function entry for simplicity), and some comes from the register allocator that doesn't have enough understanding of hot/cold path (so that it believes that hoisting a register move or a stack spill from the cold path into the fast path is an optimization while it actually isn't). Compilers are always evolving and get better, but at least in this case it isn't enough. ↩
4. One can also imagine an optimization that makes each "case" directly jumps to the next "case", instead of the switch dispatcher. This is known as "direct threading (https://dl.acm.org/doi/abs/10.1145/362248.362270)" in the literature for continuation-passing-style-based interpreter, or more widely known as a "computed-goto interpreter" for switch-case-based interpreter (since GCC computed-goto extension is the most straightforward way to implement such an optimization). ↩
5. If one looks at the problem globally, clearly the better solution is to only save all the callee-saved registers once when one enters the interpreter, and restores it when the interpreter finishes, instead of having each bytecode function doing the same work again and again. But it's impossible to tell the C/C++ compiler that "this function doesn't need to abide to the calling convention, because by high-level design someone else will do the job for it". ↩
6. In the current implementation, for the interpreter, each bytecode is only allowed to keep one inline cache entry, so the newly-created entry always overwrites the existing entry. However, for JIT compilers, each inline cache entry will be a piece of JIT-generated code, so there can be multiple IC entries for each bytecode. ↩

‹ Pitfalls of using C++ Global Variable Constructor as a Registration Mechanism (../../../10/02/2022-10-02/)

Building a baseline JIT for Lua automatically (../../../../2023/05/12/2023-05-12/) ›

Archives

2023 (../../../../archives/2023/) (2)
2022 (../../../../archives/2022/) (9)
2021 (../../../../archives/2021/) (7)

Recents

Debugging a Bit-Flip Error (../../../../2023/06/11/2023-06-11/)

Building a baseline JIT for Lua automatically (../../../../2023/05/12/2023-05-12/)

Building the fastest Lua interpreter.. automatically! ()

Pitfalls of using C++ Global Variable Constructor as a Registration Mechanism (../../../10/02/2022-10-02/)