Later Binding: Just-in-Time Compilation of a Younger Dynamic Programming Language

Max Rottenkolber
max@mr.gy
Interstellar Ventures
Bonn, Germany

ABSTRACT

We examine LuaJIT, an implementation of the dynamic programming language Lua. By using a technique known as *tracing just-in-time compilation* LuaJIT is able to evaluate high-level language features with great efficiency. It does this by using only a conservative set of optimization passes, and without resorting to explicit type declarations, or abandoning type safety. In presenting the implementation's design we consider its strengths and weaknesses. Finally, we propose future directions for dynamic language implementations that wish to leverage this technique.

CCS CONCEPTS

• **Software and its engineering → Just-in-time compilers**; *Runtime environments*; **Interpreters**; *Dynamic compilers*; *Extensible languages*; *Functional languages*; *Object oriented languages.*

KEYWORDS

compilers, just-in-time, dynamic languages, object orientation, functional programming, late binding

ACM Reference Format:
Max Rottenkolber. 2020. Later Binding: Just-in-Time Compilation of a Younger Dynamic Programming Language. In *Proceedings of the 13th European Lisp Symposium (ELS'20).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.5281/zenodo.3743226

1 INTRODUCTION

Lua is a minimalist, dynamic programming language with Pascalesque syntax and Schemeish semantics. *LuaJIT* [Pall 2017] is an implementation of a Lua interpreter that uses tracing just-in-time compilation [Bala et al. 2000] to accelerate the evaluation of Lua programs.

A *just-in-time* (JIT) compiler intertwines run-time and compilation-time of the program to leverage run-time information to guide program optimization. Initially, the program is evaluated by a traditional interpreter program [Mccarthy 1960]. But soon enough the interpreter pulls off a magical trick. It considers the program it is evaluating while it is executing it on a given input, and speculatively compiles machine code[1] to perform the remainder of the evaluation on the remaining input.

This trick has interesting implications for the evaluation of dynamically typed, late binding programming languages such as Lua, Smalltalk, and Lisp.[2] In implementations of these languages there tends to be a lot of information about the program available at run-time. However, traditional *ahead-of-time* (AOT) compilers are in many cases not able to leverage this abundance of information to optimize emitted code. At AOT compile-time, information about the types of values and, by the extension, the specialization of methods may be overly conservative due to limits of inference.[3] The result is redundant dispatch on value types during evaluation.

2 MOTIVATING EXAMPLES: COLLAPSING ABSTRACTIONS

Consider this Common Lisp program which computes the sum of the integers $1..n < x$.

```
(defun sum (x)
  (loop for n from 1 to x sum n))
```

ELS'20, April 27–28 2020, Zürich, Switzerland

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.3743226

---

[1] Machine code here refers to a program for the *Instruction Set Architecture* (ISA) of the computer that runs our interpreter.

[2] Incidentally, Smalltalk hackers pioneered many aspects of modern JIT compilation.

[3] Here we consider implicit type information primarily, (optional) type declarations are a separate *can of worms.*

When compiled with Clozure Common Lisp on `x86_64` we can observe the following disassembly for the loop body.

```
L38
    (movq (% save0) (% arg_y))              ;    [45]
    (movq (% save2) (% arg_z))              ;    [48]
    (movl (% arg_y.l) (% imm0.l))           ;    [51]
    (orl (% arg_z.l) (% imm0.l))            ;    [53]
    (testb ($ 7) (% imm0.b))                ;    [55]
    (jne L63)                               ;    [58]
    (cmpq (% arg_z) (% arg_y))              ;    [60]
    (jle L90)                               ;    [63]
    (jmpq L204)                             ;    [65]
L63
    (lisp-call  (@ .SPBUILTIN-GT))          ;    [77]
    (recover-fn-from-rip)                   ;    [84]
    (cmpb ($ 11) (% arg_z.b))               ;    [91]
    (jne L204)                              ;    [95]
L90
    (movq (% save1) (% arg_y))              ;    [97]
    (movq (% save0) (% arg_z))              ;   [100]
    (movl (% arg_y.l) (% imm0.l))           ;   [103]
    (orl (% arg_z.l) (% imm0.l))            ;   [105]
    (testb ($ 7) (% imm0.b))                ;   [107]
    (jne L126)                              ;   [110]
    (addq (% arg_y) (% arg_z))              ;   [112]
    (jno L140)                              ;   [115]
    (lisp-call  (@ .SPFIX-OVERFLOW))        ;   [117]
    (recover-fn-from-rip)                   ;   [124]
    (jmp L140)                              ;   [131]
L126
    (lisp-call  (@ .SPBUILTIN-PLUS))        ;   [133]
    (recover-fn-from-rip)                   ;   [140]
L140
    (movq (% arg_z) (% save1))              ;   [147]
    (movq (% save0) (% arg_z))              ;   [150]
    (testb ($ 7) (% arg_z.b))               ;   [153]
    (jne L174)                              ;   [157]
    (addq ($ 8) (% arg_z))                  ;   [159]
    (jno L196)                              ;   [163]
    (lisp-call  (@ .SPFIX-OVERFLOW))        ;   [165]
    (recover-fn-from-rip)                   ;   [172]
    (jmp L196)                              ;   [179]
L174
    (movl ($ 8) (% arg_y.l))                ;   [181]
    (lisp-call  (@ .SPBUILTIN-PLUS))        ;   [189]
    (recover-fn-from-rip)                   ;   [196]
L196
    (movq (% arg_z) (% save0))              ;   [203]
    (jmpq L38)                              ;   [206]
```

In the disassembly of the compiled loop we quickly see a pattern. Run-time type checks ([55], [107], [153]) followed by a branch to either *fixnum*-specialized code ([60], [112], [159]) or generic run-time dispatch routines (`SPBUILTIN-GT`, `SPBUILTIN-PLUS`). Also notably, as characteristic for Lisp, arithmetic overflow is checked for ([115], [163]), and handled by type promotion (`SPFIX-OVERFLOW`).

Let us look at a similar program written in Lua, and the machine code emitted for the inner loop by LuaJIT.

```
function sum (x)
   local a = 0
   for n=1,x do a = a + n end
   return a
end
```

If the above function was called as `sum(100)` the following code will end up being executed.

```
->LOOP:
2a53ffe0  xorps xmm6, xmm6
2a53ffe3  cvtsi2sd xmm6, ebp
2a53ffe7  addsd xmm7, xmm6
2a53ffeb  add ebp, +0x01
2a53ffee  cmp ebp, eax
2a53fff0  jle 0x2a53ffe0 ->LOOP
```

Lua uses a single type for all numeric values—typically *double* floats. Hence the accumulator a is held in an SSE floating point register (*xmm*7). LuaJIT managed to infer that the type of the index variable i can be narrowed to a 32-bit integer, held in *ebp*, and converted to a double for arithmetic in *xmm*6. For the actual arithmetic, native integer and floating point addition instructions are emitted (*add*, *addsd*). Notably absent from the inner loop are any dispatches on value types.[4] Any guards needed to ensure correctness of the program—say, what if x is a string?—are hoisted before the loop.

The fundamental difference between the two compilers we just examined is *when* they emit code. Clozure Common Lisp compiles the function ahead of run-time, and emits one code path for all run-time cases possibly encountered during the lifetime of the function. LuaJIT on the other hand emits code at run-time, and only for code paths that are actually executed. Subsequently, emitted code is more narrowly specialized on particular evaluations of the program.

2.1 Object Orientation

Lua has no direct notion of *object oriented* programming. Instead, the built-in *setmetatable* allows programmers to overload the various built-in operators such as indexing (`.`, `:`) by setting the so-called "metatable" of an object. This mechanism lends itself to all sorts of meta-programming, and enables customizations of the language such as operator overloading, or prototype based object orientation.[5]

```
Acc = {}
```

```
function Acc:new ()
   return setmetatable({a=0}, {__index=Acc})
end
```

```
function Acc:sum (n)
   self.a = self.a + n
end
```

Again, with a similar program, we look at how LuaJIT compiles a different set of abstractions. Instantiating our accumulator class,

---

[4] Also absent is handling of arithmetic overflow. However, we argue that this is satisfyingly handled by the underlying hardware's implementation of IEEE double-precision floating point numbers.

[5] As a metatable can itself have a metatable installed, inhertiance comes quite naturally.

and calling its *sum* method in a loop causes the following loop body to be emitted.

```
local a = Acc:new()
for i=1,100 do a:sum(i) end
```

```
->LOOP:
2a53ffe0  xorps xmm6, xmm6
2a53ffe3  cvtsi2sd xmm6, ebp
2a53ffe7  addsd xmm7, xmm6
2a53ffeb  movsd [rax], xmm7
2a53ffef  add ebp, +0x01
2a53fff2  cmp ebp, +0x64
2a53fff5  jle 0x2a53ffe0 ->LOOP
```

To our satisfaction, the emitted code is almost unchanged. The only differences are the store of our accumulator (`2a53ffeb`) not being forwarded beyond the loop body, and the loop limit being emitted as a constant literal instead of being held in *eax*. Notably, the *sum* method has been inlined, hence there is no function call overhead.

2.2 Functional Abstractions & Polymorphism

Lua supports closures and higher-order functions. Let us try higher-order functions next, and add some gratuitously explicit polymorphism, too.

```
function make_acc ()
   local a
   return function (x)
      if x == nil then
         return a
      elseif type(x) == 'number' then
         a = (a or 0) + x
      elseif type(x) == 'string' then
         a = (a or "") .. x
      end
   end
end
```

For the last example, we create an accumulator closure. We want to see how LuaJIT inlines the closure into the emitted loop body code.

```
local acc = make_acc()
for i=1,100 do acc(i) end
```

```
->LOOP:
2a53ffd0  xorps xmm6, xmm6
2a53ffd3  cvtsi2sd xmm6, ebp
2a53ffd7  addsd xmm7, xmm6
2a53ffdb  movsd [0x41d741d0], xmm7
2a53ffe4  add ebp, +0x01
2a53ffe7  cmp ebp, +0x64
2a53ffea  jle 0x2a53ffd0 ->LOOP
```

The exact same code, again. Where did the branches go? *Dead code elimination* did its trick since LuaJIT could specialize the emitted code on numbers using run-time type information. Within the loop body, code paths for handling strings and *nil* were not emitted at all.

These few examples are intended to show the depths of abstraction that can be collapsed by means of JIT compilation, and to motivate the reader's interest in LuaJIT. In the following sections, we wish to shine a light on LuaJIT's design, and its limitations.

3 ARCHITECTURE AND IMPLICATIONS OF A TRACING JIT COMPILER

At its heart, LuaJIT is a bytecode interpreter. Embedded in this interpreter is a special-purpose run-time profiler. For certain branching bytecodes a table of "hot counts" is maintained. This table is indexed through a hash of the program counter.

Whenever the interpreter encounters one of the bytecodes to be tracked it increments its associated hot count. When incrementing a hot count causes it to overflow beyond a certain value the interpreter will begin recording a trace, starting from the next bytecode instruction.

```
0005  FORI     i=1,n
0006  MODVN    tmp1=i%2
0007  KSHORT   const1=0
0008  ISGE     const1>=tmp1
0009  JMP      if 0008 is true => 0011
0010  ADDVV    A=A+i
0011  FORL     i=i+1, i>n => 0006
```

The exemplary bytecodes above represent a *for* loop that sums odd integers 1..*n*. The `FORL` bytecode controls loop iteration and is tracked in a hot count for the program counter position 0011.

When the `FORL` bytecode becomes hot the interpreter begins recording the following instructions it executes until a trace stop condition is met. In this case, the trace stop condition will be triggered upon encountering the `FORL` bytecode at position 0011 for a second time, or on exiting the loop.

Assuming the loop exit condition is not met, the first bytecode to be executed—and recorded in the trace—will be the `MODVN` bytecode at position 0006, which calculates modulo 2 of *i*. The next bytecodes recorded are then 0007..0008 which load the constant zero, and check if *i* is odd—i.e., whether *i%2* is greater than zero. If *i* is odd at the time of recording then the following `JMP` bytecode will not be executed, and the remaining bytecodes to be recorded are `ADDVV` and the initial `FORL` that closes the loop, and causes the interpreter to stop recording with a successful trace (0006..0011).

This trace is then handed over to the JIT compiler, which translates the recorded bytecode instructions into a native program for the target instruction set, optimized using the information gathered during recording. [Gal et al. 2009] The interpreter then "patches" the `FORL` bytecode at 0007 by replacing it with a `JFORL` bytecode that causes the emitted code to be executed instead of the original bytecode.

During code generation the recorded bytecodes are translated into a SSA [Cytron et al. 1991] *immediate representation* (IR), and a number of optimizing transformations are performed:

- `FOLD`: A rule-based fold engine dispatches to later optimization stages, but also performs algebraic simplifications.
- `ABC`: Array Bounds Check Elimination.
- `CSE`: Common Sub-expression Elimination.
- `LOOP`: Loop invariant hoisting, and loop unrolling.
- `DCE`: Dead code elimination.
- `AA`: Alias Analysis.
- `FWD`: Load and store forwarding.

- `DSE`: Dead-Store Elimination.
- `NARROW`: Narrowing of numbers (doubles to 32-bit integers).
- `STRIPOV`: Stripping of overflow checks.
- `SINK`: Allocation Sinking and Store Sinking.

Eventually, execution of a compiled trace will exit and return to the interpreter. A compiled trace can exit for a number of reasons. One way to exit emitted code is by executing it successfuly to completion. Additionally, any deviation from the invariants encountered during trace recording will trigger an exit in the emitted code. Any branches in the recorded trace as well as any checks for invariants of the specialized code are converted to “guards” where each guard represents an invariant assertion and a dedicated exit point.

```
0002 >  tab SLOAD  #1    T
0003    p32 HREF   0002  "sum"
0004 >  p32 EQ     0003  [0x41526458]
```

In the SSA immediate representation of a trace above we can see two dependent guards, marked by > characters. The first guard at 0002 loads an object via `SLOAD`, and asserts that it is of type *table*. The second guard at 0004 ensures that the *sum* slot of the table contains the object at address `0x41526458`[6]. If either of these invariants is violated the compiled trace will abort execution, and exit to the interpreter.

The same is true for branches converted to guards during trace recording. In the IR below we can see a guard that exits the loop body if the index is odd.

```
     ------ LOOP ------------
0010    int BAND   0007  +1
0011 >  int GT     0010  +0
```

Two things should be said about trace exits. First, each guarded trace exit is tracked with a dedicated hot count, and repeatedly taken exits will cause a new trace to be recorded starting from the respective branch. In LuaJIT, traces starting from a trace exit form a distinct class of traces called “side traces”, and can not themselves record loops.

Second, exiting a compiled trace represents the end of a compilation unit, and requires consolidation between the interpreter state, and the state mutated by the emitted code. Mutations performed by emitted code are organized by LuaJIT in “snapshots”, and these snapshots need to be restored to the interpreter state upon trace exit. Likewise for transitions between compiled traces, side traces cannot return directly into a loop body of their parent, and force repeated execution of invariant guards.

3.1 Mechanical Sympathy

Trace selection in LuaJIT works analogous to a CPU branch predictor. While a modern computer speculatively executes certain branches, a tracing JIT compiler might speculatively compile, and hence bias, certain branches. Pitfalls of speculative execution apply equally to CPU branch predictors, and JIT engines. In LuaJIT specifically, the speculative aspects of the compiler are less mature than their hardware counterparts, and some pitfalls are present in exacerbated variants.

It is easy, to construct a Lua program, even unknowingly, that executes an unbiased branch in a loop which cannot be hoisted before the loop body. In LuaJIT's current implementation, and specifically under the limitations of the interaction between the trace as a compilation unit, trace exits, and exit snapshots, the emitted code can be unfavorable compared to traditional AOT compilers.

Furthermore, adversarial inputs can manipulate the outcomes of speculative execution. [Kocher et al. 2019] This is an inherent aspect of speculative execution in general, and deserves particular attention when designing JIT compilers.

Listed below is the machine code emitted by LuaJIT for our branchy loop from section 3. The first trace recorded covers the loop. The second trace begins at the fifth exit (`->5`) of trace #1, and covers a single iteration of the loop.

```
---- TRACE 1 mcode 100
2a53ff90  mov dword [0x41991410], 0x1
2a53ff9b  cvttsd2si ebp, [rdx+0x8]
2a53ffa0  test ebp, 0x1
2a53ffa6  jle 0x2a530014 ->1
2a53ffac  cmp dword [rdx+0x4], 0xfffeffff
2a53ffb3  jnb 0x2a530018 ->2
2a53ffb9  xorps xmm7, xmm7
2a53ffbc  cvtsi2sd xmm7, ebp
2a53ffc0  addsd xmm7, [rdx]
2a53ffc4  add ebp, +0x01
2a53ffc7  cmp ebp, +0x64
2a53ffca  jg 0x2a53001c ->3
->LOOP:
2a53ffd0  test ebp, 0x1
2a53ffd6  jle 0x2a530024 ->5
2a53ffdc  xorps xmm6, xmm6
2a53ffdf  cvtsi2sd xmm6, ebp
2a53ffe3  addsd xmm7, xmm6
2a53ffe7  add ebp, +0x01
2a53ffea  cmp ebp, +0x64
2a53ffed  jle 0x2a53ffd0 ->LOOP
2a53ffef  jmp 0x2a530028 ->6
---- TRACE 1 stop -> loop
```

```
---- TRACE 2 mcode 49
2a53ff58  mov dword [0x41991410], 0x2
2a53ff63  add ebp, +0x01
2a53ff66  cmp ebp, +0x64
2a53ff69  jg 0x2a530014 ->1
2a53ff6f  xorps xmm6, xmm6
2a53ff72  cvtsi2sd xmm6, ebp
2a53ff76  movsd [rdx+0x20], xmm6
2a53ff7b  movsd [rdx+0x8], xmm6
2a53ff80  movsd [rdx], xmm7
2a53ff84  jmp 0x2a53ff90
---- TRACE 2 stop -> 1
```

Note how in trace #1 the first loop iteration is unrolled, and invariant checks performed in the first iteration are not repeated in subsequent interations. Trace #2 performs a single iteration of the loop where *ebp* is even, and returns to the beginning of trace #1. Given the unbiased branch, every other iteration of the loop will exit trace #1, and likely cause a rentry at its top re-executing any

---

[6] I.e., the object must be the *sum* method from section 2.1

invariant guards. This compiler behavior effectively cancels out high-impact loop optimizations.

Another important observation is that the emitted code is dependent on which branch taken at the time of recording. Naturally, control flow is exercised by the input to the evaluation of the program. Situations arise in which for a heavily biased branch—more common in practice than unbiased branches—either favourable (as in trace #1) or unfavourable (as in trace #2) code is emitted depending on the input to the evaluation. The quality of generated code and, by extension, execution performance being affected by adversarial input is problematic.[7]

3.2 Virtual Machine Words

LuaJIT uses *NaN tagging* to represent doubles and other built-in types as single tagged 64-bit *virtual machine* (VM) words. [8] This representation allows the most common types of values to be stack-allocated without cooperation from the garbage collector (GC).

We have experience using LuaJIT for systems applications that handle many 64-bit values such as large integers and pointers that do not fit within a tagged VM word. This was made possible because, within emitted machine code, LuaJIT is able to *sink* allocations of objects, which otherwise must generally be heap-allocated, as long as they are held in registers.[9]

However, we found this optimization to be unreliable in situations where 64-bit values spill out of registers onto the stack, and subsequently cause GC pressure.

4 WHERE TO GO NEXT

LuaJIT is yet incomplete. Advancements in JIT compilation techniques, such as in better code generation for loops with unbiased branches could be incorporated in future implementations of JIT compilers. [Gal and Franz 2006]

Double-precision floating point numbers have become a popular base type for numbers in dynamic programming languages. Considering the advanced floating-point support of dominant ISAs, we would like to pose a question: rather than building a machine for their Lisp, should hackers build a Lisp for the best available machine?

If we look at the interpreter as a component of an optimizing compiler, rather than the primary execution engine itself, we might wish to choose nontraditional trade-offs. We might increase the VMs word size to fit the common 64-bit values we are having trouble with. [Soldatov and IPONWEB 2018] After all, the stack overhead of the interpreter is rendered mostly irrelevant in our emitted code.

A general design goal should be to find optimizing transformations with high-generality in order to provide reliable performance. Brittle performance is giving JITs a bad name as it is. To no lesser importance, future JIT compilers must ensure that adversarial program inputs can only control which code paths are to be compiled, but can never affect the quality of the emitted code.

With respect to Lisp, there are implementations such as *Armed Bear Common Lisp* and *Clojure* that have inherited big, mature JIT compilers. However, there is also *Guile* which recently added a new young JIT compiler. [Wingo 2020]

We hope to present JIT compilers as an exciting, young field. And in an ode to *Squeak*, we hope to garner interest in JIT compilation as a technique for iteratively writing small, beautiful, and fast dynamic language implementations. [Ingalls et al. 1997]

REFERENCES

Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A Transparent Dynamic Optimization System.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (Oct 1991), 451–490. http://doi.acm.org/10.1145/115372.115320

Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David M, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, and Mozilla Corporation. 2009. Trace-based Just-in-Time Type Specialization for Dynamic Languages.

Andreas Gal and Michael Franz. 2006. Incremental Dynamic Code Generation with Trace Trees.

Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, Alan Kay, and Walt Disney Imagineering. 1997. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *In Proceedings OOPSLA '97, ACM SIGPLAN Notices*. ACM Press, 318–326.

Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

John Mccarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I.

Mike Pall. 2017. The LuaJIT Project. https://luajit.org/

Anton Soldatov and IPONWEB. 2018. Rewriting LuaJIT: Why and How. https://www.lua.org/wshop18/Soldatov.pdf

Andy Wingo. 2011. value representation in javascript implementations. https://wingolog.org/archives/2011/05/18/value-representation-in-javascript-implementations

Andy Wingo. 2020. lessons learned from guile, the ancient & spry. http://wingolog.org/archives/2020/02/07/lessons-learned-from-guile-the-ancient-spry

---

[7] Adversarial need not imply malevolent; undeterministic performance depending on the workload handled within the first milliseconds of your application's run-time is frustrating, to say the least.

[8] *NaN tagging* or *NaN boxing* [Wingo 2011]

[9] Sinking here refers to avoiding *boxing* and *unboxing* of the object.