Exercise Sheet 12

VU Performance-oriented Computing, Summer Semester 2024

Calvin Hoy, Luca Rahm, Jannis Voigt

2024-06-18

A) Setup and Basic Execution

Benchmark results with default Lua interpreter compilation (-O2):

test	wall (mean)	wall (stddev)
iter	10.8730	0.036
naive	12.5483	0.322
tail	12.5980	0.046

B) Profiling

Profiling with Gprof

Setup

I first had to recompile the interpreter with the added `-pg` flag, i had some issues so i had to add `-g` and `-no-pie` aswell to get a good output.

Those flags were very easy to set in the Makefile in the `/src/` directory as there was a place for custom flags.

Results

gprof output

Figure 1: gprof output

From the flat profile we get by gprof we can see that 77.25% of the time is spent in the function `luaV_execute` that is therefore a function we should have a look at, here it's also interesting to note that we only have one call to this function, so it might also be something like a main.

Next in the ranking is the function `luaD_pretailcall` with 10.88% of the execution time, this function has a lot of calls on it (306,946,270), which means that even a small percentage improvement could have a big impact.

The same goes for `luaD_precall`.

The full profile can be found in `gprof_results.txt`.

Profiling with Tracy

The modified source code including `Makefile` can be found in the `lua_tracy/` directory. To enable Tracy, compile with `make EXTRACFLAGS=-DTRACY_ENABLE`. To additionally enable per-opcode zones (refer to the section "Opcode Statistics"), compile with `make EXTRACFLAGS=-DTRACY_ENABLE`.

The -g flag was added to the default compilation flags in order to make Tracy's source code view would work properly. This did not appear to significantly impact runtime, at least on my machine.

I added the ZoneScoped macro to the following functions:

- luaV_execute
- luaF_findupval
- luaF_closeupval
- luaH_getstr
- luaH_getshortstr
- luaC_step

Additionally, we added a FrameMark to the end of the interpreter loop in an attempt to measure the number of opcodes executed per second, although this is seemingly never hit.

Most of the runtime is spent, unsurprisingly, directly inside the main interpreter loop.

Which Opcodes?

Using the Source view in the Tracy Profiler application, we can narrow down the Lua Virtual Machine opcodes that are actually used by the fib.lua benchmark:

(This is also possible using e.g. perf report, although Tracy's UI is much more navigable.)

- OP_MOVE
- OP_LOADI
- OP_LOADK
- OP_GETUPVAL
- OP_GETTABUP
- OP_GETFIELD
- OP_SELF
- OP_ADDI
- OP_ADD
- OP_EQI
- OP_LTI
- OP_GEI
- OP_TESTSET
- OP_CALL
- OP_TAILCALL
- OP_RETURN1
- OP_FORLOOP
- OP_FORPREP
- OP_TFORLOOP
- OP_CLOSURE
- OP_VARARG
- OP_VARARGPREP
- OP_EXTRAARG

This represents a small subset of the opcodes available in the interpreter, giving a better idea of where optimizations should be made.

Opcode Statistics

To get an idea of which opcodes are executed how often, I added named zones to each opcode in the luaV_execute function as follows:

```
vmdispatch (GET_OPCODE(i)) {
    // [...]
    vmcase(OP_LTI) {
        TracyCZoneN(ctx, "OP_LTI", true);
        op_orderI(L, l_lti, Luai_numlt, 0, TM_LT);
        TracyCZoneEnd(ctx);
        vmbreak;
```

```
    }
    // [...]
}
```

Special care had to be taken around `goto` statements in the interpreter, particularly around `OP_RETURN1` and the `ret` label.

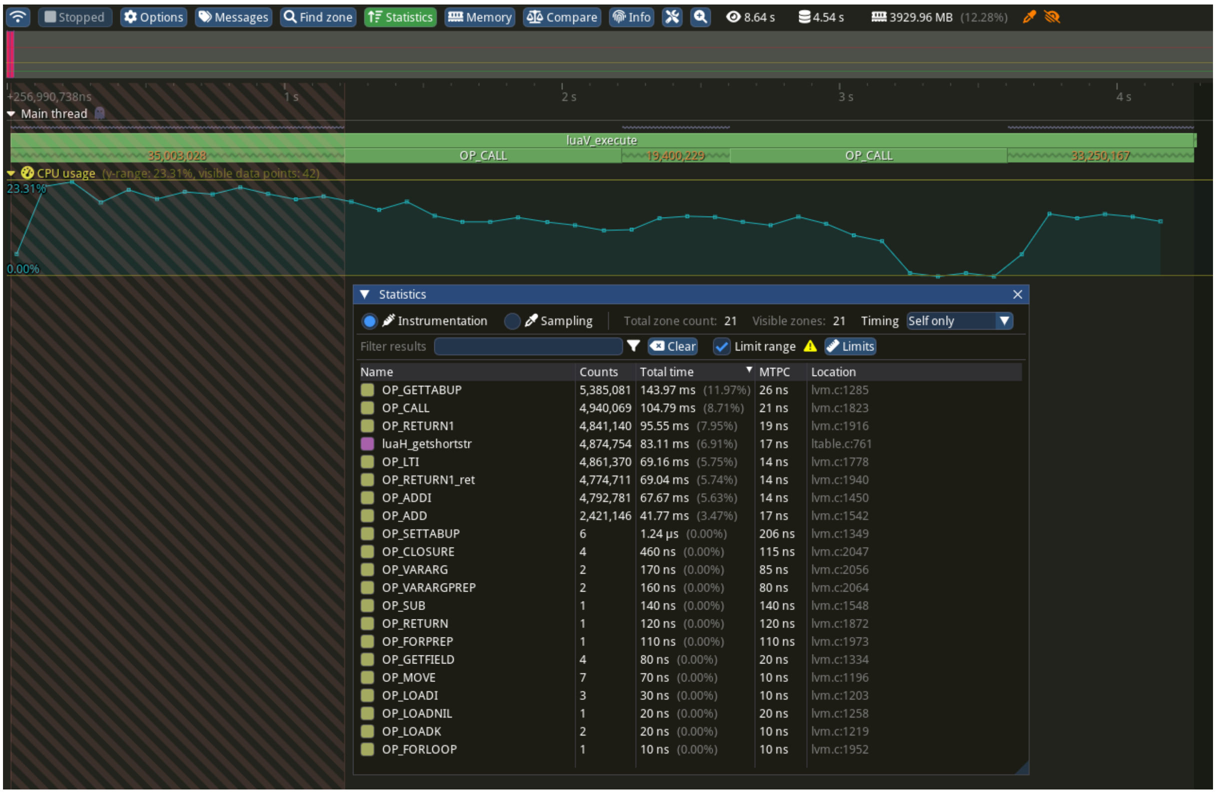
As the large number of zones created this way quickly overwhelms the Tracy Profiler host application, I created a modified version of the `fib.lua` script that calls each `fibonacci_` variant only 1/100x as often, and sleeps for one second in between testing the three variants:

Note that this *did* have a substantial performance impact, increasing the runtime by a factor of 2-4.

```
-- [...]

t, v = measure(1, fibonacci_naive, 30)
print(string.format("1 x fibonacci_naive(30)      time: %8.4f s  --  %s", t, v))

os.execute("sleep 1")

t, v = measure(100000, fibonacci_tail, 30)
print(string.format("100000 x fibonacci_tail(30) time: %8.4f s  --  %s", t, v))

os.execute("sleep 1")

t, v = measure(250000, fibonacci_iter, 30)
print(string.format("250000 x fibonacci_iter(30) time: %8.4f s  --  %s", t, v))
```

The following screenshot shows

1. the overall execution profile (note the 1-second long `OP_CALL` zones in between the orange-numbered ones representing the `sleep` durations).
2. statistics for `100x fibonacci_naive` (obtained by right-clicking on the first orange-numbered section, choosing “Limit statistics time range”, then opening the Statistics window using the button at the top).



Name	Counts	Total time	MTPC	Location
OP_GETTABUP	5,385,081	143.97 ms (11.97%)	26 ns	lvm.c:1285
OP_CALL	4,940,069	104.79 ms (8.71%)	21 ns	lvm.c:1823
OP_RETURN1	4,841,140	95.55 ms (7.95%)	19 ns	lvm.c:1916
luaH_getshortstr	4,874,754	83.11 ms (6.91%)	17 ns	ltable.c:761
OP_LTI	4,861,370	69.16 ms (5.75%)	14 ns	lvm.c:1778
OP_RETURN1_ret	4,774,711	69.04 ms (5.74%)	14 ns	lvm.c:1940
OP_ADDI	4,792,781	67.67 ms (5.63%)	14 ns	lvm.c:1450
OP_ADD	2,421,146	41.77 ms (3.47%)	17 ns	lvm.c:1542
OP_SETTABUP	6	1.24 µs (0.00%)	206 ns	lvm.c:1349
OP_CLOSURE	4	460 ns (0.00%)	115 ns	lvm.c:2047
OP_VARARG	2	170 ns (0.00%)	85 ns	lvm.c:2056
OP_VARARGPREP	2	160 ns (0.00%)	80 ns	lvm.c:2064
OP_SUB	1	140 ns (0.00%)	140 ns	lvm.c:1548
OP_RETURN	1	120 ns (0.00%)	120 ns	lvm.c:1872
OP_FORPREP	1	110 ns (0.00%)	110 ns	lvm.c:1973
OP_GETFIELD	4	80 ns (0.00%)	20 ns	lvm.c:1334
OP_MOVE	7	70 ns (0.00%)	10 ns	lvm.c:1196
OP_LOADI	3	30 ns (0.00%)	10 ns	lvm.c:1203
OP_LOADNIL	1	20 ns (0.00%)	20 ns	lvm.c:1258
OP_LOADK	2	20 ns (0.00%)	10 ns	lvm.c:1219
OP_FORLOOP	1	10 ns (0.00%)	10 ns	lvm.c:1952

All opcode statistics are shown in the following screenshots.

Name	Counts	Total time	MTPC	Location
OP_GETTABUP	5,385,081	143.97 ms (11.97%)	26 ns	lvm.c:1285
OP_CALL	4,940,069	104.79 ms (8.71%)	21 ns	lvm.c:1823
OP_RETURN1	4,841,140	95.55 ms (7.95%)	19 ns	lvm.c:1916
luaH_getshortstr	4,874,754	83.11 ms (6.91%)	17 ns	ltable.c:761
OP_LTI	4,861,370	69.16 ms (5.75%)	14 ns	lvm.c:1778
OP_RETURN1_ret	4,774,711	69.04 ms (5.74%)	14 ns	lvm.c:1940
OP_ADDI	4,792,781	67.67 ms (5.63%)	14 ns	lvm.c:1450
OP_ADD	2,421,146	41.77 ms (3.47%)	17 ns	lvm.c:1542
OP_SETTABUP	6	1.24 µs (0.00%)	206 ns	lvm.c:1349
OP_CLOSURE	4	460 ns (0.00%)	115 ns	lvm.c:2047
OP_VARARG	2	170 ns (0.00%)	85 ns	lvm.c:2056
OP_VARARGPREP	2	160 ns (0.00%)	80 ns	lvm.c:2064
OP_SUB	1	140 ns (0.00%)	140 ns	lvm.c:1548
OP_RETURN	1	120 ns (0.00%)	120 ns	lvm.c:1872
OP_FORPREP	1	110 ns (0.00%)	110 ns	lvm.c:1973
OP_GETFIELD	4	80 ns (0.00%)	20 ns	lvm.c:1334
OP_MOVE	7	70 ns (0.00%)	10 ns	lvm.c:1196
OP_LOADI	3	30 ns (0.00%)	10 ns	lvm.c:1203
OP_LOADNIL	1	20 ns (0.00%)	20 ns	lvm.c:1258
OP_LOADK	2	20 ns (0.00%)	10 ns	lvm.c:1219
OP_FORLOOP	1	10 ns (0.00%)	10 ns	lvm.c:1952

Figure 2: fibonacci_naive

Name	Counts	Total time	MTPC	Location
OP_TAILCALL	3,019,114	37.02 ms (9.55%)	12 ns	lvm.c:1843
OP_MOVE	3,033,352	31.64 ms (8.16%)	10 ns	lvm.c:1196
OP_EQI	2,793,737	30.29 ms (7.81%)	10 ns	lvm.c:1763
OP_GETUPVAL	2,663,668	27.93 ms (7.21%)	10 ns	lvm.c:1268
OP_ADD	2,654,272	27.81 ms (7.17%)	10 ns	lvm.c:1542
OP_ADDI	2,655,581	27.54 ms (7.11%)	10 ns	lvm.c:1450
OP_CLOSURE	100,001	3.52 ms (0.91%)	35 ns	lvm.c:2047
luaF_findupval	97,955	3.16 ms (0.81%)	32 ns	lfunc.c:89
luaC_step	1,429	3.12 ms (0.81%)	2.19 µs	lgc.c:1691
OP_LOADI	178,995	1.91 ms (0.49%)	10 ns	lvm.c:1203
OP_CALL	97,941	1.29 ms (0.33%)	13 ns	lvm.c:1823
OP_VARARG	88,267	950.59 µs (0.25%)	10 ns	lvm.c:2056
OP_RETURN1	90,338	947.99 µs (0.24%)	10 ns	lvm.c:1916
OP_RETURN1_ret	88,557	935.45 µs (0.24%)	10 ns	lvm.c:1940
OP_FORLOOP	88,516	905.18 µs (0.23%)	10 ns	lvm.c:1952
luaF_closeupval	87,300	886.07 µs (0.23%)	10 ns	lfunc.c:197
OP_GETTABUP	9	990 ns (0.00%)	110 ns	lvm.c:1285
luaH_getshortstr	14	590 ns (0.00%)	42 ns	ltable.c:761
OP_GETFIELD	4	290 ns (0.00%)	72 ns	lvm.c:1334
OP_FORPREP	1	210 ns (0.00%)	210 ns	lvm.c:1973
OP_SUB	1	120 ns (0.00%)	120 ns	lvm.c:1548
OP_VARARGPREP	1	120 ns (0.00%)	120 ns	lvm.c:2064
OP_LOADK	2	90 ns (0.00%)	45 ns	lvm.c:1219
OP_SETTABUP	2	70 ns (0.00%)	35 ns	lvm.c:1349
OP_RETURN	1	60 ns (0.00%)	60 ns	lvm.c:1872
OP_LOADNIL	1	30 ns (0.00%)	30 ns	lvm.c:1258

Figure 3: fibonacci_tail

Name	Counts	Total time	MTPC	Location
OP_MOVE	13,996,315	168.63 ms (25.11%)	12 ns	lvm.c:1196
OP_FORLOOP	6,878,884	85.67 ms (12.76%)	12 ns	lvm.c:1952
OP_ADD	6,615,254	80.55 ms (12.00%)	12 ns	lvm.c:1542
OP_LOADI	886,116	10.86 ms (1.62%)	12 ns	lvm.c:1203
OP_CALL	241,059	3.16 ms (0.47%)	13 ns	lvm.c:1823
OP_RETURN1_ret	220,566	3.01 ms (0.45%)	13 ns	lvm.c:1940
OP_FORPREP	235,608	2.94 ms (0.44%)	12 ns	lvm.c:1973
OP_RETURN1	224,413	2.38 ms (0.35%)	10 ns	lvm.c:1916
OP_VARARG	216,250	2.3 ms (0.34%)	10 ns	lvm.c:2056
OP_GETTABUP	8	5.41 µs (0.00%)	676 ns	lvm.c:1285
luaH_getshortstr	13	560 ns (0.00%)	43 ns	ltable.c:761
OP_SETTABUP	2	270 ns (0.00%)	135 ns	lvm.c:1349
OP_GETFIELD	3	180 ns (0.00%)	60 ns	lvm.c:1334
OP_LOADK	2	150 ns (0.00%)	75 ns	lvm.c:1219
OP_SUB	1	140 ns (0.00%)	140 ns	lvm.c:1548
OP_VARARGPREP	1	130 ns (0.00%)	130 ns	lvm.c:2064
OP_RETURN	2	130 ns (0.00%)	65 ns	lvm.c:1872
OP_LOADNIL	1	30 ns (0.00%)	30 ns	lvm.c:1258

Figure 4: fibonacci_iter

Name	Counts	Total time	MTPC	Location
luaV_execute	1	1.51 s (49.33%)	1.51 s	lvm.c:1156
OP_MOVE	17,216,014	196.06 ms (6.40%)	11 ns	lvm.c:1196
OP_ADD	11,795,864	148.88 ms (4.86%)	12 ns	lvm.c:1542
OP_GETTABUP	5,385,096	133.06 ms (4.34%)	24 ns	lvm.c:1285
OP_ADDI	7,493,650	122.28 ms (3.99%)	16 ns	lvm.c:1450
OP_RETURN1_ret	5,090,855	92.29 ms (3.01%)	18 ns	lvm.c:1940
OP_CALL	5,292,011	89.19 ms (2.91%)	16 ns	lvm.c:1823
luaH_getshortstr	4,931,336	88.48 ms (2.89%)	17 ns	ltable.c:761
OP_LTI	4,851,445	84.25 ms (2.75%)	17 ns	lvm.c:1778
OP_RETURN1	5,162,988	79.39 ms (2.59%)	15 ns	lvm.c:1916
OP_FORLOOP	7,043,579	75.81 ms (2.47%)	10 ns	lvm.c:1952
OP_TAILCALL	3,036,706	56.96 ms (1.86%)	18 ns	lvm.c:1843
OP_EQI	2,830,541	48.58 ms (1.58%)	17 ns	lvm.c:1763
OP_GETUPVAL	2,700,741	45.04 ms (1.47%)	16 ns	lvm.c:1268
OP_LOADI	1,074,713	11.92 ms (0.39%)	11 ns	lvm.c:1203
OP_CLOSURE	100,005	4.06 ms (0.13%)	40 ns	lvm.c:2047
luaF_findupval	98,507	3.92 ms (0.13%)	39 ns	lfunc.c:89
luaC_step	1,437	3.49 ms (0.11%)	2.43 µs	lgc.c:1691
OP_VARARG	307,268	3.34 ms (0.11%)	10 ns	lvm.c:2056
OP_FORPREP	238,084	2.72 ms (0.09%)	11 ns	lvm.c:1973
luaF_closeupval	88,094	1.72 ms (0.06%)	19 ns	lfunc.c:197
luaH_getstr	373	9.83 µs (0.00%)	26 ns	ltable.c:779
OP_RETURN	4	2.13 µs (0.00%)	532 ns	lvm.c:1872
OP_SETTABUP	10	1.75 µs (0.00%)	175 ns	lvm.c:1349
OP_VARARGPREP	4	820 ns (0.00%)	205 ns	lvm.c:2064
OP_SUB	3	590 ns (0.00%)	196 ns	lvm.c:1548
OP_GETFIELD	9	390 ns (0.00%)	43 ns	lvm.c:1334
OP_LOADK	5	130 ns (0.00%)	26 ns	lvm.c:1219
OP_LOADNIL	3	50 ns (0.00%)	16 ns	lvm.c:1258

Figure 5: all three

Main Takeaways

- `fibonacci_naive` makes many, many calls to `getshortstr` via `OP_GETTABUP` – I assume this has to do with accessing the variable `n` from within recursive calls.
- `OP_GETTABUP` is used much more often than `OP_SETTABUP` and its mean-time-per-call is roughly twice that of other opcodes. Optimising `OP_GETTABUP` at the expense of `OP_SETTABUP` is probably a good call for this benchmark.
- `fibonacci_tail` calls `findupval` and `closeupval` roughly once per iteration. This is related to creation of the "inner" closure via `OP_CLOSURE`.
- The garbage collector (`luaC_step`) runs most often during `fibonacci_tail` – taking up about as much runtime as `findupval`.
- `OP_MOVE`, `OP_FORLOOP` and `OP_ADD` represent the bulk of the opcodes used by `fibonacci_iter` with several million calls each.

Conclusion

We decided to take a look at the functions using the most time and tried to optimize them in exercise D.

C) Code Understanding

Overall process of Lua execution in the interpreter

- Loading: The Lua interpreter reads the script from a file or string.
- Lexical Analysis (Lexing): The script is broken down into tokens.
- Syntax Analysis (Parsing): The tokens are parsed to create an Abstract Syntax Tree (AST).
- Code Generation: The AST is converted into bytecode, which is the intermediate representation of the script.
- Execution: The bytecode is executed by the Lua Virtual Machine (VM).

`LUA_USE_JUMPTABLE`

The bulk of the main loop of the interpreter is taken up by a `switch`/`case` construct, with one `case` label per opcode supported by the Lua Virtual Machine (e.g. `case OP_MOVE:`).

When the macro `LUA_USE_JUMPTABLE` is defined, the `switch`/`case` construct is replaced with a manually-defined jump table:

1. Instead of a `case` label, each Virtual Machine opcode is assigned a "regular" label (e.g. `L_OP_MOVE:`), and the addresses for each of these are stored in the table.
2. Immediately after an opcode has been processed, the next opcode is fetched and execution is transferred to its corresponding label using `goto` – as opposed to `break`ing out of the `switch` block, then fetching and jumping to the next opcode on the next loop iteration.

While the `switch`/`case` construct is also a jump table, the idea of constructing one manually is that, by avoiding a jump back to the top of the interpreter loop before processing the next opcode, performance could be improved.

However as Roberto (creator of Lua) noted in 2018, it may actually cause performance *regressions* on some systems.

My performance testing seems to reflect this – across ten runs of `fib.lua` on LCC3, the jump table seems to make performance ever so slightly worse. The difference in runtime is roughly one standard deviation for the `iter` and `tail` variants, and one fifth of a standard deviation for the `naive` variant.

test	wall (mean)	wall (stddev)
iter	10.8730	0.036
iter (jumptable)	10.9105	0.036
naive	12.5483	0.322
naive (jumptable)	12.6001	0.316
tail	12.5980	0.046
tail (jumptable)	12.6484	0.027

(D) Optimization

Varying the compilation flags

We tried different compiler flags – this did not produce any meaningful improvements. With the exception of `fibonacci_tail`, compiling with `-O3` or `-Ofast` made performance *worse*:

test	wall (mean)	wall (stddev)	% vs `-O2`
iter	10.8730	0.036	
iter (`-O3`)	12.2039	0.042	112.24

test	wall (mean)	wall (stddev)	% vs -O2
iter (-Ofast)	11.6290	0.031	106.95
naive	12.5483	0.322	
naive (-O3)	13.4102	0.296	106.87
naive (-Ofast)	12.7103	0.306	101.29
tail	12.5980	0.046	
tail (-O3)	12.5004	0.020	99.23
tail (-Ofast)	12.0485	0.018	95.64

Applying individual flags from the `-O3` optimization level in addition to `-O2` also did not have any meaningful impact. The greatest improvement achieved by a single flag is `-fsplit-paths` for `fibonacci_tail`, being 0.15 seconds faster than `-O2` (~1.3%).

All mean results are contained in the file named `flag_results.csv`.

Improving Lua source code

We tried to improve the functions mentioned in *B - Profiling with Gprof*. We tried to inline external function calls. However the only function that was no define which is inlined by the compiler anyway was `prepCallInfo` in `luaD_precall` in `ldo.c`. (Note: Only adapted the last case, as it is use for running Lua files and our benchmark is in Lua)

The results were slightly faster (no jump table was also faster than jump table):

- naive: 12.3457 s
- tail: 12.5954 s
- iter: 10.8881 s

The adapted Code looks like this:

```
CallInfo *luaD_precall (lua_State *L, StkId func, int nresults) {
 retry:
  switch (ttypetag(s2v(func))) {
    case LUA_VCCL:  /* C closure */
      precallC(L, func, nresults, clCvalue(s2v(func))->f);
      return NULL;
    case LUA_VLCF:  /* light C function */
      precallC(L, func, nresults, fvalue(s2v(func)));
      return NULL;
    case LUA_VLCL: {  /* Lua function */
      CallInfo *ci;
      Proto *p = clLvalue(s2v(func))->p;
      int narg = cast_int(L->top.p - func) - 1;  /* number of real arguments */
      int nfixparams = p->numparams;
      int fsize = p->maxstacksize;  /* frame size */
      checkstackGCp(L, fsize, func);

      // ####################################### try inlining this function
      // original code
      //L->ci = ci = prepCallInfo(L, func, nresults, 0, func + 1 + fsize);
      // #######################################

      ci = L->ci = next_ci(L);  /* new frame */
      ci->func.p = func;
      ci->nresults = nresults;
      ci->callstatus = 0;
      ci->top.p = func + 1 + fsize;

      L->ci = ci;
```

```
      // ######################################
      // original function
      //l_sinline CallInfo *prepCallInfo (lua_State *L, StkId func, int nret,
      //                                          int mask, StkId top) {
      //CallInfo *ci = L->ci = next_ci(L);  /* new frame */
      //ci->func.p = func;
      //ci->nresults = nret;
      //ci->callstatus = mask;
      //ci->top.p = top;
      //return ci;
      //}

      // ######################################

      ci->u.l.savedpc = p->code;  /* starting point */
      for (; narg < nfixparams; narg++)
        setnilvalue(s2v(L->top.p++));  /* complete missing arguments */
      Lua_assert(ci->top.p <= L->stack_last.p);
      return ci;
    }
    default: {  /* not a function */
      func = LuaD_tryfuncTM(L, func);  /* try to get '__call' metamethod */
      /* return LuaD_precall(L, func, nresults); */
      goto retry;  /* try again with metamethod */
    }
  }
}
```

Improving the op calls analyzed with tracy seemed very hard. Probably implementing caching / memoization would rapidly improve the benchmark for a problem like fibonacci calculation. However implementing this in the Lua interpreters source code and not simply adding memoization to the `.lua` file would exceed the scope of this exercise sheet.

Using a custom memory allocator for the Lua interpreter

As we had good experience with custom memory allocators in terms of performance improvements in previous exercises, we decided to try `mimalloc` and see if it can improve the perfomance.

Mimalloc Setup

First we build mimalloc from source, as described in the repository. After that we added the Flag `-lmimalloc` during the build of the interpreter to use it for all the allocations.

Mimalloc Results

Unfortunatly mimalloc couldn't really speed up the execution times for this scenario.

The baseline results were (avg. from 10 runs):

```
func,wall (mean)
fibonacci_iter(30),10.8779
fibonacci_naive(30),12.7706
fibonacci_tail(30),12.6074
```

And with mimalloc:

```
func,wall (mean)
fibonacci_iter(30),10.9223
fibonacci_naive(30),12.5884
fibonacci_tail(30),12.6433
```

We can see that the "improvements" fall within a margin of error. Our possible explanations for that were:

1. The program probably is rather compute bound and not so heavy on the memory.
2. Maybe Lua already uses a non-default memory allocator / they have their own implementation. A look in the source code, we can see the file `lmem.c` which contains the implementation of Lua's memory handling. Of course this is completed by the garbage collector in seperate files.