Performance Oriented Computing - Sheet 12

Tobias Beiser Johannes Karrer Andreas Kirchmair

June 11, 2024

1 Task 1: Setup and Basic Execution

Goal of this task was to download and build lua and execute the ‘fib.lua‘ benchmark. Steps:

- Get the latest lua version

  `wget https://github.com/PeterTh/perf-oriented-dev/blob/master/lua/fib.lua`

- Build lua using make.
- Execute the benchmark.
- Results:

```
100 x fibonacci_naive(30)     time:  18.2945 s  --  832040
10000000 x fibonacci_tail(30) time:  17.2246 s  --  832040
25000000 x fibonacci_iter(30) time:  14.9098 s  --  832040
```

2 Task 2: Profiling

PASS

3 Task 3: Code Understanding

3.1 Lua execution process

The lua exection process involves the following steps:

1. Lexical Analysis: The source code is broken down into tokens.
2. Syntactic Analysis (Parsing): Tokens are parsed into an abstract syntax tree (AST).
3. Semantic Analysis: The AST is analyzed to ensure semantic correctness.
4. Bytecode Generation: The AST is compiled into Lua bytecode.
5. Execution: The Lua virtual machine executes the bytecode.

The execution time of the first 4 steps can be measured by adjusting the *handle_script* function in lua.c:

```
static int handle_script (lua_State *L, char **argv) {
        int status;
        const char *fname = argv[0];
        if (strcmp(fname, "-") == 0 && strcmp(argv[-1], "--") != 0)
        fname = NULL;  /* stdin */

        clock_t start_compile = clock();
```

```
        status = luaL_loadfile(L, fname);
        clock_t end_compile = clock();

        double compile_time = (double)(end_compile - start_compile) / CLOCKS_PER_SEC;
        printf("Compile time: %f seconds\n", compile_time);

        if (status == LUA_OK) {
                int n = pushargs(L);  /* push arguments to script */
                status = docall(L, n, LUA_MULTRET);
        }
        return report(L, status);
}
```

The execution time of the step 5 can be measured by adjusting the *docall* function in lua.c:

```
static int docall (lua_State *L, int narg, int nres) {
        int status;
        int base = lua_gettop(L) - narg;  /* function index */
        lua_pushcfunction(L, msghandler);  /* push message handler */
        lua_insert(L, base);  /* put it under function and args */
        globalL = L;  /* to be available to 'laction' */
        setsignal(SIGINT, laction);  /* set C-signal handler */

        clock_t start_exec = clock();
        status = lua_pcall(L, narg, nres, base);
        clock_t end_exec = clock();

        double exec_time = (double)(end_exec - start_exec) / CLOCKS_PER_SEC;
        printf("Execution time: %f seconds\n", exec_time);

        setsignal(SIGINT, SIG_DFL); /* reset C-signal handler */
        lua_remove(L, base);  /* remove message handler from the stack */
        return status;
}
```

We can see that the whole analysis and compilation process takes way less time than the execution of the actual function:

```
Compile time: 0.000064 seconds
100 x fibonacci_naive(30)     time:  18.2945 s  --  832040
10000000 x fibonacci_tail(30) time:  17.2246 s  --  832040
25000000 x fibonacci_iter(30) time:  14.9098 s  --  832040
Execution time: 50.428908 seconds
```

3.2 LUA_USE_JUMPTABLE

The LUA_USE_JUMPTABLE option is a compilation flag used in the Lua interpreter to improve the performance of the virtual machine by using a jump table instead of a switch statement for the main loop of the VM.

- LUA_USE_JUMPTABLE = false: The interpreter uses a switch-case statement to dispatch byte-code instructions.
- LUA_USE_JUMPTABLE = true: The interpreter uses a direct threading technique, where each opcode directly jumps to the corresponding handler, reducing the overhead of branch prediction and making instruction dispatch faster.

For execution time comparison, the lua interpreter is compiled twice. The results show that using a jumptable is slightly faster, but the difference is not very significant, at least for this benchmark.

- LUA_USE_JUMPTABLE = false

```
100 x fibonacci_naive(30)     time:  18.2835 s  --  832040
10000000 x fibonacci_tail(30) time:  17.2264 s  --  832040
25000000 x fibonacci_iter(30) time:  14.9101 s  --  832040
```

- LUA_USE_JUMPTABLE = true

```
100 x fibonacci_naive(30)     time:  18.0832 s  --  832040
10000000 x fibonacci_tail(30) time:  17.0894 s  --  832040
25000000 x fibonacci_iter(30) time:  14.8177 s  --  832040
```

4 Task 4: Optimization

PASS