

■ GET-INTERNAL-RUN-TIME has very low fidelity

The typical way to do timing in Common Lisp is either to call the high-level `time` macro (for a rough result), or to call `get-internal-run-time` before and after (for more accurate numbers). However, for some reason the Lucid clock only ticks in increments of 0.01 seconds, so the difference between subsequent calls to `get-internal-run-time` will never be less than 0.01 sec. This is fine for timing long processes, but for these very fast operations, more fidelity is needed. One solution is to call the routine multiple times, time it, and then take the average time. You don't want to do this in an explicit loop, because you are then timing the overhead of the looping as well. So a macro that expands directly into repeated calls is a good solution. Loading `~hall/Lisp/Timing-Macros.sbin` will give you a simple timing macro `Time-Form` that takes a Lisp form and optionally a number `N` (defaulting to 20). It executes the form `N` times, records the total time, and then divides that time by `N`. E.g. assuming `Find-Matching-Slot-Value` is your function from #1,

```
(Time-Form (Find-Matching-Slot-Value Slot-Value Instance-List))
```

■ Garbage collection can throw timing off

Garbage collection (`gc`) is when Lisp reclaims memory that was allocated but is no longer used. An ephemeral `gc`, the most common type, takes around 0.1 seconds, and can make your timing results vary. In some sense you want to count this time, since a routine that generates more garbage does slow the system down. But since it is sporadic, you might want to try turning garbage collection off while timing. You can call `gc-off` and `gc-on/egc-on` yourself, or use `t` macro, also defined in `~hall/Lisp/Timing-Macros.sbin`. E.g.

```
(Without-GC
 (Time-Form (Find-Matching-Slot-Value Slot-Value Instance-List)))
```

■ The compiler optimization settings affect the results

You might want to do your timing runs twice. Once with the default, high error-checking settings, and once with the fast, highly optimized settings. To do the first, type

```
(proclaim `(optimize (safety 3) (compilation-speed 3) (speed 2)))
```

then recompile your files, then rerun them. To do the optimized case

```
(proclaim `(optimize (safety 1) (compilation-speed 0) (speed 3)))
```

then recompile your files and rerun your test cases.

■ The compiler can optimize away some operations.

Consider a function that is used to try to determine the floating-point speed of Lisp:

```
(loop for I from 1 to 100000 do (* 0.1 0.2 0.3 0.4 0.5))
```

With high optimization settings, the compiler will notice that the inner multiplication never gets saved, and will simply remove it, leaving you with the same timing results as if you had done `(loop for I from 1 to 100000 do NIL)`

Even if you do

```
(loop for I from 1 to 100000 do
 (setq *Global* (* 0.1 0.2 0.3 0.4 0.5)))
```

the compiler could do the multiplication once outside the loop, or even (if particularly clever), only execute the loop once. The only 100% foolproof method is to `disassemble` the code and look at the resultant assembler, but being vigilant for this is half the battle.

■ Use the built-in profiler to find out where a routine spends its time

The built-in metering system will trace a chain of function calls and show the percent of time spent in various subfunctions. It should definitely be used for almost any serious optimization effort. See Chapter 2 of Lucid's Delivery Tool Kit docs for the functions `start-backtrace-logging`, `stop-backtrace-logging`, and `summarize-backtrace-logging`. However, if you wrap your code inside `With-Metering` (also defined in `~hall/Lisp/Timing-Macros.sbin`) the total time (slightly inflated due to the overhead of timing/metering) will be printed along with the backtrace logging info showing the breakdown of that time.

```
(With-Metering ()
 <Code to Profile>)
```