Performance Evaluation of Crystal

Nicolas Ganz, Prof. Jürgen Spielberger
ZHAW Zurich University of Applied Sciences

July 2021

Abstract

Crystal is a new programming language, which tries to combine the simplicity to write software of Ruby with the performance of C. This study aims to compare the performance of Crystal with the programming languages Ruby, C and Go.

This is done by using different example programs that use specific parts used in real world applications. Those include iterative and recursive implementations of the Fibonacci sequence, reading and writing files, listening to sockets, as well as calling a method written in C.

The results show that Crystal can be considered a fast programming language. While C with all optimisations of `gcc` is still faster, the performance of Crystal is comparable with Go. As expected is Ruby, with just-in-time (JIT) compilation or without, by a factor of 8 respectively 9 slower than Crystal.

Contents

List of Tables

List of Figures

1 Introduction

Crystal is a new programming language. It has the goal of combining Ruby's efficiency for writing code and C's efficiency for running code [1]. The goal of this report is to compare the performance of Crystal with different programming languages.

1.1 Comparing Performance

For comparing performance of programming languages benchmarks are often used. There exist lists of different programs that are implemented in different languages to compare them. *The Computer Language Benchmarks Game* [2] is one of them and is implemented in different languages. It shows that measuring performance of programming languages using real world programs would be ideal but requires a lot of work. Additionally it also requires in depth knowledge of all languages to not accidentally implement a part of the program inefficiently. While there is no official implementation of *the Computer Language Benchmarks Game* in Crystal there exists an unofficial one [3]. Other languages are compared in many different benchmarks as well [4, 5, 6].

Comparing real world applications is too complex, but the issue with the simplified applications is that it mostly uses different algorithms and only measuring those entirely using the programming language itself. Real world applications on the other hand also interact with things outside of the programming language, like files, sockets and libraries written in other languages like C. What this report tries to achieve is to measure the performance of specific parts of programs used in real world applications. These parts include recursive and iterative functionalities, reading and writing files, using sockets, as well as calling methods written in C.

2 Method

2.1 Test Setup

The general system information used to measure the performance of the programming languages is described in table 1. The version numbers of all compilers and interpreters are shown in table 2 on the next page.

OS	Linux-5.10.36-2-MANJARO-x86_64-with-glibc2.33
CPU	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
RAM	15.4 GiB
Disk	NVMe disk - HFS512GD9TNG-62A0A

Table 1: The system setup used for the performance measurements

Crystal	1.0.0 (LLVM: 10.0.1)
Ruby	3.0.1p64
GCC	10.2.0
Go	1.16.4
GNU time	1.9-3

Table 2: Version numbers of the compilers and interpreters

2.2 Performance Tests

To measure the performance of the programming languages multiple methods were chosen. The Crystal code is shown in this report as an example of the exact implementation.

2.2.1 Startup Time

To get a better understanding of how much time the basic setup — for example interpreting the source code — of the program requires the algorithm 1 is used. It takes the real time at the beginning of the program, sleeps for one second and calculates the time spent sleeping.

If this internal time is subtracted from the duration of the entire program the result will be the time it takes for the basic setup and one second. This second is added because the call to `sleep` is not measured by the CPU time.

Algorithm 1: Measuring the startup time

```
1 start_time = Time.utc
2 sleep 1
3 end_time = Time.utc
4
5 puts (end_time - start_time).total_seconds
```

2.2.2 Recursive Fibonacci

To measure how efficient method calls are a recursive Fibonacci algorithm, as described in algorithm 2, is used to calculate F_{42}. F_{42} is used, because it runs in a reasonable time in Ruby but all other programming languages still take a measurable amount of time.

Algorithm 2: Recursive Fibonacci implementation

```
def fibonacci(n)
  return n if n == 0 || n == 1
  fibonacci(n - 1) + fibonacci(n - 2)
end

puts fibonacci(42)
```

2.2.3 Recursive Fibonacci Without Optimisations

The problem with algorithm 2 is that some compilers can detect and eliminate tail-recursion [7]. LLVM, used in Crystal, for example uses this [8]. The algorithm 3 is introduced to see if this tail-recursion optimisation was used by any compiler.

Just assigning the recursive calls to variables does not necessarily remove the possibility of this optimisation. It is necessary to actually use those variables. This is done by comparing the variables if $n \neq 3$. If $n = 3$ then a and b are both 1 and would therefore fail.

Algorithm 3: Recursive Fibonacci implementation without optimisations

```
def fibonacci(n)
  return n if n == 0 || n == 1
  a = fibonacci(n - 1)
  b = fibonacci(n - 2)
  raise "should not happen" if n != 3 && a == b
  a + b
end

puts fibonacci(42)
```

2.2.4 Iterative Fibonacci

Algorithm 4 calculates the Fibonacci number F_{93}. F_{93} is chosen, because it is the largest Fibonacci number that can be contained in a 64 bit unsigned integer. But since this implementation is extremely fast in comparison with the recursive implementation it is run 10 000 000 times and one additional time that gets printed. Otherwise it would not be measurable. The result is compared with the expected result to make sure that the result is actually used and the calls are not skipped by the optimiser.

Algorithm 4: Iterative Fibonacci implementation

```
def fibonacci(n)
  a = 0_u64
  b = 1_u64
  (n - 1).times do
    a, b = b, a + b
  end
  b
end

10_000_000.times do
  raise "invalid number" if fibonacci(93) != 12200160415121876738
end
puts fibonacci(93)
```

2.2.5 Writing Lines to Files

To measure how efficiently the programming language can write to files algorithm 5 creates a file with 100 000 000 lines one at at time containing “0123456789”. This leads to a file with a size of 1.1 GiB.

Algorithm 5: Writing lines to a file

```
CONTENT = "0123456789\n"

def write_file
  File.open("../tmp/crystal_file.tmp", "w") do |file|
    100_000_000.times do
      file.print CONTENT
    end
  end
end

write_file
```

2.2.6 Writing Longer Lines to Files

To verify that write buffers are used algorithm 6 writes ten lines at a time and only runs 10 000 000 times. This creates the exact same file as in algorithm 5 on the previous page, but it will write it in larger chunks.

Algorithm 6: Writing longer lines to a file

```
CONTENT = 10.times.map { "0123456789\n" }.join

def write_file
  File.open("../tmp/crystal_file.tmp", "w") do |file|
    10_000_000.times do
      file.print CONTENT
    end
  end
end

write_file
```

2.2.7 Reading Lines from Files

To measure the speed at which files can be read algorithm 7 reads one line at a time of a file and verifies that each line is correct. A file created with algorithm 5 on the previous page is used.

Algorithm 7: Reading lines from a file

```
CONTENT = "0123456789"

def read_file
  File.read("../example.txt").each_line do |line|
    raise "invalid line: #{line}" if line != CONTENT
  end
end

read_file
```

2.2.8 C Bindings

Interact with code written in C is useful to get access to many important libraries and frameworks. Algorithm 8 measures how efficient it is to call a method in C. For that the `cos` function from `math.h` is used. To prevent the compiler from optimising away any calls the result is compared with the previous result. This is repeated for all numbers from 0 up to but excluding 100 000 000.

Algorithm 8: Calling C methods

```
lib C
  fun cos(value : Float64) : Float64
end

def calculate_cos(n)
  C.cos n
end

prev = -1
100_000_000.times do |i|
  value = calculate_cos i
  raise "stayed the same" if value == prev
  prev = value
end
puts prev
```

2.2.9 TCP Sockets

Another important aspect of programming languages is how efficient TCP sockets can be used. To measure this a simple web server is implemented in algorithm 9. Every time a request is received it returns a predefined plain text HTTP response.

Algorithm 9: Listening to TCP sockets

```
require "socket"

MESSAGE = "HTTP/1.1 200 OK\r\n" +
  "Content-Type: text/plain\r\n" +
  "Content-Length: 12\r\n" +
  "Connection: close\r\n\r\n" +
  "Test Message"

TCPServer.open 2000 do |server|
  puts "ready"
  while client = server.accept?
    client.gets
    client.print MESSAGE
    client.close
  end
end
```

2.3 Parallelism

Parallelism is also important to compare, but parallelism is not supported by all languages.

Crystal does not yet fully support parallelism [9]. There is a preview option where you can pass the number of parallel workers as an environment variable, which will enable running Fibers in parallel [10].

Before version 3 of Ruby it did not support parallelism either, since there was a Global Interpreter Lock (GIL) for the entire runtime environment. With version 3, released on 25 December 2020, they have introduced a more complex abstraction where there is a GIL per Ractor and is therefore able to run multiple Ractors in parallel [11].

Go has yet another approach to parallelism using Goroutines and Channels [12].

Since every language has different concepts, all with their advantages and limitations, it is nearly impossible to get significant and comparable results. This is therefore not further looked at in this report.

2.4 Measuring Performance

For measuring the running time of the programs there are multiple options. Using the well known GNU `time` command [13] there are the following five options concerning the running time of a program.

```
Time
%E Elapsed real time (in [hours:]minutes:seconds).
%e (Not in tcsh(1).)  Elapsed real time (in seconds).
%S Total number of CPU-seconds that the process spent in
 ↪  kernel mode.
%U Total number of CPU-seconds that the process spent in
 ↪  user mode.
%P Percentage of the CPU that this job got, computed as (%U
 ↪  + %S) / %E.
```

— man page of the GNU `time` command [14]

Using the real time, as in option %E, %e or %P, would not be helpful to measure the performance of the programming languages, because it depends on what other programs are running on the computer at the time. This would lead to fluctuating results.

What is interesting for comparing the performance of programming languages are the two CPU-seconds measures. Those measure the amount of CPU time the process got and ignore the time where the CPU is processing other applications. The kernel mode is the privileged mode used to access hardware resources and the user mode for the rest of the application [15]. Therefore the most relevant measure for this project is the combination of CPU-seconds spent in kernel mode and the CPU-seconds in user mode.

To ensure that the result is stable enough and fewer external factors play a role the programs will be run ten times each and the mean is calculated from all runs. To flatten out performance spikes and falloffs between languages each programming language will run one after another and this in turn will be repeated ten times. The combination of those two factors leads to stable test results.

Special Case Measuring the time it takes to listen to TCP sockets cannot be measured as the running time of the program since it is a simple server that responds to requests and keeps running. In this case the measurement is taken using `siege` [16]. This tool is used because it is simple to use, yet configurable. The following options are used to measure the performance.

`--concurrent=1` Use 1 concurrent user since the servers are single-threaded.

`--reps=1000` Complete 1000 requests how ever long it takes.

`--benchmark` Do not add delays in between the iterations.

The duration the `siege` command takes to complete 1000 requests is used as the result for each language.

2.5 Language Options

2.5.1 Ruby

To run the Ruby scripts the command `ruby $test_name.rb` is used. Since it is a scripting language there are no optimisation options to be used.

2.5.2 Ruby (JIT)

One option in ruby is to enable just-in-time (JIT) compilation. This is done by using the command `ruby --jit $test_name.rb`. To see if JIT compilation improves the performance all results are also generated with this option.

2.5.3 C

For compiling C code the well known `gcc` [17] is used. This compiler allows for a lot of optimisations [18]. The most improvements can be achieved with the command `gcc -O3 $test_name.c`.

2.5.4 Go

Go is compiled using the default `gc` compiler flag. [19] This will in turn call the `go tool compile` compiler, which already runs optimisations on the code. The command to compile the Go files is `go build $test_name.go`.

2.5.5 Crystal

Crystal builds the programs by default in a non-optimised version of a binary used for development. To optimise the binary a release mode can be used with the following command `crystal build $test_name.cr --release`.

3 Results

3.1 Performance

3.1.1 Startup Time

Language	**Mean (s)**	**Min (s)**	**Max (s)**	**Stddev (σ)**
C	0.001	0.001	0.002	0.000
Crystal	0.006	0.006	0.007	0.001
Go	0.005	0.002	0.005	0.001
Ruby	0.064	0.052	0.068	0.004
Ruby (JIT)	0.070	0.068	0.074	0.002

Table 3: Difference between the internal and external real time

The "difference" column in table 3 shows that Ruby takes a lot of time to run this script. The usage of JIT compilation improves the performance significantly, but is still about a factor of 10 slower as precompiled languages. The precompiled languages all take a similar amount of time. Of those the

biggest difference is between Crystal and C, where Crystal takes approximately twice as long as C.

3.1.2 Recursive Fibonacci

Language	**Mean (s)**	**Min (s)**	**Max (s)**	**Stddev (σ)**
C	0.490	0.480	0.500	0.005
Crystal	1.127	1.110	1.160	0.013
Go	1.483	1.470	1.530	0.018
Ruby	29.763	29.170	31.060	0.607
Ruby (JIT)	8.630	8.220	9.460	0.341

Table 4: Measurements of the simple Fibonacci algorithm

Table 4 shows that there is a huge difference between Ruby running with JIT compilation or without. Crystal and Go have similar performance, where Crystal is a little bit faster. C on the other hand is by a factor of 2 faster than Crystal and Go.

3.1.3 Recursive Fibonacci Without Optimisations

Language	**Mean (s)**	**Min (s)**	**Max (s)**	**Stddev (σ)**
C	0.630	0.630	0.630	0.000
Crystal	1.264	1.260	1.270	0.005
Go	1.644	1.640	1.650	0.005
Ruby	38.026	36.820	41.460	1.349
Ruby (JIT)	16.453	16.190	16.760	0.242

Table 5: Measurements of the simple Fibonacci algorithm without optimisations

Table 5 shows a minor increase of time compared to table 4 in each programming language which is around the same for all languages.

3.1.4 Iterative Fibonacci

Language	Mean (s)	Min (s)	Max (s)	Stddev (σ)
C	0.475	0.470	0.500	0.011
Crystal	0.357	0.350	0.360	0.005
Go	0.300	0.300	0.300	0.000
Ruby	75.357	73.260	80.030	2.402
Ruby (JIT)	73.593	70.050	76.370	2.208

Table 6: Total CPU time for the iterative Fibonacci implementation

Table 6 shows that by using an iterative approach Go has the best performance followed by Crystal and than C. Ruby, with or without JIT compilation, takes a lot more time.

3.1.5 Writing Lines to Files

Language	Mean (s)	Min (s)	Max (s)	Stddev (σ)
C	2.457	2.370	2.520	0.060
Crystal	1.186	1.160	1.230	0.022
Go	1.202	1.170	1.240	0.027
Ruby	12.968	12.550	13.540	0.323
Ruby (JIT)	13.640	13.300	14.050	0.298

Table 7: Total CPU time for writing lines to files

As shown in table 7, Go and Crystal are very fast at writing short lines to files. C takes twice the amount of time, while Ruby is 10 times slower than Crystal.

3.1.6 Writing Longer Lines to Files

Language	Mean (s)	Min (s)	Max (s)	Stddev (σ)
C	0.773	0.770	0.780	0.005
Crystal	0.657	0.650	0.660	0.005
Go	0.667	0.660	0.680	0.007
Ruby	2.085	2.050	2.220	0.053
Ruby (JIT)	2.520	2.480	2.570	0.031

Table 8: Total CPU time for writing longer lines to files

With the change of writing longer lines to the files all languages got faster as shown in table 8. Most improvements are found in Ruby and C.

3.1.7 Reading Lines from Files

Language	Mean (s)	Min (s)	Max (s)	Stddev (σ)
C	1.866	1.830	1.920	0.037
Crystal	3.026	2.970	3.090	0.040
Go	2.239	2.150	2.310	0.060
Ruby	12.909	12.440	13.180	0.241
Ruby (JIT)	13.372	12.970	13.740	0.321

Table 9: Total CPU time for reading files

When reading files C and Go are the fastest as shown in table 9, closely followed by Crystal. Ruby takes a lot more time.

3.1.8 C Bindings

Language	Mean (s)	Min (s)	Max (s)	Stddev (σ)
C	1.163	1.140	1.320	0.055
Crystal	1.173	1.170	1.180	0.005
Go	6.330	6.290	6.430	0.044
Ruby	10.913	10.620	11.480	0.264
Ruby (JIT)	12.060	11.850	12.420	0.197

Table 10: Total CPU time for calling C methods

Table 10 shows that calling C methods uses almost the same time in C as in Crystal. Both Go and Ruby take a lot more time.

3.1.9 TCP Sockets

Language	Mean (s)	Min (s)	Max (s)	Stddev (σ)
C	0.835	0.810	0.870	0.018
Crystal	0.936	0.920	0.980	0.018
Go	0.906	0.880	0.940	0.017
Ruby	0.935	0.910	0.970	0.021
Ruby (JIT)	0.957	0.920	1.000	0.029

Table 11: Measurements of listening to sockets

As shown in table 11 the differences in performance for calling TCP sockets are tiny and insignificant.

3.1.10 Comparison

Figure 1 shows that the largest difference in performance compared to Crystal can be found in Ruby, with or without JIT compilation. C is mostly faster than Crystal. Go and Crystal have a similar performance, where both languages have benchmarks where they stand out.

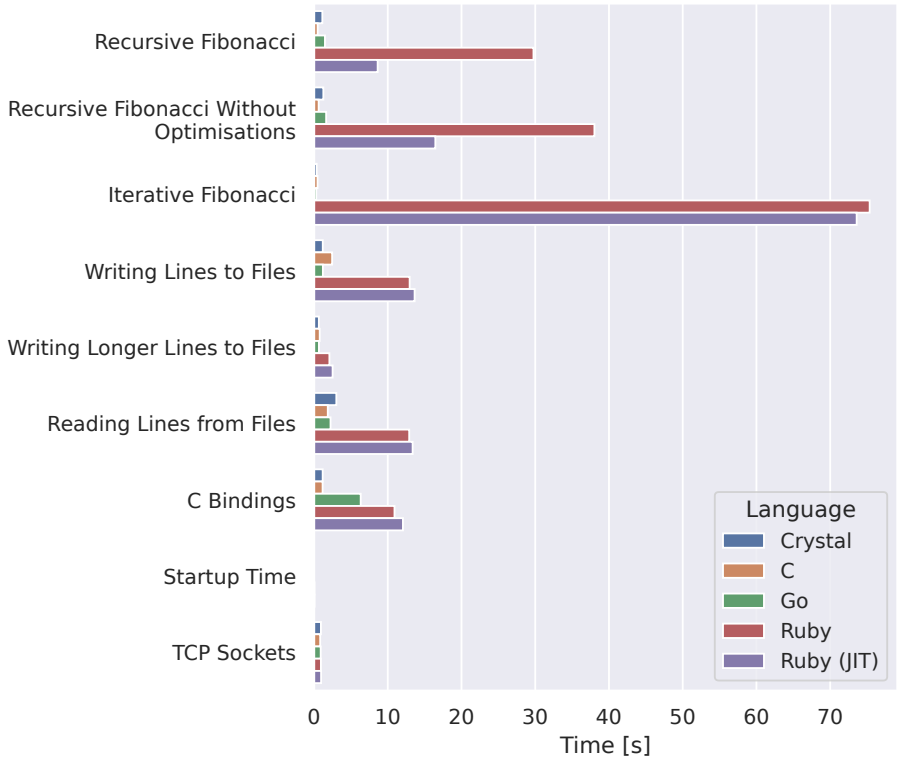

Figure 1: Comparison of all measurements

4 Discussion

The goal of this report was to compare the performance of Crystal with C, Go and Ruby. Overall this report shows that Crystal can be considered a fast language. As described in table 12 its performance is very close to Go, where it is, depending on the benchmark, sometimes slower and sometimes faster. C, especially with all optimisations of `gcc`, is still about 10 % faster than Crystal.

Measurement	**C**	**Go**	**Ruby**	**Ruby (JIT)**
Recursive Fibonacci	0.435	1.316	26.409	7.657
Recursive Fibonacci Without Optimisations	0.498	1.301	30.084	13.017
Iterative Fibonacci	1.331	0.840	211.084	206.143
Writing Lines to Files	2.072	1.013	10.934	11.501
Writing Longer Lines to Files	1.177	1.015	3.174	3.836
Reading Lines from Files	0.617	0.740	4.266	4.419
C Bindings	0.991	5.396	9.303	10.281
Startup Time	0.224	0.726	9.948	10.850
TCP Sockets	0.892	0.968	0.999	1.022
Mean	0.915	1.480	34.022	29.859
Median	0.892	1.013	9.948	10.281

Table 12: Comparison of all measurements relative to Crystal

One benchmark that stood out is the execution of C code. In this C itself was only 3.2 % faster than Crystal and both Go and Ruby were noticeably slower with a factor of 5, respectively 8. This is most likely due to the fact that Crystal is using the LLVM which can also be used for compiling C code with the Clang project [20].

Another interesting finding is, that the buffered writers used to write files in Crystal and Go have a big impact on the performance. Writing the same file in lines that are 10 times longer reduced the speed of C from 2.457 s to 0.773 s. Crystal and Go also showed improvements, but it was not as much as C.

Something unexpected was that the performance of TCP Sockets were almost the same for all languages, even Ruby. This shows, that when using sockets the bottleneck is the operating system itself. Of course when running a complex web server where each request generates multiple objects and text gets parsed then there will be differences between the programming languages, but listening to TCP sockets themselves takes a comparable time.

Ensuring that the tail-recursion could not be optimised lead to slower performance in all languages. This is expected, because more comparisons are introduced. This reduction in performance is for all languages the same, which shows that all optimisers are not able to optimise tail-recursion in this case.

4.1 Limitations

The performance of the code is immensely dependent on the implementation. Due to that fact, the comparisons need to be interpreted with caution. It is

possible that in one language the implementation is written more efficiently than in another. Another possibility is that an implementation could also be optimised by one compiler but not another. This can lead to unrepresentative results.

The `time` command only has a precision of 0.01 s. When averaging the 10 test runs the precision can be a little bit bigger, but this needs to be interpreted with caution.

It is also possible that context switches between processes or invalidated CPU caches could lead to different environments for the languages. Using an average of 10 test runs reduces the chances that this is possible, but it is still worth keeping in mind when interpreting the results.

4.2 Further Research

There are still unanswered questions after this report. Those are described in this section.

Memory Usage When comparing programming languages it is also interesting to look at the memory usage. Especially with Ruby and Crystal treating everything as objects it would be interesting to see if it is still as memory efficient as C with its primitive types.

Other Tests To verify if the tests in this report did not by accident measure something unexpected similar tests could be run. For example another recursive algorithm could be used to verify that the results are significant.

Completely different tests could be used as well, measuring other aspects of the programming languages. For example measuring objects could be interesting as well, but for this a switch from C to C++ would be necessary.

Other Compilers Another interesting aspect would be to build the C code with the Clang compiler, which also uses the LLVM. It would be interesting to see if the performance of C compared to Crystal would increase, decrease or stay the same.

Build Times One thing that was noticed during writing the test programs and measuring the performance was that the compilation time required in the different languages had noticeable differences. Crystal took significantly more time to compile than Go and C, even with these small programs. It would be interesting to see how much slower it is and what the implications for large software would be.

References

[1] "Crystal README." GitHub. https://github.com/crystal-lang/crystal/blob/d9b757adecca4a8a33fb86813cef39e3426d8006/README.md. (accessed September 26, 2020).

[2] Gouy, Isaac, "The Computer Language Benchmarks Game." Debian Salsa. https://benchmarksgame-team.pages.debian.net/benchmarksgame/. (accessed October 23, 2020).

[3] kostya, "Crystal implementations for The Computer Language Benchmarks Game." GitHub. https://github.com/kostya/crystal-benchmarks-game. (accessed November 9, 2020).

[4] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2000.

[5] M. S. Bhat, D. G. Nair, D. Bansal, and J. Vaishnavi, "Data structure based performance evaluation of emerging technologies — a comparison of scala, ruby, groovy, and python," in *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*, pp. 1–5, 2012.

[6] S. Nanz and C. A. Furia, "A comparative study of programming languages in rosetta code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 778–788, 2015.

[7] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*. Cambridge: The MIT Press, 1996.

[8] "The LLVM Target-Independent Code Generator — LLVM 12 documentation." LLVM. https://llvm.org/docs/CodeGenerator.html#tail-call-optimization. (accessed November 10, 2020).

[9] "Concurrency." Crystal. https://crystal-lang.org/reference/guides/concurrency.html. (accessed December 29, 2020).

[10] "Parallelism in Crystal." Crystal. https://crystal-lang.org/2019/09/06/parallelism-in-crystal.html. (accessed December 29, 2020).

[11] "Ractor - Ruby's Actor-like concurrent abstraction." GitHub. https://github.com/ruby/ruby/blob/master/doc/ractor.md. (accessed December 29, 2020).

[12] "Effective Go." Go. https://golang.org/doc/effective_go.html. (accessed December 29, 2020).

[13] "GNU Time — GNU Project." GNU. https://www.gnu.org/software/time/. (accessed October 26, 2020).

[14] "time(1) — Linux manual page." man(7). https://man7.org/linux/man-pages/man1/time.1.html. (accessed October 26, 2020).

[15] E. Glatz, "Privilegierte Programmausführung (Benutzer-/Kernmodus)," in *Betriebssysteme*, ch. 2.3.6, pp. 55–57, dpunkt, 2015.

[16] J. Fulmer, "Siege." GitHub. https://github.com/JoeDog/siege. (accessed November 23, 2020).

[17] "GCC, the GNU Compiler Collection." GNU Project. https://gcc.gnu.org/. (accessed December 16, 2020).

[18] "gcc(1) — Linux manual page." man(7). https://www.man7.org/linux/man-pages/man1/gcc.1.html. (accessed December 16, 2020).

[19] "go." The Go Programming Language. https://golang.org/cmd/go/. (accessed December 16, 2020).

[20] "Clang: a C language family frontend for LLVM." LLVM. https://clang.llvm.org/. (accessed December 30, 2020).