EVALUATING YONA LANGUAGE

Adam Kövári, Alexander Meduna and Zbyňek Křivka
Department of Information Systems, Faculty of Information Technology, Brno University of Technology
Božetěchova 2, 612 00 Brno, Czech Republic

ABSTRACT

The paper evaluates a new concurrent, functional programming language Yona. Specific focus is placed on the asynchronous IO aspects of this language and its current implementation. The evaluation in the later chapter serves as the demonstration of Yona's capabilities, and it helps to set the direction of further research and development in this language by identifying significant bottlenecks.

KEYWORDS

Functional Programming, Disruptor-Inspired Ring-Buffer, Non-Blocking IO, Truffle Framework, GraalVM JVM, Benchmarks

1. INTRODUCTION

Yona is a high-level dynamic functional programming language with a strong focus on non-blocking concurrent computation. Yona has a rich runtime system, immutable data structures, concurrency model inspired by the LMAX disruptor (Thompson, et al., 2011) and JIT compilation and interoperability with other languages on the GraalVM platform (Würthinger, et al., 2013). Yona language is implemented using the Truffle framework (Würthinger, et al., 2017), provided by the GraalVM, which allows the implementation of interpreters that can use JIT capabilities on this VM. This paper explains the fundamental design decisions of the implementation and testing of this language; it also presents some initial set of results.

1.1 Design Goals

Yona hides the complexity of concurrent programming in its runtime. The concurrency system of Yona wraps future values in a Promise [1] -like structure (Liskov & Shrira, 1988), executes them in a disruptor-inspired ring-buffer, and then unwraps actual values whenever it becomes available, all this hidden on the runtime level. This optimization prevents the programmer from seeing any difference in values that have been computed or are yet to be computed in the future. It does not expose any low-level threading and since it contains only immutable data structures, nor it needs any synchronization primitives. Yona contains highly optimized immutable built-in data structures, including `Set`, `Dict` (Steindorfer & Vinju, 2015), and `Seq` (based on Finger Trees (Hinze & Paterson, 2006)), which eliminate concurrent mutation type of errors. Advanced concurrency in Yona can be implemented using the built-in Software Transactional Memory (STM) module (Fernandes & Cachopo, 2011). In addition to these features, Yona is a powerful functional language, with advanced pattern matching (Ramesh & Ramakrishnan, 1992), tail-call optimization, first-class module support, resource management, enabling programmers to write efficient programs in a very high-level style.

[1] Promise represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.

1.2 Short Syntax and Semantics Guide

Full syntax and semantics of the language can be found on the language website[2]. However, we provide one example here that shows the most relevant pieces of the syntax and semantics of the language, necessary to understand the next section about the socket implementation.

```
try
  let
    keys =
      with File::open "keys.txt" {:read} as keys_file
        File::read_lines keys_file
      end

    values =
      with File::open "values.txt" {:read} as values_file
        File::read_lines values_file
      end
  in
    Seq::zip keys values |> Dict::from_seq
catch
  (:ioerror, _msg, _stacktrace) -> {}
end
```

Figure 1. Zipping keys and values read from two files concurrently in Yona

This program in Figure 1 reads two files, keys.txt and values.txt, in parallel and in a non-blocking way (no thread is blocked), zipping lines read, producing a dictionary of these keys and values as a result.

As diving deeper into this process, the following actions take place:

- Because `let` expression is used, Yona will perform a static analysis of dependencies between individual aliases[3] defined within the scope of these expressions, `keys` and `values` in this case. Since they do not depend on each other, they **may** be executed in parallel - they are put into two independent buckets of tasks.
- No other aliases are defined; thus, buckets of tasks begin their execution. The task from the first bucket begins execution - lines are being read from the keys.txt file. Function `Files::read_lines` is implemented as a non-blocking function in the standard library. It returns an underlying promise value immediately and puts a task to read lines into the runtime buffer of tasks. Promises are fully transparent to the programmer, and they do not need to be aware of this, as it is only a runtime type.
- Since `File::read_lines` returned, the next task from the second bucket may begin to be processed. Like before, this task reads lines in the values.txt file, independently in a non-blocking way, returning immediately.
- In this example, no further aliases are defined, so the body[4] of the `let` expression may be processed now. The body is processed after both aliases, `keys` and `values` become available. Function `Seq::zip` takes both of them, zipping keys with values, producing a sequence of tuples then passed to function `Dict::from_seq`, which produces the final dictionary. The `let` expression needs to wait for aliases defined within its scope to be ready but it does not mean that the thread executing this `let` expression is blocked. In fact, if this `let` expression was nested in some other expression or returned as a result of a function, there could be other computations being executed in the same thread, while waiting for the result of this particular `let` expression.

[2] http://yona-lang.org/ - Language description, standard library documentation, homepage
[3] Because there are no mutable variables in Yona, names referring to value will be called "aliases". They could be seen as "final" or "constant" variables in other languages
[4] The body of the `let` expression is an expression following the `in` keyword

This relatively simple example shows the execution model in Yona. Hopefully, it demonstrates how easy it is to write non-blocking, concurrent programs in Yona, without any explicit interaction from the programmer to make it so. The programmer only needs to use the standard library, and the runtime takes care of all the underlying concurrency implementation details.

1.3 Implementation of Files

All file operations in Yona are implemented in a non-blocking way. In the case of files, the underlying runtime uses Java NIO2 to implement read/write operations (Ganesh & Sharma, 2013). From the programmers' perspective, a file is represented as a file context manager[5], used by the read and write functions from module `File`. Function `open` creates this context manager. For example, see Figure 1.

File operations in Yona are implemented as an abstraction on top of Java NIO2 `AsynchronousFileChannel`, a callback-based API for Java, and it internally uses Java Executors to execute the non-blocking operations. Future versions of Yona intend to remove this level of indirection and bypass any use of Java Executor, which has a similar purpose to the Yona's disruptor.

1.4 Implementation of Sockets

This section describes the implementation of TCP Sockets in Yona's standard library. Socket IO implementation, same as File IO, is significant in the context of Yona, since one of the main focus areas of this language is non-blocking IO. Sockets in Yona use underlying Java NIO socket infrastructure.

Implementation of TCP Sockets consists of three modules: `socket\tcp\Server`, `socket\tcp\Client` and `socket\tcp\Connection`. These modules provide functionality for opening channels, accepting clients, making client connections, and reading and writing to sockets.

In addition to these three modules, there is additional infrastructure in the runtime that supports the non-blocking nature of the socket modules. Specifically, there is an NIO Selector thread, which polls for changes in socket states, such as that socket becomes acceptable, connectable, readable, and writable. Once any of these events happens, the runtime will look for an appropriate request[6], to fulfill and then once its work is done, that particular request gets completed, and the program is resumed to continue handling the obtained data.

1.4.1 TCP Server

Module `socket\tcp\Server` has two functions creating context managers for TCP channels and connections. Function `channel` creates new TCP channels and returns a context manager that is used when accepting new clients.

```
with socket\tcp\Server::channel (:tcp, addr port) as channel
    infi (\-> accept channel) # accept new connections in an infinite loop
end
```

Function `accept` accepts a client connection as soon as it becomes ready.

```
with daemon socket\tcp\Server::accept channel as connection
    # deal with the accepted connection
end
```

[5] Context managers are Yona's way for managing resources in a consistent way, so that resources are closed as soon as they are not needed anymore.
[6] We are using term "request" here, since it seems more proper in the context of sockets, however in reality, it is just a runtime promise, same as any other in context of Yona

The example above uses a "daemon" context manager, which causes Yona to wait only until the connection is made, but not until the whole body of this `with` expression is processed, before returning the result of the `with` expression. This allows Yona to handle connections concurrently, since as soon as the connection is accepted, its handling is moved to the background. The use of the `with` expression still makes sure that resources related to this connection are disposed after the work on this connection has finished.

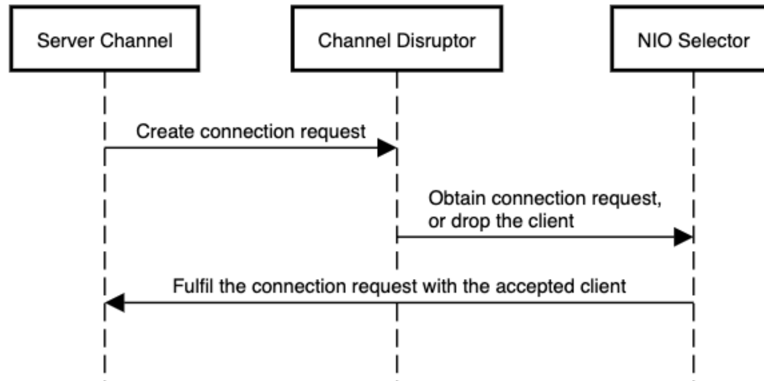

Figure 2. Flow of accepting a connection on an open TCP channel

The diagram in Figure 2 visualizes how a connection request is made and when it is fulfilled. The whole process is asynchronous, and server code creates requests, or promises, that are fulfilled once connection is ready. This way, the program is not blocked, and it can do some other processing without waiting for the result of the `accept` operation.

1.4.2 TCP Client

This module provides function `connect` that creates a TCP connection to a server, represented as a context manager.

```
with socket\tcp\Client::connect "localhost" 5555 as connection
    # read/write from and to the server
end
```

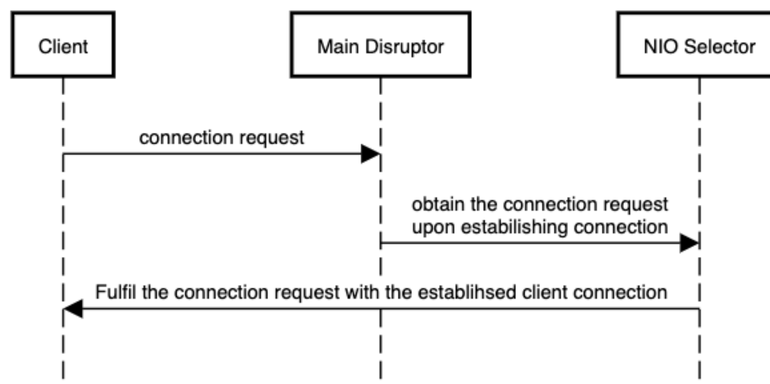

Figure 3. Flow of making a client connection to a server

Figure 3 above shows how a client connection is made in a non-blocking way. Function `connect` will produce a connection request in the background and only once the client connection is established, the connection request is fulfilled, asynchronously, by the NIO Selector thread.

1.4.3 Connection Read/Write Operations

Module `socket\tcp\Connection` contains functions for reading and writing on TCP connections. These functions work for both client and server connections.

```
socket\tcp\Connection::write connection "hello"
socket\tcp\Connection::read_until connection (\b -> b != 10b) # read until LF
```

These functions create a read or write request in the connections read or write disruptor-based queue (`NIOQueue`).

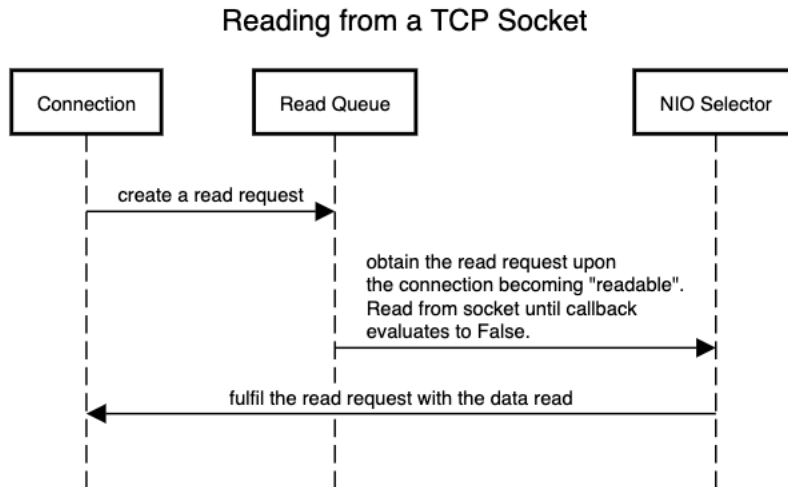

Figure 4. Reading from a TCP Socket

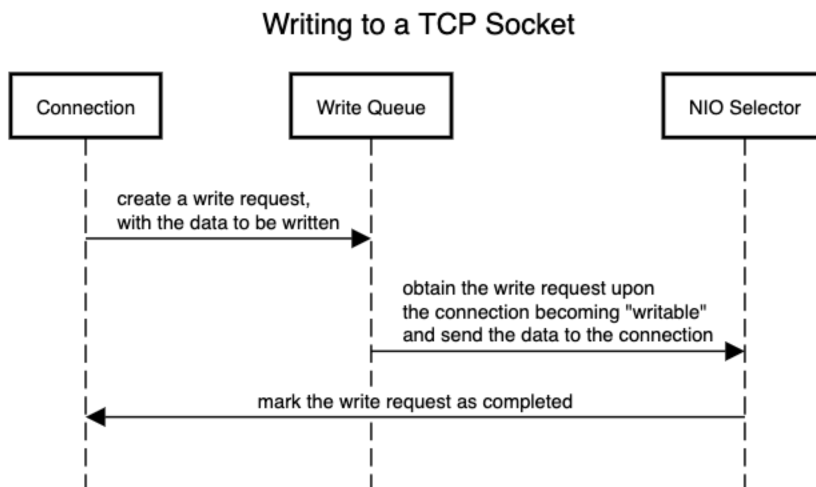

Figure 5. Writing to a TCP Socket

The read/write requests are bound to a specific connection instance (context manager), where they are buffered in their own disruptor queues. In this way Yona passes data between the NIO Selector thread and the main program.

2. EVALUATION

This chapter describes a set of benchmarks developed to discover significant bottlenecks and help drive further research and development. Yona is still under active development, including the interpreter, and runtimes, and the standard library.

Choice of the benchmarks reflects the current priorities of Yona implementation, which is the concurrency and non-blocking IO. The first three tests are well-known benchmarks, used to evaluate various languages and concurrency models, and the last benchmark is a combination of the previous ones. It provides an additional level of complexity, and its purpose is to determine if combining different types of IO (network and file), would cause any unexpected performance issues.

2.1 Method

We have measured the performance of several simple benchmarks and compared them with other major programming languages using comparable approaches[7]. Algorithms selected for these benchmarks are on purpose not the most efficient algorithms to solve a particular problem, but such algorithms that could meaningfully benchmark various implementation aspects of different languages (e.g. interpreter performance, IO performance, or standard library efficiency). Languages used to compare with Yona in these benchmarks are Python, JavaScript and Erlang. That is because Python and JavaScript are the most popular dynamic languages, while Erlang is a dynamic, concurrent and functional language and Yona as well.
Tests were performed on 64-bit Linux, Intel i5-9600K, 16 GB of memory.

2.1.1 Echo Server

Th first benchmark is a simple non-blocking TCP echo server. The purpose of this benchmark is to detect whether the implementation of non-blocking sockets in Yona's standard library has any significant bottlenecks. This benchmark is not algorithm-heavy, and it depends on the IO performance. Nevertheless, if Yona performed significantly worse than other languages, it would point to particular inefficiency in the standard library functions implemented for Yona. The results are in Table 1.

Table 1. Simple echo server benchmark

Echo Server (connections / seconds to process)	100	1000	10.000
Node 16	0.248	2.625	126.06
Python 3.9	0.254	2.61	126.06
Erlang 23	0.244	2.567	126.62
Yona 0.8.1	0.26	2.945	130.23

This benchmark tests the throughput by creating 100, 1000 and 10,000 concurrent clients connecting to a simple echo server written in respective languages. The client is a simple socket client written in Rust to minimize its system footprint. Time was measured by running the following command:

```
time parallel -n0 -j <NoC> ./echo-client ::: {1..<NoC>}
```
[8]

where `<NoC>` is the number of clients/connections 100; 1000 or 10,000. While Yona was slightly a bit slower, it was not a difference of a significant magnitude. This test suggests that the socket implementation of a TCP server in Yona has no serious bottleneck.

[7] The source code of all benchmarks performed in this paper is available at the Yona git repository: https://github.com/yona-lang/yona/tree/master/benchmarks

[8] The three colons syntax of the GNU Parallel to specify number of processes to execute. Argument `-j` specifies the number of concurrent processes, and `-n0` means that the `echo-client` process has no arguments.

2.1.2 Bubble Sort

The second benchmark is focused on algorithmic performance and can help detect potential issues with the interpreter performance. Since the algorithm implemented in this benchmark is not tail-recursive, it will suffer from stack overflow for larger inputs. Bubble sort is a well-known algorithm with the worst-case complexity of O(n2), where n is the number of items, so it can indicate performance issues in the language interpreter.

Table 2. Bubble Sort benchmark

Bubble Sort (numbers / microseconds)	10	100	200	300
Erlang 23	19	1,172	3,818	7,771
Node 16	119	6,387	19,659	55,703
Python 3.9	56	7,592	41,902	204,889
Yona 0.8.1	77,380	1,066,452	2,848,917	6,217,559

The results demonstrate the exponential complexity growing with the size of the input. The input contains a list of random integers. It is very clear from this benchmark that the performance of Yona is significantly, several magnitudes of worse than that of other languages. The exact cause of this performance issue is not yet known at the time of writing this article. Solving this bottleneck will be crucial in the next development of Yona.

2.1.3 Read Lines

This benchmark is designed to evaluate the performance of non-blocking file IO. Similar to the Echo Server benchmark, the purpose of this test is to determine whether Yona contains any bottleneck in its standard library module for reading files. Since Python does not contain this functionality in its standard library, a third-party library `aiofiles`[9] was used to achieve the same functionality as other languages. This benchmark reads a large file with 128,457 lines, working in line-by-line way.

Table 3. Read Lines benchmark

Read Lines (lines / microseconds)	128,457
Node 16	65,955
Erlang 23	533,595
Python 3.9	6,649,067
Yona 0.8.1	2,611,637

This result clearly points to an inefficiency in the Python's third-party library. While Yona's performance is slower than the one in Node and Erlang, the difference is not even one magnitude large. The result of this benchmark suggests that there is a room for improvement in Yona, the exact details of which are yet to be determined in future work.

2.1.4 SCP Server-Client

This benchmark is a combination of the Echo Server and the Read Lines tests. It is a simple server-client application, where the server reads input from the client, line-by-line and writes it to a file, and the client reads a file, line-by-line and sends it to the server. This test was chosen to detect possible issues when combining non-blocking file and socket operations in Yona. It is a more complex, real-world application testing the benefits of non-blocking IO.

[9] https://pypi.org/project/aiofiles/0.6.0/

ISBN: 978-989-8704-34-4 © 2021

Table 4. SCP benchmark

SCP (server language/ microseconds)	SCP (client language / microseconds)	
Erlang 23	Yona 0.8.1	8,381,385
Yona 0.8.1	Yona 0.8.1	9,589,741
Yona 0.8.1	Python 3.9	11,546,680

For the sake of comparing different server implementations as well, there is an Erlang and Yona server implementation. In this case, the client was Yona, in both runs. There is a difference between Yona and Erlang as a server, the difference is 13%. In case of using a different client implementation, Python performed about 15% worse than Yona. This benchmark shows that the non-blocking IO implemented in the standard library of Yona performs roughly similarly than that in other popular programming languages.

3. CONCLUSION

The tests performed during this evaluation were designed to test the capabilities of Yona, primarily in the scope of its concurrent and non-blocking nature. Yona's performance in these tests is in line with other mainstream programming languages. The Bubble sort test is focused more on CPU performance and clearly indicates a severe bottleneck in Yona interpreter or built-in data structures. Solving this bottleneck will likely improve performance in other areas as well, but solving it must become a top priority for the subsequent work on this programming language.

Yona provides a higher level of abstraction than all other languages it was compared with, and even though is still in its early days of development, it shows very interesting performance in key areas of concurrency, which has been the primary focus area of the development so far. Focusing on the interpreter performance and addressing the bottlenecks in the CPU-bound algorithms will be crucial to allow Yona to become competitive with other long-established and highly optimized languages and runtimes. We hope the method and results of our evaluation of Yona will push for further research and improvements to this language and can be used as a basis for future comparisons against comparable languages and platforms.

ACKNOWLEDGEMENT

This work has been supported by the Czech Science Foundation, project No. 19-24397S and the BUT grant FIT-S-20-629.

REFERENCES

Fernandes, S. M. & Cachopo, J., 2011. *Lock-Free and Scalable Multi-Version Software Transactional Memory.* New York, NY, USA, ACM, p. 179–-188.

Hinze, R. & Paterson, R., 2006. Finger Trees: A Simple General-purpose Data Structure. *J. Funct. Program.,* 3, Volume 16, p. 197–217.

Ramesh, R. & Ramakrishnan, I. V., 1992. Nonlinear Pattern Matching in Trees. *J. ACM,* 4, Volume 39, p. 295–316.

Ganesh, S. G. & Sharma, T., 2013. Java File I/O (NIO.2). In: *Oracle Certified Professional Java SE 7 Programmer Exams 1Z0-804 and 1Z0-805: A Comprehensive OCPJP 7 Certification Guide.* Berkeley(CA): Apress, p. 251–280.

Liskov, B. & Shrira, L., 1988. *Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems.* New York, NY, USA, ACM, p. 260–267.

Steindorfer, M. J. & Vinju, J. J., 2015. *Optimizing Hash-array Mapped Tries for Fast and Lean Immutable JVM Collections.* New York, NY, USA, ACM, p. 783–800.

Würthinger, T. et al., 2013. *One VM to Rule Them All.* New York, NY, USA, ACM, p. 187–204.

Würthinger, T. et al., 2017. *Practical partial evaluation for high-performance dynamic language runtimes.* New York, NY, USA, ACM, p. 662–676.

Thompson, M. et al., 2011. *Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads.* [Online] Available at: https://lmax-exchange.github.io/disruptor/disruptor.html