CoFaaS: Automatic Transformation-based Consolidation of Serverless Functions

Truls Asheim
truls.asheim@ntnu.no
Norwegian University of Science and Technology (NTNU)
Norway

Magnus Jahre
magnus.jahre@ntnu.no
Norwegian University of Science and Technology (NTNU)
Norway

Rakesh Kumar
rakesh.kumar@ntnu.no
Norwegian University of Science and Technology (NTNU)
Norway

ABSTRACT

The attractive property of decoupling deployment decisions from application development has led to fast adoption of the Function-as-a-Service (FaaS) cloud computing model. FaaS applications are typically highly modular with each function having a specific purpose and well-defined interface as this enables code reuse, simplifies maintenance, improves testability, and provides language independence. To enable these attractive features, FaaS applications often use Remote Procedure Call (RPC) interfaces for inter-function communication — which comes at the cost of orders of magnitude higher latency than native function calls. Prior works that alleviate this overhead are unattractive because they either require non-standard APIs, only support a single language, or rely on specialized languages or runtimes.

Our key insight is that we can exploit the well-defined RPC interfaces to perform code transformations that alleviate inter-function communication overhead. We hence propose CoFaaS which leverages this insight to consolidate FaaS functions on top of a WebAssembly runtime and thereby avoid accessing the network layer when functions are deployed on the same compute node. Our evaluation shows that this strategy is highly effective and reports that CoFaaS reduces inter-function communication latency by up to 100× and application-level request round-trip time by up to 6×.

1 INTRODUCTION

Function-as-a-Service (FaaS) computing is becoming an increasingly popular cloud computing model that simplifies the cloud application lifecycle by moving critical decisions about application deployment from the developer to the provider. The unit of composition in the FaaS model is a *function* which is triggered based on specific events, such as an incoming HTTP request. A function usually has a single well-defined purpose and interface as this promotes code reuse, simplifies maintenance, and enhances testability [21]. Larger FaaS *applications* are built by composing individual functions.

A key advantage of the FaaS computing model is that applications can be composed of functions written in any language. Language independence is attractive for three reasons. Firstly, it allows a function to be written in the language that supports the purpose of the function in the best possible way. Secondly, it allows applications to be gradually rewritten and ported to other languages function by function. Finally, it allows applications to continue relying on tested and stable units of functionality even if the implementation language of other parts of the application changes.

When developing traditional monolithic applications, combining languages is typically cumbersome because it requires crufty low-level foreign function interfaces that are complex and error-prone to use. In FaaS, programming language interoperability is achieved by design. This is because FaaS functions use highly abstract and language-independent Remote Procedure Call (RPC) interfaces to communicate across a network fabric, for example Google's gRPC [9]. Enabling functions to communicate, regardless of language and internal implementation details, fundamentally requires that external interfaces are specified in a formal, language-independent, and declarative manner. Modern RPC interfaces hence provide Interface Definition Languages (IDLs) to specify the contract of each function, i.e., encoding the specifics of the calls that the function supports. Code generators then take the IDL definition as input and outputs a concrete function interface.

Unfortunately, FaaS' attractive properties of simplified deployment, language-independence, and modular design come with a severe performance penalty, in part due to its reliance on RPC for inter-function communication. Whereas a local function call in a monolithic application incurs a latency of less than a single microsecond, an RPC call in a FaaS application can incur a latency of several milliseconds — a difference of several orders of magnitude. When a client issues a RPC request, the request is first serialized to the RPC wire format, then the serialized request is sent across the network, and on the receiver's side, the request is deserialized and processed. Each of these steps takes time, however, the primary contributor to the overhead of issuing an RPC request comes from accessing the network transport layer. This inter-function communication starts to dominate the end-to-end latency of real-world applications with deep function chains and frequent RPCs where the communication latency has to be paid multiple times.

Alleviating inter-function communication overheads in FaaS applications is hence critically important, and a rich body of prior work has investigated this issue [8, 10, 12, 13, 17, 18]. While these proposals effectively reduce communication latencies, they do so by relinquishing one or more of the properties that make FaaS attractive in the first place. In particular, they either 1) provide a custom and non-portableAPI demanding that existing functions need to be rewritten, 2) restrict FaaS applications to be implemented in a single language or 3) propose entirely new FaaS programming models or runtime environments that do not readily integrate with the current cloud computing platforms. Commercial offerings that aim to effectively chain function invocations such as AWS Step Functions [7] and Knative Eventing [11] require developers to adhere to platform-specific APIs for communication. This limits seamless function portability across applications and imposes a vendor lock-in on applications.

There is hence a need for an approach that reduces inter-function communication overhead while using standard APIs, retaining language independence *and* being easily integratable with current cloud computing platforms — and our goal in this work is to provide such a system. Towards that end, we make a key insight that inter-function RPC communication interfaces are statically defined. More specifically, the developer specifies the interface of each function using an established IDL which means that we have significant leeway in generating function interfaces while respecting the IDL specification. In particular, accessing the network layer — which we demonstrate is the key contributor to inter-function communication overhead — is entirely unnecessary when the functions are deployed on the same compute node.

We hence propose CoFaaS which automatically consolidates FaaS functions — while respecting the semantics of the function's IDL specification — and thereby completely avoids accessing the network layer when functions are scheduled on the same node. More specifically, CoFaaS first transforms the IDL description used by the target RPC interface to WebAssembly Interface Types (WIT). This enables us to leverage the WebAssembly (Wasm) ecosystem and co-locate FaaS functions on a single Wasm runtime; each function is hosted in its own container to maintain isolation. In this way, CoFaaS uses the Wasm runtime to provide inter-function communication and thereby avoids accessing the network layer when functions are deployed on the same node. Our evaluation shows that this strategy yields significant speedups. CoFaaS reduces inter-function communication latency by up to 100× and application-level request round-trip time by up to 6×. The overhead of applying CoFaaS in production scenarios is minimal because it is fully automatic and only slightly increases compilation and deployment times.

To summarize, we present the following key contributions:

- We observe that the well-defined interface definitions of existing FaaS applications provide significant leeway in how to generate function interfaces and that this, in turn, can be leveraged to implement powerful, automated transformations of FaaS applications.
- We exploit the above insight to design CoFaaS, the first automatic function transformation framework that significantly reduces inter-function communication latencies while being standard API compliant, language-independent *and* compatible with current cloud computing platforms.
- We demonstrate that CoFaaS yields significant speedups, i.e., it reduces inter-function message latency by up to 100× and application request round-trip time by up to 6×.

2 BACKGROUND ON WEBASSEMBLY

WebAssembly (Wasm) [6] is a portable bytecode format that originally targeted delivering high-performance compiled applications to web browsers. However, since none of Wasm's features explicitly targets web pages, it is also suitable as a general purpose bytecode format. The introduction of the WebAssembly System Interface (WASI) gives Wasm POSIX-like capabilities and allows Wasm programs to run outside the browser. A particularly appealing property of Wasm is the increasingly large number of programming languages that supports it as a target.

The WebAssembly Component Model [2] is a recent addition to the Wasm ecosystem that turns a compiled Wasm module into a portable, embedded, and composable *component* with a formally defined external API. A goal of the component model is, therefore, to allow developers to build language-independent applications that are composed of several isolated components.

Following this definition, the scope and purpose of a Wasm Component in this regard is the same as a FaaS function. The Wasm Component mode even comes with its own dedicated IDL known as WebAssembly Interface Types (WIT). A key observation, in this regard, is that IDL's that are used to specify the interfaces of FaaS functions, such as Protobuf for gRPC, are easily mapped to WIT.

Currently programs compiled from Rust, C/C++, Go, and Java are supported by the Wasm component model, but there is no limitation that prevents further languages from being supported [5].

3 COFAAS

To describe CoFaaS, we will use a simple two-function FaaS application, called ProdCon (Producer-Consumer). The application is shown in Figure 1, and its functionality is simple:

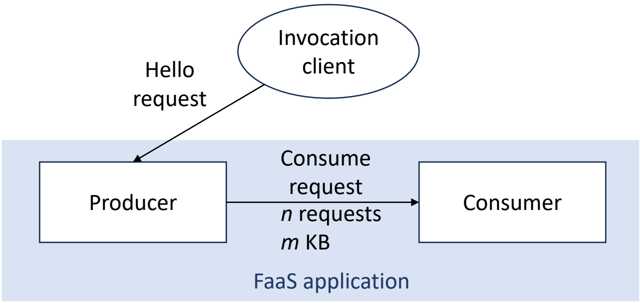


Figure 1: FaaS application used for evaluating CoFaaS

when the Producer function receives a request from the Invocation Client it sends one or more requests to the Consumer function. The size of the request can be varied and it does not need to have any particular structure.

3.1 IDL mapping

As mentioned in Introduction, a FaaS function must have a contract that specifies how it interacts with the surrounding world. For this purpose, it is common to use Interface Definition Languages that formally define a function's inputs and outputs. An IDL is purely declarative, a key reason for their language-independence, and therefore they rely on code generators to be turned into a usable interface. In practice, several different IDLs are in use by real-world FaaS applications, two notable examples being Protobuf [16], used by Google's gRPC [9], and Apache Thrift [1]. Currently, CoFaaS only supports gRPC but there is no fundamental limitation preventing other IDLs from being supported. In the following, we will detail how the transformation from Protobuf to Wasm's WIT is done.

Figure 2 shows an example of how two gRPC interfaces, marked by the blue and green shades, are transformed into their equivalent WIT definitions. Referring to the ProdCon application of Figure 1, the Producer function *exports* the `helloworld` protocol and *imports* the `prodcon` protocol. The Consumer function only exports the `prodcon` protocol. The interfaces exported by a function can be called by other functions whereas the interfaces imported by a function can be used to call another function exporting the same interface.

In Figure 2 (left), the gRPC interface of `prodcon` consists of two *messages*: `ConsumeByteRequest` and `ConsumeByteResponse`. Additionally, it defines a single service `ProducerConsumer` that defines a single method named `ConsumeByte`. To put this in the terminology of a conventional program, a function exporting the `prodcon` interface will expose a public API containing the `ConsumeByte` function that takes a struct of type `Consume-ByteRequest` as an argument and returns a struct of type `ConsumeByteReply`.

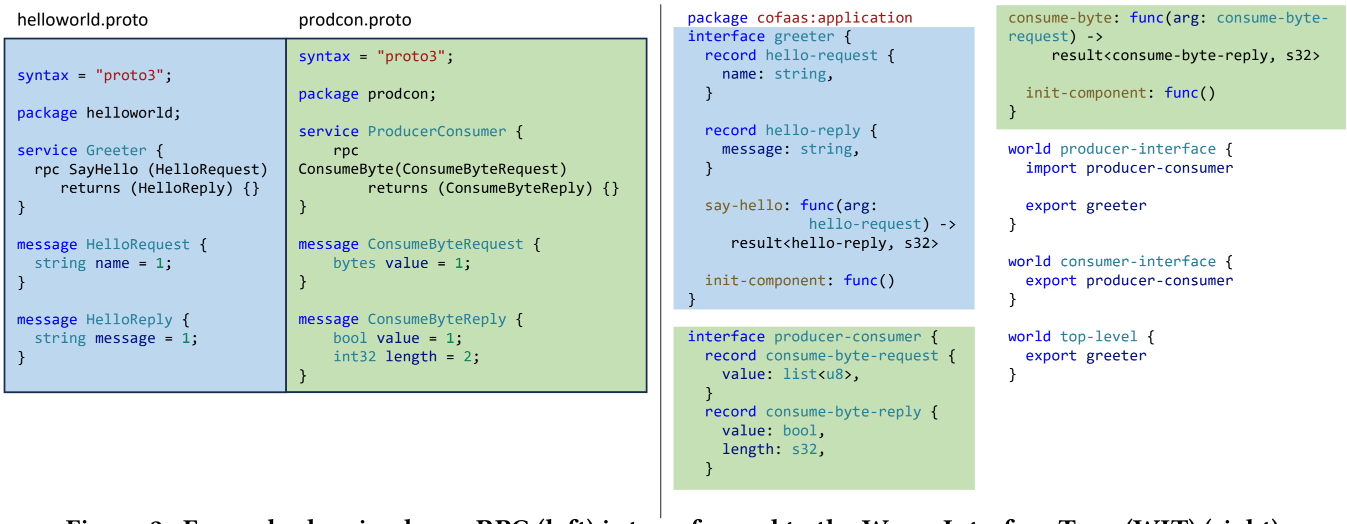
Next, we describe how the WIT interface on the right is generated. The WIT generator takes all of the protocols used by functions in an application and does the following: a) it maps each protocol to a WIT *interface* and b) it generates a WIT *world* for each function in the application. Further, it generates a world called `top-level` that defines the public interface of the whole application. A WIT world represents the entire public interface of a function, as such, we see that the world corresponding to the producer function, `producer-interface`, imports the `producer-consumer` interface enabling the Producer function is able to call the Consumer function. It exports the `greeter` interface as this is the external interface of the application.

For each interface generated, the WIT notion of a `record` corresponds to a `message` in gRPC and services are represented as a sequence of function definitions. We also note the addition of a `init-component` function. This method calls the `main` method of the original FaaS functions to perform necessary initialization of the function. When a CoFaaS-transformed application is loaded, all of its comprising functions chain-calls their `init-component` methods.

3.2 FaaS Function to CoFaaS Component

The CoFaaS transformation takes each FaaS function and transforms it into a different, but analogous, a CoFaaS component. As Figure 3 shows, this transformation enables the functions running in separate containers in native FaaS to share a runtime on the same host in WebAssembly. To ensure that we preserve the semantics of the original functions, CoFaaS seeks to change the implementation code of the original functions as little as possible. However, since the original code of the FaaS functions 1) expects to run as a separate server process that listens to a network request and 2) use the API generated by the gRPC code generator for communicating with the outside world, we can not entirely avoid minor changes to the application code.

The code generated by the default gRPC generator uses networked RPC calls for communication among functions. When transforming a FaaS function to a CoFaaS component, we eliminate and replace this RPC-backed communication. At the same time, to minimize the changes to the original code, we also need to maintain compatibility with the API produced by the gRPC code generator. To achieve this, we implement a custom gRPC code generator that generates code conforming to the same API as the gRPC-generated code but replaces network-backed RPC calls with local function calls. Additionally, the interface code that we generate contains a small wrapper that translates between the data structures passed from the Wasm component call and the gRPC-defined data structures that are used by the implementation code.



helloworld.proto

```
syntax = "proto3";

package helloworld;

service Greeter {
  rpc SayHello (HelloRequest)
     returns (HelloReply) {}
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

prodcon.proto

```
syntax = "proto3";

package prodcon;

service ProducerConsumer {
    rpc
ConsumeByte(ConsumeByteRequest)
        returns (ConsumeByteReply) {}
}

message ConsumeByteRequest {
    bytes value = 1;
}

message ConsumeByteReply {
    bool value = 1;
    int32 length = 2;
}
```

```
package cofaas:application
interface greeter {
  record hello-request {
    name: string,
  }

  record hello-reply {
    message: string,
  }

  say-hello: func(arg:
                hello-request) ->
     result<hello-reply, s32>

  init-component: func()
}

interface producer-consumer {
  record consume-byte-request {
    value: list<u8>,
  }
  record consume-byte-reply {
    value: bool,
    length: s32,
  }
```

```
consume-byte: func(arg: consume-byte-
request) ->
     result<consume-byte-reply, s32>

  init-component: func()
}

world producer-interface {
  import producer-consumer

  export greeter
}

world consumer-interface {
  export producer-consumer
}

world top-level {
  export greeter
}
```

Figure 2: Example showing how gRPC (left) is transformed to the Wasm Interface Type (WIT) (right).

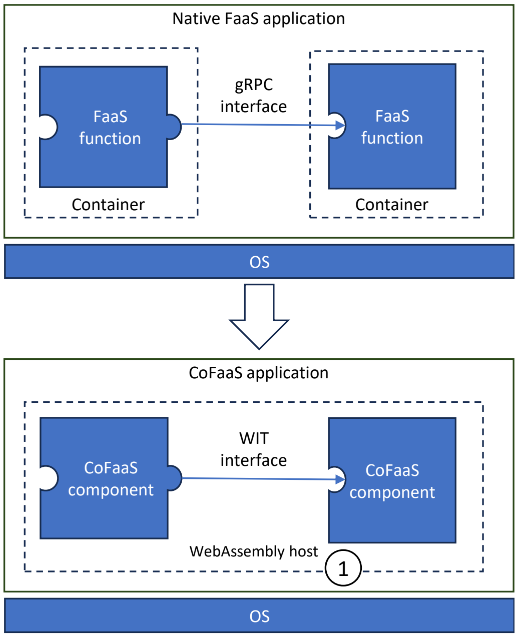


Figure 3: CoFaaS transformation.

Finally, we need to deal with now redundant library calls. For example, a function using gRPC for communication uses the following code for initializing a connection to a server.

```
conn, err := grpc.Dial(addr, grpc.WithBlock(), [...])
[...]
client := pb_client.NewProducerConsumerClient(conn)
```

The resulting `client` object holds a interface that can be used to communicate with the Consumer process in Figure 1. The first call to `grpc.Dial` initializes a network connection that is used as the transport for the RPC call. In our case, initialing this networking connection is not needed. As we aim to minimize the changes to the original function code, rewriting it to remove the call to `grpc.Dial` is not an attractive option. Therefore, we introduce our own stubbed gRPC library replacement that provides a simple nop implementation of the `Dial` function defined as follows

```
func Dial(target string, opts ...interface{})
        (*ClientConn, error) {
    return &ClientConn{}, nil
}
```

This library stubbing technique allows us to change the behavior of the function without directly re-writing critical parts of its code. To make the code use our stubbed library, we simply change the gRPC import of the client code to use it instead of the standard one. We also provide a stubbed version of the built-in Go `net` library.

Currently, we always replace imports of these libraries with our stubbed versions. For cases where all gRPC calls are transformed to local CoFaaS calls this works well. However, functions may want to open network connections for other reasons. To support this, one can add a code analysis step to the code transformation that determines which libraries the specific gRPC calls are related to and only replaces the ones that we want to turn into local CoFaaS calls.

The final step is to generate the Wasm host wrapper ① that hosts a Wasm runtime and will load and run the CoFaaS application. This host wrapper exposes a gRPC interface corresponding to the public interface of the FaaS application. When requests are received by this wrapper, they are transformed into a call to the WIT bindings of the first function of the CoFaaS application.

3.3 Putting it all together

In summary, following are the steps for applying the CoFaaS transformation to a FaaS application. We emphasize that all of these steps are completely automated.

(1) The WIT code generator is invoked to transform the gRPC protocols used by the FaaS functions to a WIT interface describing the entire application Section 3.1
(2) For every FaaS function, we generate bindings for its corresponding WIT world and use our custom gRPC code generators to generate a compatibility layer between the gRPC API used by the function implementation and the WIT bindings used for communicating within the CoFaaS application
(3) Then, we apply a set of transformations to the implementation code of the function. These transformations make the code use the replacement gRPC API that we generate and make the code use our stub libraries to disable undesired functionality
(4) Finally, we generate the Wasm host wrapper that exposes the external interface of the FaaS application over gRPC and loads and runs the CoFaaS application.

To add support for a new language in CoFaaS, only steps 2 and 3 above need to be re-implemented. Steps 1 and 4 are generic and doesn't change regardless of the function implementation language. Current, CoFaaS supports applying this transformation to functions written in Go automatically. For Rust, we manually rewrite function code in a way that is identical to the automated process.

4 EVALUATION

4.1 Methodology

The benchmarks were executed on an Intel Xeon E3-1275 v6 with 4 HT cores clocked at 3.8 GHz with 64 GB of RAM and SSD drives running Fedora 37. We disabled frequency scaling, turbo boost and swap space during the experiments.

We are using a two-process configuration, depicted in Figure 1. For every client request, the producer process sends n requests of m KB to the consumer process. We evaluate values of 1, 10 and 20 for n and, where possible, values from 2^0 to 2^9 for m. CoFaaS only optimizes the inter-function communication latency, i.e., the latency occurring when the producer process in Figure 1 calls the consumer process. Therefore, adjusting the value of n (request repetitions) allows us to estimate the performance impact of CoFaaS on larger applications that perform a variable number of inter-function requests. We refer to requests between different functions within an application as *inter-function* requests. The application and invocation client are adapted from [20].

We evaluate two functionally equivalent implementations of the application in Figure 1 written in the languages Go [3] and Rust [4], respectively. The key distinction between the two languages is that Go relies on a managed runtime for memory management, i.e. Garbage Collector (GC), whereas Rust inserts compile-time instructions to handle memory. Since languages using managed runtimes have different performance characteristics and platform requirements than unmanaged languages, our experiments demonstrate that the CoFaaS is effective in both cases.

Compilers, with Wasm targets other than web browsers, need to support the WebAssembly System Interface (WASI). For Rust, Wasm support is generally good and its compiler supports a usable WASI target. However, the WASI target of the mainline Go compiler is currently not supported by the Wasm component model. Instead, we are limited to using the TinyGo compiler which primarily targets small embedded devices. Therefore, it is positioned at a different design point than a compiler targeting general-purpose systems. In particular, its garbage collector implementation is optimized for code size rather than speed causing its GC performance to trail the mainline Go compiler [19]. Because of this, our example application implemented in Go is heavily penalized when executed on Wasm. Therefore, we disable GC for our Go benchmarks. Naturally, this limits how long we can run our benchmarks for and the payload sizes we can use. When running on Wasm, we are further limited by the current Wasm specification only supporting 32-bit pointers [6]. In our case, this means that we can only evaluate our Go application with a payload size of up to 16KB, as the experiment otherwise runs out of memory. These limitations are unrelated to CoFaaS, and, at the time of writing, efforts are underway to alleviate both of these limitations [14, 15].

For the round-trip-time benchmarks (Section 4.2), we invoke 6,000 requests back-to-back for every configuration. We chose this number of requests to get a representative number of data points while not exceeding memory capacity in the non-GC configurations. For evaluating the latency of inter-function requests, we report the mean of 100 inter-function requests and we repeat each experiment 100 times.

4.2 Round-trip latency

The round-trip latency is defined as the time it takes for a client to receive the reply for a request that it sends. The speedups achieved by the CoFaaS optimized application over the native baseline are shown in Figure 4. The blue/shaded boxes mark the median request payload size as observed from traces of real-world FaaS deployments on the Azure cloud [13] and is hence a notable data point. We run the Rust benchmarks with payload sizes from 1KB to 512KB and Go benchmarks with sizes from 1KB to 16KB. Due to the GC limitations outlined in Section 4.1, Go benchmark cannot run with payloads of more than 16KB. In both cases, we perform 1, 10 and 20 inter-function calls per client request.

Figure 4 shows that the payload size does not impact configurations that issue only a single inter-function call, i.e., CoFaaS consistently yields speedups of 2.75× and 1.75× for

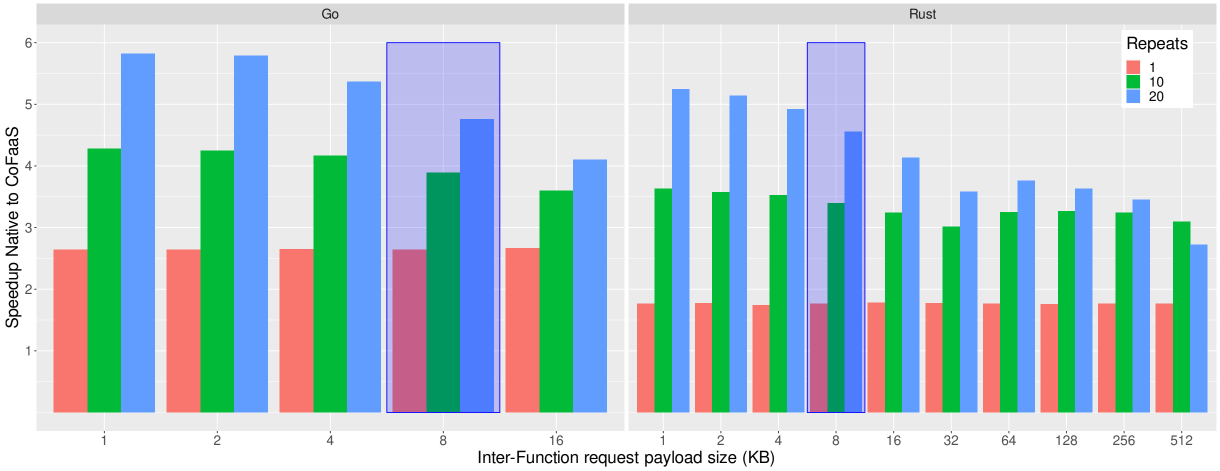


Figure 4: The round-trip latency of issuing a request to our example application.

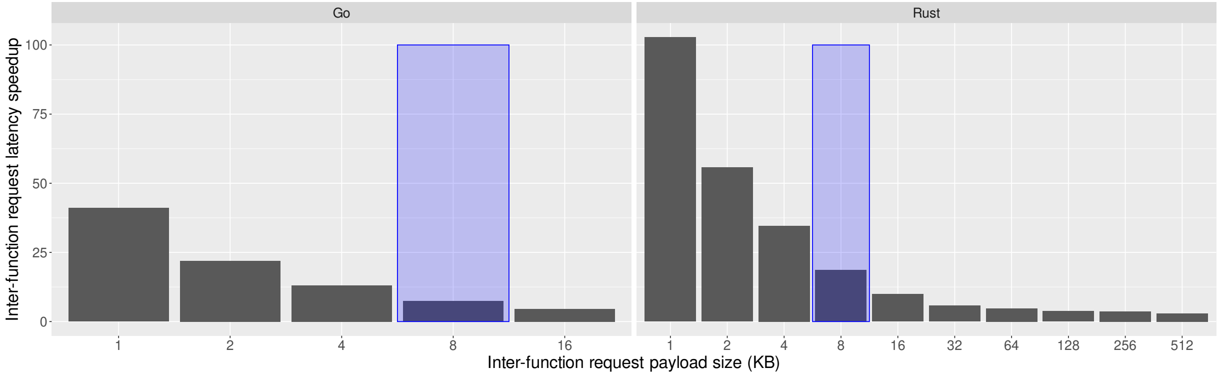


Figure 5: The speedups of a single request.

Go and Rust, respectively. The reason is that a single inter-function call accounts for a moderate but significant fraction of the complete request round-trip time. Thus, as we increase the number of inter-function calls, the achieved speedup also increases. Peak speedups are achieved when an inter-function 1KB request is repeated 20 times, yielding a 5.9× speedup for Go and 5.2× for Rust. This shows that the beneficial impact of CoFaaS is larger for applications that perform a lot of inter-function requests.

4.3 Inter-Function Request Latency

Next, we investigate the latency of individual inter-function requests; the primary target of CoFaaS. Figure 5 shows the speedups of the CoFaaS optimized application over the native baseline when measuring the latency of a single request. Compared to the results in Figure 4, the speedups are significantly higher as this measurement bypasses constant factors involved in every request issued to the application.

This result cements that CoFaaS is highly effective at reducing the latency of inter-function calls in serverless applications. For the Rust application with a 1KB request payload size, we see a 100× speedup. Similar to our round-trip time evaluation, we see diminishing returns when increasing payload sizes. We can understand this trend using the same intuition as before; for larger payload sizes, the time needed to transfer the request payload data becomes dominating regardless of the transfer method used.

4.4 Scalability

We now discuss an important aspect of the FaaS computing model: the ability to elastically scale additional function instances as needed and execute multiple smaller functions in parallel. In FaaS, this is easily achieved and is a side-effect of the function-level granularity in FaaS and the RPC-level function communication. The idea is, on load spikes, we can easily spawn additional instances of a function to handle the load. Additionally, we can distribute functions across several nodes as the network-backed RPC interfaces used for communication naturally enable this.

CoFaaS does not currently support function-level scaling in FaaS applications. This is due to several reasons. First of all, Wasm currently has no support for threads. Secondly, we do not support to optionally maintain networked coupling of FaaS functions when multi-node processing is needed. It is important to note that none of these limitations are fundamental to CoFaaS. Efforts are under way to add supports for threads to Wasm. This will allow CoFaaS to orchestrate function to run in parallel when this is desirable. Furthermore, optionally keeping the network coupling of functions in some cases is simply a matter of additional engineering.

4.5 Security implications

The security of a CoFaaS-transformed application is equivalent to the native FaaS application because the WebAssembly component model gives a separate memory space to each component and no other resources are shared. Therefore, the components do not have direct access to the memory spaces of each other, thus providing isolation. This means that the fundamental security assumptions of an application are not changed by applying the CoFaaS transformation.

5 CONCLUSION

We have now presented CoFaaS which exploits the well-defined RPC interfaces of FaaS applications to automatically consolidate functions on top of a Wasm runtime — thereby alleviating inter-function communication overhead by not accessing the network layer when functions are scheduled on the same compute node. Our evaluation showed that CoFaaS is highly effective and reduces inter-function communication latency by up to 100× and application-level request round-trip time by up to 6×.

REFERENCES

[1] [n. d.]. Apache Thrift. https://thrift.apache.org/. Accessed: 2023-10-19.

[2] [n. d.]. Componet Model design and specification. https://archive.ph/jHHgn. Accessed: 2023-10-18.

[3] [n. d.]. The Go Programming Language. https://go.dev/. Accessed: 2023-10-19.

[4] [n. d.]. Rust Programming Language. https://rust-lang.org/. Accessed: 2023-10-19.

[5] [n. d.]. wit-bindgen readme. https://archive.ph/wjeV2. Accessed: 2023-10-19.

[6] Andreas Rossberg (editor). 2022. *WebAssembly Core Specification.* Technical Report. W3C. https://www.w3.org/TR/wasm-core-2/ https://webassembly.github.io/spec/core/_download/WebAssembly.pdf.

[7] AWS. 2024. What is AWS Step Functions? https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html. Accessed: 2024-03-10.

[8] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 41–54. https://doi.org/10.1145/3361525.3361535

[9] gRPC. [n. d.]. A high performance, open source universal RPC framework. https://grpc.io/. Accessed: 2023-10-19.

[10] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 152–166. https://doi.org/10.1145/3445814.3446701

[11] Knative. 2024. What is AWS Step Functions? https://knative.dev/docs/eventing/. Accessed: 2024-03-10.

[12] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 805–820. https://www.usenix.org/conference/atc21/presentation/kotni

[13] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. 2022. Wisefuse: Workload Characterization and Dag Transformation for Serverless Workflows. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 2, Article 26 (jun 2022), 28 pages. https://doi.org/10.1145/3530892

[14] Mossaka. [n. d.]. wit-bindgen github issue 499: Go bindgen todos. https://archive.ph/mjnRV. Accessed: 2023-10-18.

[15] WebAssembly Propsals. [n. d.]. Memory64 Proposal for WebAssembly. https://archive.ph/wpvrp. Accessed: 2023-10-18.

[16] Protobuf. [n. d.]. Protocol Buffers. https://protobuf.dev/. Accessed: 2023-10-19.

[17] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[18] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-As-A-service. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2438–2452. https://doi.org/10.14778/3407790.3407836

[19] TinyGo. [n. d.]. Go language features. https://archive.ph/AFlUi. Accessed: 2023-10-14.

[20] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. ACM. https://doi.org/10.1145/3445814.3446714

[21] Christopher L. Williams, Jeffrey C. Sica, Robert T. Killen, and Ulysses G.J. Balis. 2016. The Growing Need for Microservices in Bioinformatics. *Journal of Pathology Informatics* 7, 1 (2016), 45. https://doi.org/10.4103/2153-3539.194835