future internet

MDPI

Article

Efficient Data Exchange between WebAssembly Modules

Lucas Silva [1], José Metrôlho [1,2] and Fernando Ribeiro [1,2,*]

[1] Polytechnic Institute of Castelo Branco, 6000-081 Castelo Branco, Portugal; inkeliz@inkeliz.com (L.S.); metrolho@ipcb.pt (J.M.)
[2] CISeD—Research Center in Digital Services, Instituto Politécnico de Viseu, 3504-510 Viseu, Portugal
* Correspondence: fribeiro@ipcb.pt

Abstract: In the past two decades, there has been a noticeable decoupling of machines and operating systems. In this context, WebAssembly has emerged as a promising alternative to traditional virtual machines. Originally designed for execution in web browsers, it has expanded to executing code in restricted and secure environments, and it stands out for its rapid startup, small footprint, and portability. However, WebAssembly presents significant challenges in data transfer and the management of interactions with the module. Its specification requires each module to have its own memory, resulting in a "share-nothing" architecture. This restriction, combined with the limitations of importing and exporting functions that only handle numerical values, and the absence of an application binary interface (ABI) for sharing more complex data structures, leads to efficiency problems; these are exacerbated by the variety of programming languages that can be compiled and executed in the same environment. To address this inefficiency, the Karmem was designed and developed. It includes a new interface description language (IDL) aimed at facilitating the definition of data, functions, and relationships between modules. Additionally, a proprietary protocol—an optimized ABI for efficient data reading and minimal decoding cost—was created. A code generator capable of producing code for various programming languages was also conceived, ensuring harmonious interaction with the ABI and the foreign function interface. Finally, the compact runtime of Karmem, built atop a WebAssembly runtime, enables communication between modules and shared memory. Results of the experiments conducted show that Karmem represents an innovation in data communication for WASM in multiple environments and demonstrates the ability to overcome challenges of efficiency and interoperability.

Keywords: Data-Oriented Design; inter-process communication; performance optimization; serialization; WebAssembly

Citation: Silva, L.; Metrôlho, J.; Ribeiro, F. Efficient Data Exchange between WebAssembly Modules. *Future Internet* **2024**, *16*, 341. https://doi.org/10.3390/fi16090341

Academic Editor: Franco Davoli

Received: 4 July 2024
Revised: 25 August 2024
Accepted: 11 September 2024
Published: 20 September 2024

1. Introduction

For years, web development was almost exclusively in JavaScript (JS). Although various technologies such as Microsoft Silverlight, Java Applets, and Google Native Client (NaCL) have been used to support other languages in the browser, these solutions have never been universally adopted natively by all browsers. WebAssembly (WASM) has recently emerged as a complementary solution to JS. It is a new bytecode with instructions closely resembling those of a conventional processor, but it runs in a sandbox, which limits access to memory and communication with the external world; this makes it secure and fast. The instructions are like those found in most processor architectures, somewhat akin to the Dis Virtual Machine used in InfernoOS [1]. WASM was designed to be general-purpose and usable in any language, although it initially emphasizes C/C++. This concept differs from other bytecodes, such as the Java Virtual Machine (JVM), which was originally developed for just one language, Java, and may have limitations when used to express programs in other languages [2].

Due to the security and portability features of WASM, and its growing adoption as a compilation target by various languages, the technology has been used in other scenarios

beyond the browser context. Currently, it can be found in cloud computing [3], edge computing [4], embedded devices [5], and blockchain [6]. However, with these new possibilities, new challenges have also emerged. The issue of efficient data communication between WASM modules has become a clear obstacle. Transferring data, such as text or vectors, between different modules is not a trivial task for many programmers [7], and the binary format of WASM 1.0 does not provide native features for transferring more complex data structures. In some applications, this can imply serious performance issues when communicating with the module's external environment [8]. The specification of this technology, with its "share-nothing" nature, requires frequent copying to the memory of each of the modules [9]. This need for copying and data transformations has been observed as a problem in some applications, such as Proxy-WASM [10]. The requirement to copy for each individual module can be particularly exacerbated in computational architectures with a "one-to-many" (fanout) pattern, or when large volumes of data need to be transferred to multiple distinct modules.

Karmem aims to overcome the limitations of WASM by disregarding the share-nothing concept [11] and enabling more efficient communication. Its goal is to facilitate improved communication between WebAssembly modules through the implementation of a new interface description language (IDL) and its own communication protocol. Therefore, Karmem makes a meaningful contribution to the realm of WASM development by providing solutions for effective data communication among modules written in different languages. This holds practical significance for distributed applications built on WASM, thereby enhancing efficiency and interoperability.

In addition to the research objective, motivation, and identification of the main contributions presented in this section, this article consists of five further sections. Section 2 provides a theoretical contextualization of WASM. Then, Section 3 summarizes the current state of alternative solutions and their goals, such as Karmem, in the context of communication in WASM. Section 4 elaborates on the goals and priorities of the Karmem approach. It introduces the features of the IDL designed to describe types, data structures, functions, and shared memory regions. The section also outlines the communication protocol specification and introduces a multifunctional development tool employed for code generation in various languages. Section 5 describes the tests conducted using Karmem to assess its ability to address challenges related to efficiency and interoperability in data communication between modules written in different languages. Finally, Section 6 concludes by summarizing the conclusions, limitations, and challenges that remain for future explorations.

2. Background

The development of WebAssembly began in 2015, drawing inspiration from ASM.js and NaCL. WASM's primary goal was to enable web browser applications to achieve performance levels comparable to native applications [12]. WASM is supported by all major web browsers, such as Chrome, Firefox, Safari, and Edge, to ensure consistent execution across various browser environments.

WASM is a virtual instruction set architecture (ISA), consisting of a set of instructions, memory, and a system for communicating with the external environment, and it is executed by a virtual machine (VM). The interaction between the VM and the module can follow certain patterns. As it is a binary format, the WebAssembly Text Format (WAT) was also created, where instructions are written in text, in a more understandable way for humans. The VM, also referred to as "runtime", is responsible for executing WASM, which is either compiled or interpreted. The bytecode must be loaded and validated to ensure functionality and compatibility with supported extensions. Generally, there are three execution strategies: interpreted; Just-In-Time (JIT); and Ahead-Of-Time (AOT). The operating mode can affect Karmem, with only AOT being considered due to its superior runtime performance. By definition, WASM only allows access to information within an isolated and unique memory for each module, which makes it impossible to read

the memory regions of other modules. Furthermore, communication with the external environment is conducted through imported functions exposed by the VM.

For a module to communicate with the external environment, it is necessary for it to import functions, and for another module to export those functions. This is the basic operating principle, like a dynamic-link library (DLL). Conversely, the opposite is also possible: the module can be called and utilized by the VM.

The bytecode's design also addresses the challenge of data transmission over the internet by emphasizing a compact size to allow faster parsing and compilation without the need to download the entire file. This is because bytecode lacks instructions and native support for AES-NI. The WASM bytecode was designed to serve as a unified format, capable of execution across diverse processor architectures, operating systems, and various browsers. Thus, the instructions and operations available in WASM are generally common to all platforms or are virtually possible. WASM extensions introduce additional instructions, some of which align with the introduction of instructions by processors. New instructions are regularly integrated [13], thereby enabling native software to achieve faster execution for specific tasks, while WASM may not have this advantage.

Like JS, this technology runs in its own isolated sandbox. WASM is restricted from accessing information and data from other addresses or programs, and it cannot violate the constraints imposed by the browser, including cross-origin limitations. Its behavior is intended to be highly predictable and deterministic, maintaining consistency across all architectures. This requirement is more stringent than mere portability; WASM outlines specific contexts that might lead to non-deterministic behavior [14]. Examples of such scenarios include the use of shared memory, multiple threads, or attempts to allocate additional memory. Its capability to deliver near-native performance, portability, language independence, and security has made WASM a versatile and powerful technology for web development.

3. Related Work

Inter-module communication in WASM is a common difficulty. Thus, various technologies exist to mitigate this issue. Among them, we will categorize two types: technologies developed for inter-module communication, and technologies for data serialization. The former technologies address protocols that allow data sharing between different modules, compiled to WASM. Serialization technologies focus on the characteristics of each serialization format and are very common for enabling data transfer between different languages and software. Hence, there is an overlap between both technologies.

3.1. Inter-Module Communication

Currently, WASM Interface Type (WIT) [15] is the most popular technology for defining types and interfaces, as used by WASI Preview 1 [16,17]. Alternatives such as waPC use MessagePack for serialization to enable host–WASM communication, with ease of implementation and support across multiple languages [18]. Language-specific options, including Go-Plugin, use Google Protobuf for communication between the host and TinyGo modules [19].

The component model extends WASM, using WIT for description and establishing a specific ABI termed "Canonical ABI" [20], which features a sophisticated type conversion system, "Lifting and Lowering Values", for interoperability across different languages and binary representations. However, it requires a specialized code generator and new WASM instructions and, thus, demands adaptability from languages, compilers, and runtimes to support the proposed extensions. Currently, this extension is still under development and is not fully implemented or defined [21].

Other projects like Extism [22] simplify data transfer for simple data types such as text, and JSON is suggested as one means of sharing more complex data [23], but this may be inefficient for data-conversion needs. The rise of WASM in cloud and edge computing services has led to companies creating their own communication protocols and defining

their own foreign function (FFI) and ABI interfaces. Projects like Envoy and WasmCloud are examples of such solutions [3,24].

All alternatives follow the "share-nothing" philosophy, which prevents data sharing by reference and requires data to be copied. Communication from guest to host can be zero-copy, but communication from host to guest or guest to another guest requires memory copies, and, in some cases, serializations such as JSON, Google Protobuf, or MessagePack are still necessary, which potentially impacts performance.

3.2. Serialization

Considering that WASM is a compilation target for many programming languages and compilers, which implies different memory representations for each language, serializers can be used to establish a standardized data organization. This enables different languages to interpret and access information regardless of their internal memory structure.

In this field, Google reported that 5% of its data center's computational cycles were dedicated to data serialization in 2015 [25], with estimates suggesting a potential increase [26]. Meta, formerly Facebook, disclosed that a mere 18% of CPU cycles were used for core application logic, with the remainder spent on common operations not central to app logic, such as I/O processing, logging, and compression. Serialization and deserialization alone could account for up to 14% of CPU cycles [27].

We compare several data serialization types most used in WASM and IPC:

JSON (JavaScript Object Notation): JSON is widely recognized for its human-friendly text format and adaptability. It is schema-less, with no need for predefined data structures, and thus offers flexibility in data types and structures at the expense of computational efficiency. Its high readability and adaptability entail higher memory consumption for parsing and lack of random access, which make it less suitable for scenarios with heavy binary and numeric data communication.

Google Protocol Buffers (Protobuf): A binary serialization framework that requires predefining data schemas. This enables more efficient data access due to the use of known data types. It has greater data compactness than JSON and is a superior choice for data transfer across networked devices. It requires generated code for each data structure [28] and often outperforms other serializers in resourceful environments but lacks random access reads and partial reads.

MessagePack: This binary format aims to combine the best of JSON and Protobuf. It avoids the need for predefined schemas and operates in binary mode [29]. In general, MessagePack offers better efficiency than JSON without compromising on the self-describing feature. However, MessagePack does include metadata, which may lead to larger sizes than Protobuf, and it does not support random or partial decoding.

Google FlatBuffers: Designed for performance, FlatBuffers allows in-place data access without a parsing step, making it fast and suitable for environments with limited resources. It demands predefined schemas but allows for partial data reads without full decoding, which enhances its efficiency. However, security concerns arise due to the absence of bounds checking. This potentially leads to sensitive data exposure or application crashes [30]. The generated code can be complex, as the optional bounds checking, when available, impacts performance and negates the advantage of random access [31]. Furthermore, each access may require one lookup-table due to dynamic structures and the possibility of omitting non-used fields.

Cap'n'Proto: Developed by a former Google Protobuf developer, Cap'n'Proto is a binary serialization format that enables data reading without prior decoding while allowing for random and partial reads. It needs a schema and uses code generation for various languages and offers security benefits over FlatBuffers through built-in memory bounds checking. Although it introduces a slight performance penalty, it maintains efficiency by not requiring a full data scan for access [32].

Karmem: The initial versions of Karmem shared objectives with FlatBuffers and Cap'n'Proto. Karmem emphasized memory reuse and enabled random data access without

the need for decoding. It used its schema and description language and supported various programming languages. Compared to other alternatives, initial versions of Karmem did not require any data transformation between big-endian and little-endian formats, and while it was limited to 4 GB and could not directly support arrays of arrays, it included bounds checking for each accessed field. Dynamic structures were already supported.

Table 1 provides a comprehensive comparison of various serialization formats, detailing each feature and its level of support across the different serializers. The table highlights key characteristics, such as schema requirements, schema evolution, compression, security, random access, availability, and other specific capabilities.

Table 1. Comparison between serializers.

Feature	JSON	MessagePack	ProtoBuf	FlatBuffers	Cap'n'Proto	Karmem
Requires schema description	No	No	Yes	Yes	Yes	Yes
Schema evolution	N/A	N/A	Yes	Yes	Yes	Yes
Compression	None	Low	High	Low	None	None
Supports random access	No	No	No	Yes	Yes	Yes
Security	High	High	High	Low	High	Medium
Languages coverage	High	High	High	Medium	Medium	Low
Generator integration	N/A	N/A	External	Built-In	External	Built-In
Reference implementation	JS	C	C++	C++	C++	Go and Zig
Maximum Size	N/A	N/A	N/A	2 GB	16 EB	4 GB
Supports custom default value	N/A	N/A	Yes	Yes	Yes	No
Supports fixed-size vector	N/A	N/A	Yes	Yes	No	Yes
Supports vector of vectors	N/A	N/A	No	No	Yes	No
Supports map/dictionary	N/A	N/A	Yes	No	No	No

Since JSON and MessagePack do not offer schema definitions, evaluating features such as fixed-size vectors, vectors of vectors, and dictionaries is not feasible. The compression aspect attempts to highlight potential output sizes: ProtoBuf provides variable size integers and omits undefined values, while FlatBuffers only omits undefined values. Security considerations are challenging to synthesize due to varying attack models. Notably, FlatBuffers lacks built-in boundary checks, which implies the highest risk. In contrast, Karmem and Cap'n'Proto offer these verifications, though Karmem lacks a mechanism for limiting recursion depth in arrays, potentially leading to infinite loops. Cap'n'Proto addresses this issue by allowing for a customizable threshold.

3.3. Conclusions

Considering that different standards have different objectives, they are difficult to compare directly. However, all popular solutions define specific functions to be imported by the module and exported by the host. A common issue among these systems is potential incompatibility, which is notably seen in WASI. As standards evolve and new functions are added, outdated hosts may not support these functionalities. This potentially renders the module inoperable. By contrast, the component model attempts to handle evolution and extend the communication between modules, but it is still under development and requires copying and transforming data between each module.

When analyzing existent serialization formats, JSON is versatile and popular due to its wide support across programming languages. However, it is inefficient for inter-process communication (IPC) and does not allow for partial reading. MessagePack and Protobuf are efficient in compression and performance, requiring a description language and generating specific code for each data field. They excel in scenarios with full data reads and efficient memory management.

Zero-copy serializers such as FlatBuffers, Cap'n'Proto, and Karmem enable direct data reading, making them useful for random access without full message decoding. FlatBuffers support the omission of empty fields, unlike Karmem and Cap'n'Proto. However, Flat-

Buffers' default lacks data verification, which poses security risks, and its generated code may be less efficient in some languages. Additionally, table lookups and multiple pointer traversals can introduce performance penalties, especially with large amounts of data.

4. Karmem Approach

The primary goal of Karmem is to enable efficient communication between WASM modules. It aims to establish a high level of portability and ensure compatibility with many programming languages and with WASM 1.0 without requiring any additional extensions. This is achieved by creating a new protocol for binary representation and decoding.

Performance and security are also a priority. Instead of using "share-nothing", Karmem aims to improve performance through memory mapping techniques, thus enabling direct access to data without the need for additional decoding or copying. Karmem prioritizes error handling over crash susceptibility and emphasizes data integrity and runtime safety with bounds checks. Karmem is designed to be adaptable, supporting both fixed and dynamic structures in its description language. This ensures backward compatibility and allows for careful evolution without disrupting communication with older module versions. As stability is paramount post-standardization, Karmem maintains consistency across various versions and enables the creation of third-party generators, which is currently a common issue for WIT.

4.1. Schema Language

Karmem uses its own description language, KarmemIDL. Other alternatives and strategies were considered, which might avoid the need for a new language but would introduce different problems. Among the possibilities, KarmemIDL could be replaced by pre-compilation attributes (known as decorators/hints/annotations) being integrated into a programming language (Section 4.1.1), or the use of an existing description language (Section 4.1.2).

4.1.1. Compilation Attributes

Some programming languages, such as Rust, Java, C#, and PHP, allow the definition of "decorators" (the functionality may have different names depending on the language). It is also possible to analyze the syntax tree of any language and interpret comments and custom attributes. However, the use of this implementation method was discarded because of the direct link to a specific programming language. Karmem aims to support multiple languages, and defining it within a specific language could hinder understanding and cause greater problems for feature equivalences. This alternative could offer a better user experience when used exclusively in that language.

4.1.2. Existing Language

Another option is to employ other description languages, such as WIT and Protobuf, which are well-known. Although using an existing language may have advantages in terms of ease of use and implementation, it might not be adequate to express the functionalities of Karmem. Thus, creating a language enables a more natural expression of Karmem's features and usage characteristics.

4.1.3. KarmemIDL

KarmemIDL is a custom IDL, specifically made for Karmem. This language is designed to be easily understandable, like C++ and Java, which are widely used for educational purposes [33]. However, it overcomes the syntax limitations of these languages by drawing inspiration from Zig and Odin.

Karmem adopts numeric type names, such as "u64", for greater clarity and less ambiguity, in contrast to terms like "long" or "long long". Unlike many programming languages, Karmem lacks native support for strings, similarly to C and Zig. Strings are

represented as arrays of "u8" or "u16", depending on the encoding. There is an additional attribute to flag them as strings, but sentinels are not used to identify their ending.

The major difference between KarmemIDL and other types of IDL and programming languages is the ability to specify the layout of the array, such as Struct-of-Arrays (SoA) and Arrays-of-Struct (AoS), as well as representation of Unions arrays, which can also use new technics named Bunch-of-Values-of-Arrays (BoVoA) and Bunch-of-Values-of-Struct (BoVoS). The ways of exporting or importing interfaces are not only limited by functions (or methods) but extend further to support shared memories and communications patterns, such as FanOut, Direct, Singleton, and Pool.

Figure 1 describes one struct and one interface, using some unique features from Karmem. The "struct_name" struct contains two fields, which are arrays of "another_struct"; the usage of "#aos" and "#soa" changes the memory layout. The "dynamic" flag makes it possible to extend the struct with new fields after the last existent one. The "updater" interface is also a "dynamic" interface, enabling extension with more methods and memories, and "fanout" is the communication pattern, which defines how other packages will communicate with them.

```
struct another_struct (dynamic) {
    a: u32;
    b: u64;
}

struct struct_name (dynamic) {
    field_as_soa: []another_struct #soa;
    field_as_aos: []another_struct #aos;
}

interface updater (dynamic, fanout) {
  shared: memory (import: write, export: read, 64);
  out: memory (import: read, export: write, 1);
  update: fn(input: struct_name @ shared) (res: another_struct @ out);
}
```

Figure 1. Description of one struct and one interface with Karmem.

The memory regions can be declared using "memory" and later declared in the function argument to enforce that such data must live in that memory zone.

4.2. Data Layout

Each programming language and serialization format defines its own ABI and memory layout, as previously discussed. Karmem attempts to take advantage of this data layout to improve its performance and interoperability.

In this context, Karmem aims to follow the Data-Oriented Design (DOD) paradigm, which prioritizes data organization while avoiding abstractions that do not consider the hardware and execution environment, aiming for better performance and easier data processing. Noel Llopis popularized the term in 2009 [34]. This paradigm is widely used in high-performance computing, especially in games and game engines, such as Titanfall, Battlefield 3, and Sunset Overdrive [35–37]. The Unity Engine developed the Data-Oriented Technology Stack (DOTS) to leverage the performance advantages of DOD, utilizing the Unity Burst compiler and the High-Performance C# (HPC#) language [38,39].

Additionally, new languages such as Odin, Jai, and Zig, which are still in development, facilitate the use of DOD [40–42]. Richard Fabian highlights that, despite being controversial, DOD can coexist with other programming paradigms [43]. However, there is little formal and academic definition of DOD, as it is more discussed in the video game industry [44,45]. Implementations of Object-Oriented Design (OOD) can cause performance issues due to the coupling of data with behaviors, thereby compromising cache efficiency and the use of SIMD [46–48]. Karmem is designed to optimize data communication by

proposing an appropriate memory layout, minimizing overheads in reading large volumes of data, and offering the flexibility to describe data optimally for different platforms, such as WASM.

Karmem is focused on WASM, and due to limitations inherent in that technology, offsets (also named pointers) are limited to 32 bits, which restricts the maximum size to 4 GB.

4.2.1. Structs

Structs can store different data types and can be either inline or dynamic. Inline structs are the most basic as they do not support evolution and will be in line with the parent structure. In contrast, dynamic structs can be modified by adding new fields at the end to maintain backward compatibility with previous versions.

The resulting memory layout is similar to C but uses a fixed padding size of 64 bits, the ordering of the declaration is the same as the memory layout, and no optimization or reordering is performed. However, when the struct is used as a type for arrays, we offer different kinds of optimizations.

4.2.2. Arrays

Arrays represent a common data structure that allows for the storage of multiple items and iterations through them. In this document, we will consider "arrays" as dynamic-sized arrays. Each programming language can implement this feature differently, but, usually, we can distinguish between three types of implementations: Arrays-of-Struct (AoS), Struct-of-Arrays (SoA), and Arrays-of-Pointers (AoP).

Usually, OOD tends to use AoP or AoS, which makes it harder to use SIMD instructions [46,47] and also introduces additional performance penalties in handling scheme evolutions. KarmemIDL allows the programmer to specify the layout type. The default one is SoA, which is a method of representing homogeneous data structures in vectors instead of representing a vector of heterogeneous data structures. This technique offers several advantages, particularly by enabling the use of SIMD and always having a fixed step size. Due to the data-access pattern, SoA can be beneficial by allowing partial data reads and avoiding cache pollution with unnecessary data. Furthermore, it can save space by eliminating paddings between each field. This method is particularly useful in scenarios that benefit from sequential data access and efficient cache usage, as was highlighted in a study by Mattias Karlsson [45]. While DOD has received limited research attention, we hypothesize that Karmem can benefit from DOD. This approach simplifies implementation across multiple languages and leverages compiler optimizations such as auto-vectorization. This advantage is due to the usage of a stable homogeneous array representation, which is native in most programming languages, unlike the abstracted representations offered by other serializers such as FlatBuffers and Cap'n'Proto.

Karmem also uses consistent alignment across all languages and architectures. Arrays are aligned by 512 bits, which is a recommendation from Intel to take advantage of AVX512 [49]. It is important to note that WASM is currently limited to 128-bit SIMD lanes.

Considering the shared-memory aspect of Karmem and the potential reusability of such arrays, they include not only the size but also the capacity. This design helps to avoid reallocations and reduces the performance cost associated with copying and shifting data.

Figure 2 illustrates the binary representation of an array of 2D Point (X, Y), ignoring padding and additional data (such as sizes) for simplification. The AoS method (A) is used in C, Go, Zig, and many other languages, whereas the AoP method (C) is commonly found in languages based on OOD, such as Java. Karmem uses the SoA (B) by default.

4.2.3. Unions

Unions are one of the most basic data structures to enable polymorphism, even though each programming language names this type differently. It can be classified as Tagged-Union or Untagged-Union, relative to the explicit knowledge of the type being stored. It can

also be classified as direct or indirect, relative to how it uses the space and how much space it can take. For instance, C uses the largest value size as the union size and for additional paddings, and it does not store the type of data. Languages such as Go or Java implement their interface using a fixed size and use pointers to the actual data and metadata.

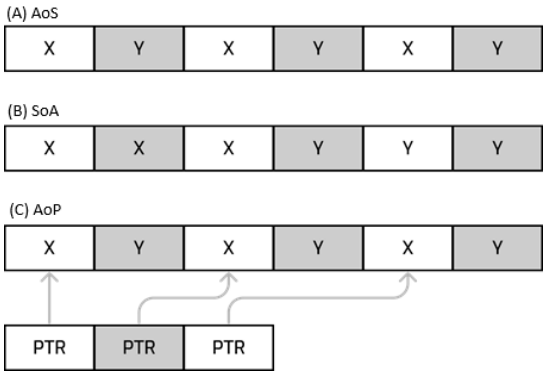

Figure 2. Comparing AoS, SoA, and AoP.

Karmem stores the metadata of the current type inline but does not necessarily use the naïve approach of Go, which stores two words (one pointer to metadata and another to the actual data). Instead, we can also gain some advantages when combining unions with arrays.

When used as a single value, Karmem's storage mechanism includes an identifier number (a tag) and the offset to the actual data. By default, the tag is 32 bits, though this can be adjusted in the schema. While using smaller numeric types (such as uint8) does not provide an advantage in this context, the feature is supported for reasons that will be discussed later.

Using unions within arrays allows for various memory layout configurations. Karmem enables programmers to define these arrays using different layouts such as AoS, SoA, and Bunch of Values (BoV). Due to the lack of formal nomenclature, we name BoV a data type, which contains multiple vectors for each of the data types represented in a union; this method lacks the proper insertion ordering. BoV's name is inspired by "Just a Bunch of Disks" (JBOD), which is used in data-storage systems where multiple hard drives operate independently without any interrelation. This definition can be extended to BoVoA (Bunch of Values of Arrays) and BoVoS (Bunch of Values of Structures), which combines the representation of the inner union as AoS or SoA, respectively.

The code, in Figure 3, defines a structure for a particle that contains an array, "pos", which can consist of either 2D or 3D points. The union definition (any_point) employs an 8-bit tag, with each tag explicitly defined, although this is entirely optional.

Figures 4–7 illustrate the memory layout of each representation, with padding and size considerations omitted for simplicity. The AoS (depicted in Figure 4) is the most common method. However, searching for a specific tag can be inefficient due to the constant 32-bit offset between each tag. Additionally, AoS cannot leverage shorter tag types effectively, resulting in an additional 24 bits of padding when using an 8-bit tag.

SoA can optimize linear search and enable the use of SIMD (single instruction, multiple data) instructions. Specifically, with an 8-bit tag, it is possible to compare one tag against 64 tags per loop using a single AVX512 instruction. In contrast, using the same AVX512 instruction with AoS limits the comparison to a maximum of eight tags due to padding and pointer data. Furthermore, accessing the pointer can lead to cache misses if prefetching is not employed for the second array.

In cases where the order is not important, BoV (either BoVoA or BoVoS) can be advantageous. These configurations create separate arrays for each type declared in the union, thus eliminating the need for searching and enabling the use of non-polymorphic functions. Figures 6 and 7 illustrate this type of construction.

```
struct point_xy (inline) {
    x: f32;
    y: f32;
}

struct point_xyz (inline) {
     x: f32;
     y: f32;
     z: f32;
}

union any_point (dynamic, u8) {
     point_2d: point_xy = 1;
     point_3d: point_xyz = 2;
}


struct particles (inline) {
      pos: []any_point #soa;
}
```

Figure 3. Example of a union capable of holding either a 2D point or a 3D point.

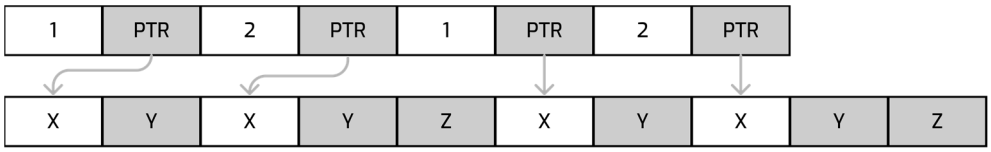

Figure 4. Simplified memory layout of AoS of unions.

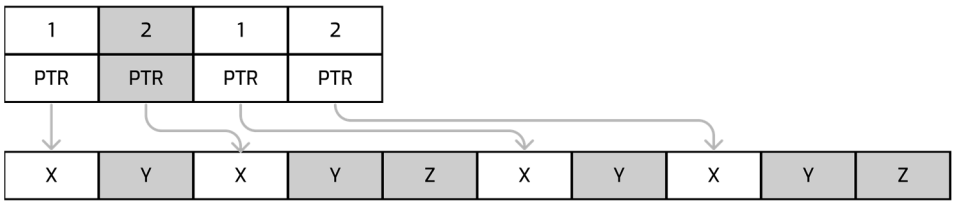

Figure 5. Simplified memory layout of SoA of unions.

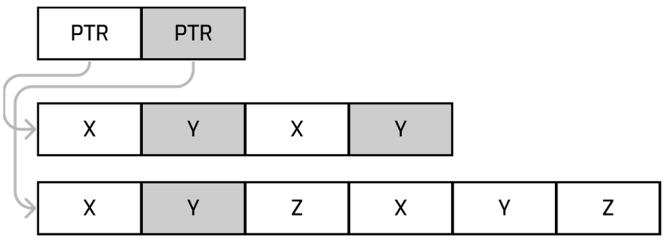

Figure 6. Simplified memory layout of BoVoA.

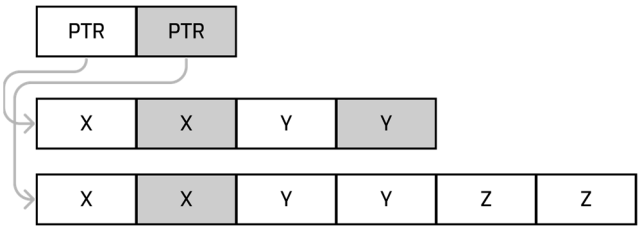

Figure 7. Simplified memory lLayout of BoVoS.

4.2.4. Evolution

Typically, the evolution of structures is not a concern for compiled languages without an IPC requirement, since types are well-known at the time of compilation and only consumed by the application itself. However, for Karmem, it is possible to have multiple versions of the same definition, all of which must be able to communicate seamlessly across multiple applications. Changing an existent structure usually changes paddings and distances between each field in an array or demands the usage of pointers and indirect access.

The construction of dynamic structures may impose performance issues since each structure can have an unknown internal layout. The strategy employed by FlatBuffers is to use one pointer to an internal lookup table for each structure. This can be useful for omitting unused fields and reducing the space required for the serialization format. However, it introduces new steps, prevents the usage of SIMD instructions, and can populate the CPU cache with unnecessary data.

Karmem takes advantage of SoA, which enforces a consistent memory layout and a single lookup table. It requires only three bound checks independent of the number of items in one array. This allows the program to take advantage of SIMD while fully ignoring the existence of newer fields. In this case, a single table lookup is responsible for returning a pointer of an array of homogenous-type data (numbers).

Dynamic structs can store different types of data and can be modified by adding new fields at the end, thereby maintaining backward compatibility with previous versions. This functionality is implemented by checking the size of the structs and comparing it with the offset of the data to be read. However, checking the size of each item in one large array can be inefficient; this is how Cap'n'Proto works.

Figures 8–10 demonstrate this functionality by adding new fields while preserving compatibility with previous versions. Although this technique is widely used for IPC, it requires the use of pointers, which can cause inefficiencies in data access, especially in the case of arrays.

```
1. struct example (dynamic) {
2.     field_1: u32;
3. }
```

Figure 8. Struct dynamic example, version 1.

```
1. struct example (dynamic) {
2.     field_1: u32;
3.     field_2: u32;
4. }
```

Figure 9. Struct dynamic example, version 2.

```
1. struct example (dynamic) {
2.     field_1: u32;
3.     field_2: u32;
4.     field_3: u16;
5. }
```

Figure 10. Struct dynamic example, version 3.

When dealing with a single struct, there is only one offset to access that single struct, as shown in Figure 11. The total size is stored in 29 bits, while the remaining three bits store the unused space (up to seven bytes). By applying delta encoding, the used value can be determined based on the total size (which is always multiplied by eight bytes): $(\text{size\_total} \times 8) - \text{size\_wasted}$.

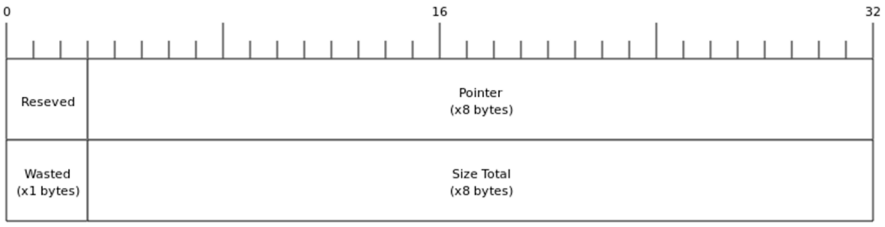

Figure 11. Memory layout of a single dynamic struct.

Assuming the data definition in Figure 4, which has 16 bytes with 10 bytes used, it would result in Figure 12.

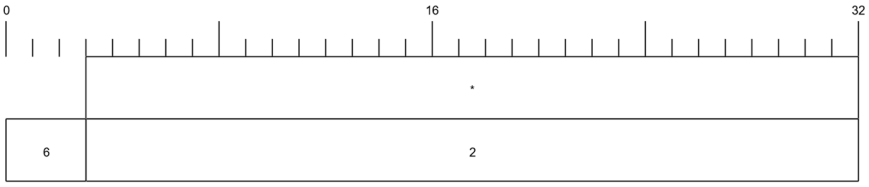

Figure 12. Memory layout example assuming the definition of Figure 13.

```
struct array_of_example (dynamic) {
    items: []example #soa;
}
```

Figure 13. Memory layout of SoA.

If the structure is used in a SoA, it is represented by an additional structure containing the offsets of each vector. All offsets are always multiplied by 512 bits, as shown in Figure 14.

To maintain compatibility and support for structure evolution, an auxiliary lookup structure was created. The lookup consists of two arrays: one for the hash and another for the respective offsets of each array. The hash considers the position of each element defined in the struct, following one hash-chain for flattening structs, to ensure stability regardless of modifications to other intermediate structs.

Karmem uses a linear search to find the entry point of each array in the SoA. This linear search is optimized with SIMD, when possible, to drastically reduce the number of required comparisons. Therefore, such arrays are always a multiple of 64 bytes, which enables the use of AVX512. The hash value is limited to 4 bytes, allowing up to 16 comparisons per loop. Once a matching hash value is found, it retrieves the pointer, as shown in Figure 15. To prevent malicious crafted message to exhaust resources with large linear search, a maximum number of fields can be defined. This limitation can cause legitimate messages to be invalidated if the threshold is too tight. However, a similar issue also exists on Cap'n'Proto when decoding deeply nested messages, and a traversal limit is used to prevent such exploit [32].

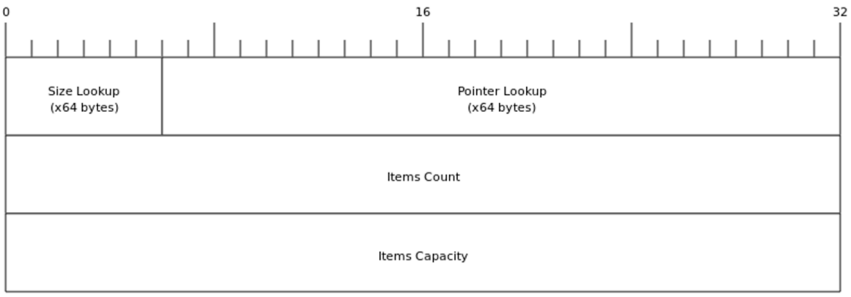

Figure 14. Illustration of SoA.

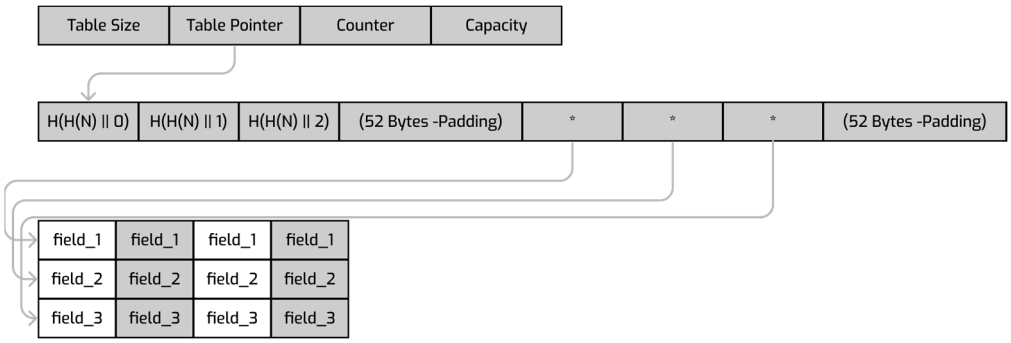

Figure 15. Struct using example as array.

4.2.5. Flatten

The technique known as "flatten" involves removing the intermediate internal structures of a more complex inner structure. This approach has the benefit of simplifying data reading by eliminating heterogeneous internal structures. Karmem uses this technique to facilitate faster data access and simplify implementation across various programming languages.

This method is combined with SoA, as previously mentioned, which reduces the number of table lookups and pointers to traverse. However, the generated code can be confusing.

4.3. Communication

Karmem enables communication between different WASM modules that are on the same virtual machine. Due to its serialization format, it also allows reading in any language.

In the current state of WASM, it is mandatory to copy data for each instance. The existence of shared memory, which is a proposal part of Threads [21], is not supported by all programming languages, compilers, or runtimes. Also, shared memory is insufficient because it requires all programs to know the stack area of each program and carefully orchestrates memory allocations. Accordingly, another extension can be used, such as multiple memories, but that can be challenging to implement and is usually not supported by multiple languages and compilers.

Karmem consists of two steps: code generator and runtime. The code generator generates code in multiple languages. This is restricted to a pre-compiler and does not require patching the bytecode. Karmem also has a new runtime, which is built on top of a WASM runtime, such as Wazero. The runtime is responsible for managing shared memories and handling calls between different modules.

4.3.1. Code Generator

Like other technologies such as FlatBuffers, Protobuf, and Cap'n'Proto, Karmem also requires a custom code generator, which enables the program to read and write data in a consistent layout while disregarding specific internal memory representations. Acting as a pre-compiler, it does not alter the generated bytecode or introduce new instructions or opcodes. Currently, the prototype can generate code for Go, Zig, and C, with plans to support Swift, Python, Odin, and Rust in the future. Language selection is based on popularity within the WASM scope [50], maturity of the supported compilers, and the programming language paradigm.

Some programming languages can be difficult to implement and, in some cases, might offer limited features. To briefly mention some issues, Java, PHP, and JS cannot handle unsigned integers with a 64-bit size. Languages such as Java and C# use mark-and-compact garbage collectors, which can move the used memory, whereas Go uses a mark-and-sweep approach, but this might change in the future [51].

The generated code can provide better performance and mitigate security issues. In the 2021 OWASP report, deserialization of untrusted data is listed among the top 10 security issues. It can lead to a potential application of logic abuse, denial of service, or arbitrary code execution, especially in PHP, Java, and C# [52,53]. Oracle acknowledges that Java's serialization poses significant security risks. Indeed, it accounts for over one-third of its security issues [54] due to the nature of dynamic programming.

While a sandbox like WASM can mitigate some security concerns, internal security checks, such as bound checks, remain essential to prevent unauthorized data access and application logic issues. However, the generated code cannot provide a native way to manage concurrent or conflicting access in a shared-memory region. That responsibility lies with the modules themselves, and non-parallel operation is the default.

Unlike protocols such as WebStorage [55], Karmem does not treat stored and transferred data as confidential and sensitive by default. To prioritize performance, the generated code does not fully erase data between serializations and thus potentially allows access to partial or complete data through paddings and unused areas. The generated code supports in-place mutability of data, including resizing an array when it is smaller than the initially allocated capacity. In such cases, the code will only erase the remaining bytes of the last 512-bit block of the array to prevent SIMD instructions from reading potentially dirty data. However, any other data will not be erased, and it is the programmer's responsibility to evaluate if such action is required and can be performed manually.

In a WASM environment, the host system is responsible for executing the guest code. Consequently, encryption and authentication of communication are useful in limited scenarios. However, in a networked environment, counter-based stream ciphers like ChaCha20 can be employed to encrypt data while maintaining the random-access capabilities of Karmem without requiring the decryption of the entire message. However, verifying the integrity and authenticity of the message may require analyzing the entire encrypted message.

The generated code, particularly for the host, allows the programmer to specify a custom timeout to prevent resource exhaustion when running untrusted WASM modules, which will be later discussed in Section 4.3.3. However, synchronization features such as atomic operations over shared memories are not available and might cause undefined behavior. The host is solely responsible for the synchronization and schedule of execution of multiple concurrent guests.

4.3.2. Interface and Patterns

Similarly to WASM, Karmem allows import and export functions through interfaces. Such interfaces are also dynamic and mutable, allowing for the addition of new functions. Furthermore, Karmem was inspired by RabbitMQ applications.

RabbitMQ defines some communication patterns such as "Fanout", "Direct", or "Topic" message exchanges [56]. Additionally, initialization types such as "Singleton" and "Multion" were considered, along with considerations for parallelism, similar to forking processes. While not exhaustive, communication patterns can be categorized in multiple ways. Notably, the well-known "Gangs of Four" design patterns [57] are sometimes critiqued as compensating for a lack of such features in the language itself [33,58,59]. Although Karmem is not a programming language, its ability to define communication, initialization, and parallelization patterns within an interface may simplify usage and enhance performance by reducing unnecessary inter-module calls.

The direct mode is the most common, in which a WASM module communicates with another WASM module, even if multiple modules export the same interface. Figure 16 illustrates this scenario. The selection of a module can be decided by the host and the importing module itself.

The fanout mode aims to communicate with multiple different modules that implement the same interface. Figure 17 demonstrates how this feature works. In this mode, whenever a new WASM module exports an interface that is imported by the module, it

will automatically bind and be called along with all the others. The Singleton mode is an initialization method that has only one instance, meaning that all imports are directed to the same module. This does not prevent multiple modules from exporting the same interface. In essence, it is the opposite of fanout, where all importers use the same exporter. Figure 18 illustrates its operation, assuming there is only one exported module. The key difference here is that when the module does not use pure functions, the altered state is persistent between "Importer 1" and "Importer 2". This method can cause performance issues when frequently called, as there is only one instance for handling all calls from all importing modules.

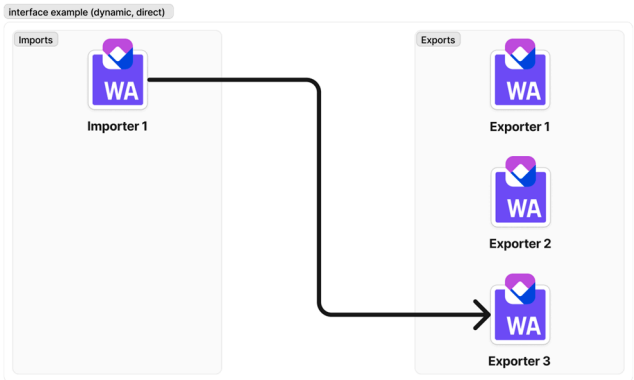

Figure 16. Example of direct interface.

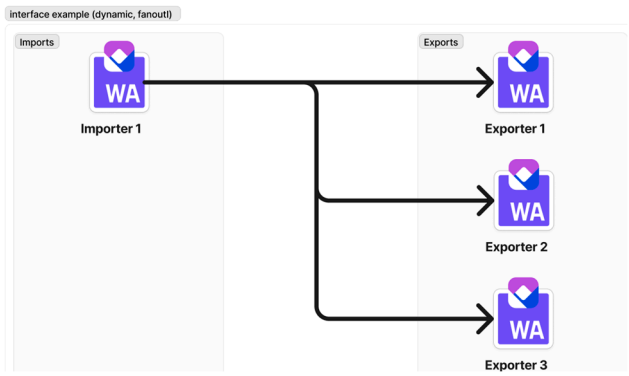

Figure 17. Example of fanout.

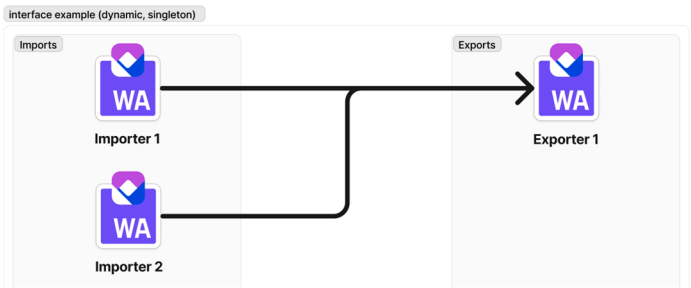

Figure 18. Example of Singleton.

4.3.3. Shared Memories

Using KarmemIDL, it is possible to describe memory areas that will be shared across the imports and exports of such an interface. Unlike a Threads extension, Karmem offers a minimum permission level, thereby restricting modules from writing into the shared memory.

Memories can be defined within the interface. This should include the access level from the perspective of both importers and exporters, as well as their maximum size, defined in 64 KB pages, the same as WASM. Although this memory limitation might seem significant, it is necessary to analyze the context of WASM usage, especially in serverless computing. In this scenario, services such as Amazon Lambda@Edge, Vercel Functions, and Shopify Functions, among others, also have limitations regarding memory and processing time [60–63]. While processing time is important, defining it in an IDL does not seem appropriate since processing times can vary due to the underlying hardware and business logic. Therefore, the generated codes and communication system may contain options to add some timeout.

4.3.4. Runtime

Karmem introduces a specialized runtime that deviates from traditional WASM by managing module memory uniquely, focusing on shared memory regions. It leverages the existing Wazero virtual machine, which offers JIT/AOT compilation capabilities but allows for customization to suit various programming languages. This approach ensures optimized performance across different hosts. Karmem requires specific OS functionalities, such as mmap for shared memory management, which constrains its deployment based on available OS resources and needs custom implementation for each OS.

Runtime prioritizes Linux and FreeBSD due to their popularity [64] and license, respectively. Windows, while being the most popular OS for desktops, has inconsistencies in its support for certain memory allocation techniques across different versions of Windows. These inconsistencies can complicate the implementation of the Karmem runtime and its tests, particularly when the process involves allocating memory addresses without the immediate commitment of physical memory. Some operating systems introduce significant limitations. iOS/iPadOS/tvOS are noticeable examples due to a lack of MAP_JIT support, which prevents WASM runtimes from creating executable memory. The feasibility of Karmem on mobile and non-traditional platforms such as gaming consoles and other BSD variants remains largely unexplored, but they potentially align with existing Linux and FreeBSD implementations. The integration challenges across different platforms emphasize the need for adaptable and flexible software solutions within diverse hardware environments.

Communication between modules, whether guest–guest or guest–host, is facilitated by the host. This enables the calling of WASM modules that implement the interfaces defined in KarmemIDL.

To allow communication and function calls, a common interface has been defined so that all WASM must export all functions described, as shown in Figure 19. This communication interface is similar to waPC as it allows for the dynamic invocation of other functions.

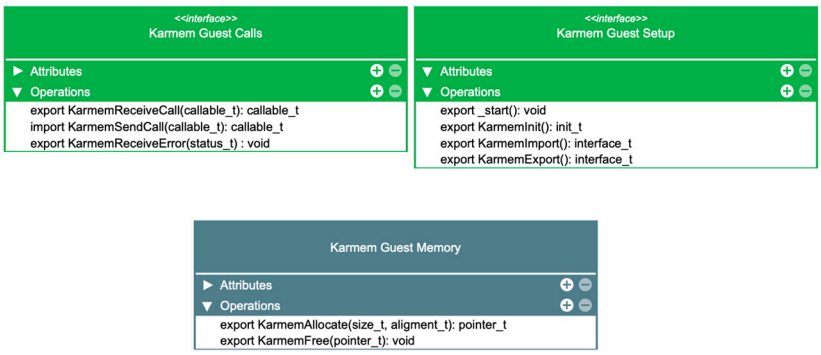

Figure 19. Functions exported or imported by the guest.

When a WASM module is loaded and initialized via _start, the runtime calls the "KarmemInit()" function. This function must report the number of interfaces it exports and imports. Subsequently, the runtime calls the "KarmemImport()" and "KarmemEx-

port()" functions to obtain information about the interfaces that are imported and exported. Memory allocations are handled by KarmemAlloc functions.

5. Results and Discussion

This section presents the tests conducted using Karmem to assess its ability to address challenges related to efficiency and interoperability in data communication between modules written in different languages. The section is divided into two subsections. The first contains the initial experiments conducted in multiple environments to evaluate Karmem's efficiency in reading and writing data. The second subsection presents the tests conducted to evaluate Karmem's performance regarding serialization times, comparing it to the performance achieved by FlatBuffers and JSON.

5.1. Performance Analysis

The main priority of Karmem is performance and memory layout. Therefore, it has been carefully developed to reduce the overhead caused by dynamic structures. Using DOD also allows for greater efficiency in reading and writing data and avoids the additional cost of size checks. Some studies related to the use of DOD confirm the performance difference compared to the OOD model [65], but these tests were conducted on fixed-size structures, consequently with a fixed stride, at compilation time (i.e., the native structs of the language), which allowed for more aggressive optimizations.

Karmem needs to provide backward compatibility and data-structure evolution. Allowing for the evolution of structures is achieved by creating them without a fixed size at compilation time. However, this may sacrifice performance when used in the Array of Structures (AoS). Therefore, the hypothesis that DOD could still offer better performance in the Karmem context also needs to be evaluated. This hypothesis is proposed because vectors always have a fixed spacing between elements, and adding new vectors at the end does not affect the existing ones. In this section, we will test this hypothesis and its motivations.

5.1.1. Methodology

The tests conducted follow some of the recommendations described in the article "Producing wrong data without doing anything obviously wrong!" [66], which recommends conducting tests in multiple environments and mentions the impact of the linker's order and the operating system's "environment variables". Based on this article, a simplified variation was created by modifying Wazero, the WASM virtual machine. The modification consists of placing each WASM function, in assembly, on a different memory page. This means there is a 64 KB padding between each WASM function. This approach should prevent potential advantage due to instruction cache, ensuring that the observed performance is attributable to the code itself rather than incidental compiler and runtime optimizations, such as the placement of frequently executed functions in contiguous memory or the inadvertent alignment of dependent functions within the same instruction cache line. To increase the diversity of the tests, different devices and virtual machines were used. While virtual machines can cause greater variation in results, it is essential to recognize that cloud computing is an important execution environment. All tests were conducted using four different processors, with the configurations listed in Table 2. PC1 and PC2 are identical, differing only in their operating systems, running FreeBSD and Ubuntu, respectively, on two virtual machines. PC5 represents a cloud server. It is an expandable series server from Azure, which has a base CPU performance of 10% but allows for short bursts of higher CPU usage [67]. We chose to test on as many operating systems and architectures as possible. The operating systems were selected based on their popularity, license, availability, and compatibility with the CPU architecture and Karmem runtime. Tests on Windows were not conducted due to limitations, as previously discussed in Section 4.3.4. Tests on macOS were performed because it is the only compatible OS for the Apple M2 Max processor at the time of writing.

Table 2. Hardware configurations of the devices used for testing.

Configurations	PC1	PC2	PC3	PC4	PC5
Operating System	FreeBSD 13.2	Ubuntu 22.04.3	Debian 10	macOS 14.0	Ubuntu 22.04.3
Virtualization	Proxmox (7.3-6)		None	None	Azure (B1s)
RAM size	8192 MB		65,536 MB	32,768 MB	1024 MB
RAM type	DDR4		DDR4	LPDDR5	- [1]
ECC	Multi-bit		Multi-bit	- [2]	- [2]
RAM Bandwidth	2400 MT/s		3200 MT/s	6400 MT/s	- [2]
CPU	AMD EPYC 7551P (2017)		AMD Ryzen 7 3700X (2019)	Apple M2 Max (2023)	Intel Xeon Platinum 8272CL (2019)
Threads number	4 [3]		16	12	14 [4]
CPU clock speed	2000.00 MHz		2194.70 MHz	2424.00 MHz	2600.00 MHz
Cache size	L1: 96 KB L2: 512 KB L3: 64 MB [4]		L1: 512 KB L2: 4 MB L3: 32 MB [5]	L1: 192 KB L2: 32 MB [5] L3: 48 MB [5]	L1: 64 KB L2: 1 MB L3: 36 MB [5]
Cache line size	64 bytes		64 bytes	128 bytes	64 bytes
TLB size	2560 4KB Page		3072 4KB Page	2048 16KB Page	1536 4KB Page

[1] Due to virtualization, this information cannot be obtained. [2] There is no publicly available information regarding the type of error correction present in the memory integrated into the Apple M2 Max. [3] This number differs from the maximum number of threads of the processor due to virtualization. [4] Shared across all cores. [5] The last commit has the hash 27624049dc46c307f0fc6c4771b7df0609c63ff1.

All tests were conducted using Go 1.21, with Wazero 1.6.0 as the WASM virtual machine modified to create functions in different memory pages. Additionally, all tests were implemented using Go's built-in testing tool [68]. The WASM modules were compiled using Zig 0.11.0 and TinyGo 0.30.0 for Zig and Go, respectively.

5.1.2. Dynamic Structures Test

This test aims to validate the tests previously conducted in another article [65], as well as to validate the hypothesis that DOD could be even better considering dynamic structures, which will be used in Karmem. The test consists of the structure demonstrated in Figure 20. Within it, there are three data structures, in AoS, SoA, and AoP, used in the variables "scooters_aos", "scooters_soa", and "scooters_aop". Additionally, there is "scooters_aos_u32", which uses AoS as a vector of u32. This vector points to the same location as "scooters_aos". The structure is exactly 64 bits (8 bytes) to allow for SIMD use, as WASM only supports 128 bits, although this may be subject to change in the future.

The function to be executed is demonstrated in Figure 20, where all fields of "distance" are updated by a single variable at runtime. The host is responsible for providing the values, and the guest exports the functions, which always receive an input value.

Since there are tests in shared environments (PC1, PC2, and PC5), all tests were executed in random order and repeatedly. All tests were performed 64 times, except for the native test, which was executed 65 times.

Different variations were created to achieve the goal of adding value to all distances. These different implementations allow for an analysis of the impact of data organization and the possible optimizations performed automatically by the compiler, which may not be applicable to non-native (or non-fixed) structures. The implementations are explained below.

Native: The native test aims to test the performance of the language's native structure in AoS format. Its implementation is as natural as possible, as can be seen in Figure 21. This code iterates through each scooter and increments its velocity without any explicit optimization. This code is automatically vectorized by the compiler, which will utilize SIMD when possible.

```
const ScooterAoS = struct {
    id: u32 = 0,
    distance: u32 = 0,
};

const ScooterSoA = struct {
    ids: []u32 = undefined,
    distances: []u32 = undefined,
};

var scooters_aos: []ScooterAoS = undefined;

var scooters_soa: ScooterSoA = ScooterSoA{};

var scooters_aos_u32: []u32 = undefined;

var scooters_aop: []*ScooterAoS = undefined;

var scooters_aop_u32: []u32 = undefined;

var scooters_count: u32 = 0;

var scooters_runtime_stride: u32 = 0;
```

Figure 20. The code of the structures used for the tests, in Zig.

```
export fn run_native(speed: u32) void {
    for (scooters_aos) |*scooter| {
        scooter.*.distance +%= speed;
    }
}
```

Figure 21. Code implementing the native test, in Zig.

Fixed: This has the same objective but uses "scooter_aos_u32" and directly utilizes "u32" instead of the language's native struct. This would resemble an implementation of reading serialized binary data. Figure 22 demonstrates the operation of this test, with the values of stride and offset constant at compilation time.

```
export fn run_fixed(speed: u32) void {
    const stride = 2;
    const offset = 1;

    var i: u32 = offset;
    var max: u32 = scooters_count * stride;
    while (i < max): (i += stride) {
        scooters_aos_u32[i] +%= speed;
    }
}
```

Figure 22. Code implementing the fixed test, in Zig.

Stride: Similar to the fixed test, but the jump sizes are not known at compilation time and are defined at runtime in a global variable "scooters_runtime_stride". Figure 23 is similar to Figure 3, with the difference that the stride is not a constant and is initialized at runtime, thus preventing potential optimizations.

```
1. export fn run_stride(speed: u32) void {
2.     var stride = scooters_runtime_stride;
3.     const offset = 1;
4.
5.     var i: u32 = offset;
6.     var max: u32 = scooters_count * stride;
7.     while (i < max): (i += stride) {
8.         scooters_aos_u32[i] +%= speed;
9.     }
10. }
```

Figure 23. Code implementing the stride test, in Zig.

Fixed+SIMD: This has an implementation identical to fixed but uses the manual vectorization feature of the Zig programming language. This allows for SIMD usage, consequently adding two elements at a time, which are the second and fourth elements of a lane of 4xu32.

Figure 24 represents the implementation of this test. Since the size is fixed, we know that the second number is the distance. SIMD can operate with up to four numbers of "u32", so it is possible to create a mask and add four numbers. However, we need to ignore the "id". To do so, we use zero, then summed with zero, as exemplified in Figure 25.

```
1. export fn run_simd(speed: u32) void {
2.     const stride = 4;
3.     var speed_mask: @Vector(stride, u32) = [stride]u32{ 0, speed, 0, speed };
4.
5.     var i: u32 = 0;
6.     var max: u32 = scooters_count * (stride * @sizeOf(u32) / @sizeOf(ScooterAoS));
7.     while (i < max): (i += stride) {
8.         var distances: @Vector(stride, u32) = scooters_aos_u32[i..][0..stride].*;
9.
10.         distances = distances + speed_mask;
11.
12.         scooters_aos_u32[i..][0..stride].* = distances;
13.     }
14. }
```

Figure 24. Code implementing the fixed+SIMD test, in Zig.

Figure 25. Example of two SIMD lanes being added (fixed+SIMD test).

SoA: This performs the same functionality as the other tests but uses a different form of data organization, as demonstrated in Figure 26, where each field becomes a vector.

SoA+SIMD: Similar to fixed+SIMD, this uses the SoA data organization structure but manually employs SIMD using the features available in Zig. This function, unlike Fixed+SIMD, allows the addition of four numbers at a time instead of two due to the continuity of the values.

Figure 27 demonstrates the implementation of this test. A mask is created, containing four numbers, and then added to another four numbers in a single instruction with the help of SIMD, as illustrated in Figure 28.

```
1. export fn run_soa(speed: u32) void {
2.     var i: u32 = 0;
3.
4.     while (i < scooters_count): (i += 1) {
5.         scooters_soa.distances[i] +%= speed;
6.     }
7. }
```

Figure 26. Code implementing the SoA test, in Zig.

```
1. export fn run_soa_simd(speed: u32) void {
2.     const stride = 4;
3.     var speed_mask: @Vector(stride, u32) = [_]u32{ speed, speed, speed, speed };
4.
5.     var i: u32 = 0;
6.     while (i < scooters_count): (i += stride) {
7.         var          distances:          @Vector(stride,          u32)          =
scooters_soa.distances[i..][0..stride].*;
8.
9.         distances = distances + speed_mask;
10.
11.         scooters_soa.distances[i..][0..stride].* = distances;
12.     }
13. }
```

Figure 27. Code implementing the SoA+SIMD test, in Zig.

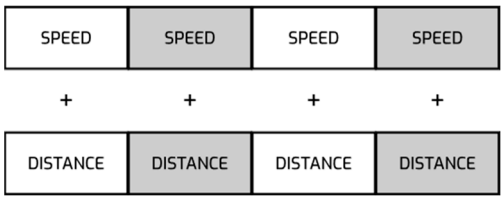

Figure 28. Example of two SIMD lanes being added (SoA+SIMD test).

AoP: Finally, AoP uses a different data structure, where all the values in a vector are pointers to the structs. This is also the traditional organization in object-oriented languages. Figure 29 demonstrates the implementation of this test.

```
1. export fn run_aop(speed: u32) void {
2.     for (scooters_aop) |scooter| {
3.         scooter.*.distance +%= speed;
4.     }
5. }
```

Figure 29. Code implementing the AoP test, in Zig.

5.1.3. Dynamic Structures Test Results

All tests were conducted as defined in Sections 5.1.1 and 5.1.2, and the results were obtained using Go and benchstat [69]. Figure 30 shows the comparison with different types of implementations using the same amount of data. In general, it confirms that the Stride version is the slowest among all methods. The Fixed+SIMD has less precise results but typically maintains the performance of the native, which confirms the use of auto-vectorization in the native version.

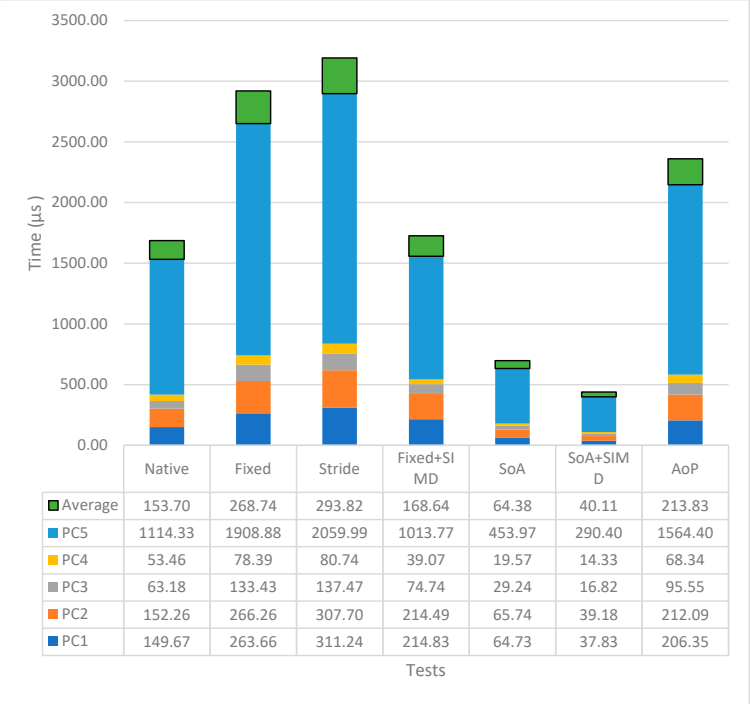

Figure 30. Performance comparison chart between different tests and machines, running the code described at Sections 5.1.1 and 5.1.2. PC1, PC2, PC3, PC4, and average use the vertical scale on the left, while PC5 uses the vertical scale on the right, with time in microseconds. The *X*-axis describes each type of test: "Native", "Fixed", "Stride", "Fixed+SIMD", "SoA", "SoA+SIMD", and "AoP", as previously mentioned in Section 5.1.2.

The results of this test coincide with the results of another study [65], thus demonstrating that SoA, even without any manual optimization, offers better performance across all architectures. The SoA+SIMD implementation manages to be even more efficient due to a simplified SIMD implementation. This indicates that this data layout is particularly efficient for the type of computations being measured. This efficiency can be attributed to better cache utilization, reduced memory access overhead, and the ability to leverage SIMD instructions effectively. Notably, this performance improvement was consistent across all tested CPU architectures.

An extra observation can be made about PC5, an Azure virtual machine with low computational power, which shows the same performance gains as the others. Furthermore, the SoA test on PC5 takes an average of 290 microseconds, while on PC1, it takes 311 microseconds in the dynamic size test ("Stride").

Table 3 summarizes the performance differences, in proportion, relative to the native. Table 4 shows the variation in performance between different runs of the same test, which highlights the performance instability of fixed+SIMD on PC1 and PC2.

Table 3. Performance comparison between different tests.

	Fixed	Stride	Fixed+SIMD	SoA	SoA+SIMD	AoP
PC1	+76.15%	+107.95%	~	−56.75%	−74.72%	+37.87%
PC2	+74.87%	+102.09%	~	−56.82%	−74.27%	+39.30%
PC3	+111.19%	+117.58%	+18.29%	−53.72%	−73.37%	+51.22%
PC4	+46.64%	+51.03%	−26.91%	−63.39%	−73.19%	+27.84%
PC5	+71.30%	+84.86%	−9.02%	−59.26%	−73.94%	+40.39%
M	+74.85%	+91.17%	+9.72%	−58.12%	−73.91%	+39.12%

Table 4. Comparison of performance difference between executions.

	Native	Fixed	Stride	Fixed+SIMD	SoA	SoA+SIMD	AoP
PC1	±1%	±2%	±14%	±38%	±1%	±1%	±1%
PC2	±1%	±1%	±11%	±36%	±1%	±2%	±0%
PC3	±0%	±3%	±4%	±9%	±6%	±1%	±1%
PC4	±1%	±1%	±1%	±0%	±1%	±1%	±0%
PC5	±1%	±1%	±0%	±1%	±1%	±1%	±1%

5.2. Generated Code

To evaluate the performance of Karmem, a prototype was developed with the purpose of generating the codes and a simplified runtime. As a prototype, it contains the main functionalities, but not all of them as yet.

Testing the performance is a complex task, as it may be biased and favorable to Karmem. For this very reason, Cap'N'Proto mentions that it does not perform or publish benchmarks [70]. Moreover, the current state of the project may not represent the performance of the finished project, considering that changes for security and stability reasons may impact performance. Therefore, comparisons will be made against FlatBuffers and JSON.

5.2.1. Experiment

Considering the structures defined in Figures 31 and 32, the code generated by Karmem differs from the code generated by FlatBuffers, and, thus, results in distinct usage. One of the major differences is the way data are initialized. Karmem uses top-down initialization, while FlatBuffers has a bottom-up approach. Figures 33 and 34 show an example of how to interact with the generated code to serialize data. A similar method is used for reading data. Other notable differences can be observed in terms of error handling, the use of SoA, and Karmem's conversion to vectors without copying.

FlatBuffers lacks any native support for WASM or ways to create exported or imported functions. For this reason, to keep the comparison focused on read/write performance, no such features were used in Karmem. All tests were conducted by exporting and copying information to the WASM module's memory.

```
namespace BatchTemperature;

table TemperatureReading {
  timestamp: long;
  value: float;
}

table TemperatureBatch {
  readings: [TemperatureReading];
}

table AnalysisResult {
  averageTemperature: float;
  maxTemperature: float;
  minTemperature: float;
}

root_type TemperatureBatch;
root_type AnalysisResult;
```

Figure 31. Description of structure in FlatBuffers.

```
struct temperature_reading (dynamic) {
  timestamp: i64;
  value: f32;
}

struct temperature_batch (dynamic) {
  readings: []temperature_reading;
}

struct analysis_result (dynamic) {
  average_temperature: f32;
  max_temperature: f32;
  min_temperature: f32;
}
```

Figure 32. Description of structure in Karmem.

```
func ToFlatBuffer(data data.TemperatureBatch) []byte {
        builder:= FlatBuffers.NewBuilder(1_000_000)

        // Create temperature readings in FlatBuffers
        readingOffsets:= make([]FlatBuffers.UOffsetT, len(data.Readings))
        for i:= range data.Readings {
                fn.TemperatureReadingStart(builder)
                fn.TemperatureReadingAddTimestamp(builder,
data.Readings[i].Timestamp)
                fn.TemperatureReadingAddValue(builder, data.Readings[i].Value)
                readingOffsets[i] = fn.TemperatureReadingEnd(builder)
        }

        // Create an array of readings
        fn.TemperatureBatchStartReadingsVector(builder, len(readingOffsets))
        for i:= len(readingOffsets) - 1; i >= 0; i-- {
                builder.PrependUOffsetT(readingOffsets[i])
        }
        readings:= builder.EndVector(len(readingOffsets))

        // Create the batch
        fn.TemperatureBatchStart(builder)
        fn.TemperatureBatchAddReadings(builder, readings)
        batch:= fn.TemperatureBatchEnd(builder)

        builder.Finish(batch)

        return builder.FinishedBytes()
}

```

Figure 33. Example of serialization with FlatBuffers.

5.2.2. Performance

In initial tests, even without any use of shared memories, Karmem demonstrates superior performance to JSON and FlatBuffers. The size of the generated WASM is also slightly smaller compared to FlatBuffers, even with excessive use of inlining. Table 5 shows the performance comparison for writing data (Figures 33 and 34), assuming different sizes for the internal vector.

```
1. func ToKarmem(data data.TemperatureBatch) []byte {
2.         builder:= fn.NewSegmentRetained(1_000_000)
3.         reader:= builder.Reader()
4.
5.         // Create the batch
6.         batch, ok:= fn.NewTemperatureBatchWriter(&reader)
7.         if !ok {
8.                   panic("not ok")
9.         }
10.
11.        // Create an array of readings
12.        readings, ok:= batch.Readings(&reader, uint32(len(data.Readings)))
13.        if !ok {
14.                  panic("not ok")
15.        }
16.
17.        // Create temperature readings in Karmem
18.        timestamps, ok:= readings.Timestamp(&reader)
19.        if !ok {
20.                  panic("not ok")
21.        }
22.        values, ok:= readings.Value(&reader)
23.        if !ok {
24.                  panic("not ok")
25.        }
26.
27.        for i:= range data.Readings {
28.                  timestamps[i] = data.Readings[i].Timestamp
29.                  values[i] = data.Readings[i].Value
30.        }
31.
32.        return builder.Bytes()
33. }
```

Figure 34. Example of serialization with Karmem.

Table 5. Comparison of serialization time, where the size represents the number of items in the "Readings" vector.

Size	Karmem	FlatBuffers	JSON
10	194.9 ns	1215.5 ns	5383.0 ns
10,000	77.19 µs	928.70 µs	4308.86 µs

The WASM module should also read and process data efficiently, and Karmem enables this in two ways: a copy-free reading format and memory sharing. Additionally, a performance advantage arises from the reduced indirection required due to the SoA layout. The code generated by Karmem can be manipulated as native homogenous arrays, allowing the compiler to easily optimize it. Table 6 shows the comparison of data-processing time described in Figures 31 and 32. The comparison with JSON was excluded because TinyGo has additional host resources to use JSON.

Table 6. Comparison of reading serialized data as a WASM module.

Size	Karmem	FlatBuffers	JSON
10	316.4 ns	5787.0 ns	-
10,000	68.15 µs	1325.57 µs	-

The code generated by Karmem is extensive, consisting of 1286 lines, while FlatBuffers has 254 lines but requires an external library to function and is thus not self-contained. The size of the compiled WASM file is also different, with Karmem being the smallest. This is unexpected considering that the code contains various inlining and duplications. The module using Karmem is 8.7 KB, while FlatBuffers is 9.1 KB, and JSON requires 262 KB. JSON uses runtime reflection techniques, which make it not only inefficient but also significantly larger.

Karmem achieves better performance, even with the use of bound checking, which is not present in FlatBuffers. Additionally, all structures used in the example are dynamic, which corresponds exactly with the FlatBuffers table. Preliminary tests indicate that adding new fields to an existing struct does not significantly impact performance, as these additional fields can be entirely ignored by the reader.

5.2.3. Integrity

Since Karmem generates code for multiple programming languages and facilitates cross-language communication through WASM via the Karmem runtime, tests are conducted using Karmem itself. The primary objective of these tests is to ensure compatibility and data integrity across languages. While this testing approach introduces potential challenges related to bootstrapping and bug propagation, it provides the flexibility to dynamically modify and compile new versions of the same schema.

Figure 35 illustrates the interface used for communication between modules, whose host is responsible to call the producer and then call all consumers. In this setup, a producer generates an arbitrary message, serializes it using Karmem, and the corresponding consumers parse and process the message. The processing involves reading all fields, converting each value to a string, and then adding the converted value to a hash digest, which is SHA-2. The final hash result is then compared with the expected checksum provided by the producer. The rationale behind converting to strings is to ensure accurate value interpretation, instead of relying on just the binary representation.

The integrated testing framework currently supports the creation and modification of simple schemas, iterating and testing all fields. It also accommodates manually written tests and allows for the reuse of pre-compiled WASM modules, which is crucial for ensuring backward compatibility between different versions of Karmem. Additionally, the framework can incorporate malicious producers to generate specially crafted messages, helping identify potential vulnerabilities.

Due to the deterministic behavior of WASM and the lack of access to OSes features, such as date–time and random number generators, tests are always deterministic, and random data can only be derived from the entropy of "producer_input". In case of a bug, it is possible to regenerate the exact output. Tests must be conducted on each Karmem update, but not exclusively. It also should be executed when new compilers and language versions are available, ensuring forward compatibility with upstream changes.

5.2.4. Usage

Karmem is being used in a multiplayer game currently under development. In this game, players take on the role of a programmer, where each player can write their own code to control a character. This use case requires multiple modules to read the same game state, which highlights one of the best uses of shared memory provided by Karmem and the isolation and safety of WASM, allowing for untrusted code to be run without locking into a single programming language.

Comparing against existent alternatives is difficult. Technologies like waPC do not handle communications between WASM modules, do not provide zero-copy serialization, and do not handle shared memory. The component model is not currently supported by Wazero or WAMR, and few languages support the component model. It also does not offer shared memory.

```
struct producer_input (dynamic) {
        entropy: []u8;
}

struct producer_result (dynamic) {
        field_path: []u8;
        checksum: []u8;
}

struct producer_output (dynamic) {
        message: []u8;
        results: []producer_result;


struct consumer_input (dynamic) {
        message: []u8;
}

struct consumer_result (dynamic) {
        field_path: []u8;
        checksum: []u8;
}

struct consumer_output (dynamic) {
        results: []consumer_result;
}

interface test_producer (dynamic, single) {
        produce: (producer_input) -> (producer_output);
}

interface test_consumer (dynamic, fanout) {
        shared: memory (import: write, export: read, 1024);
        out: memory (import: read, export: write, 1);

        consume: (consumer_input @ shared) -> (consumer_output @ out);
}
```

Figure 35. Definition of test interface with Karmem.

Karmem supports multiple languages but has usability challenges like Object–Relational Impedance Mismatch (ORIM) [71]. Although ORIM is typically associated with databases, Karmem encounters comparable issues due to the incompatibility between different programming paradigms. Specifically, it lacks object identity, meaning identical datasets are not distinguished as separate entities, unlike in languages such as Java, Swift, and C#. The absence of encapsulation allows unrestricted access and the modification of shared data across modules, which contrasts with object-oriented.

Additionally, while Karmem's approach to avoiding data copies enhances performance, it can come at the cost of usability. Developers may need to manually manage data copying and fully understand the implications of this behavior, adding complexity to the development process and potential issues like Use-After-Free (UAF). In some languages, strings, fixed-size arrays, and similar data types are assumed to be immutable. However, the generated code may use type punning from shared memory, which lacks the immutability property, further complicating the handling of data and increasing the potential for errors. Another issue is the potential misuse of set/write operations over read-only shared memory.

Unlike FlatBuffers and Cap'n Proto, Karmem does not support default custom values, resulting in a default value of zero or its equivalent. Consequently, relying on such default

values in a meaningful way is not recommended. Checking the existence of a field of dynamic structs is possible but not publicly exposed by the generated code.

6. Conclusions and Future Work

Karmem represents an innovation in data communication within WASM and demonstrates the ability to overcome challenges of efficiency and interoperability. This project developed its own IDL and an optimized communication protocol to facilitate interaction between WASM modules. Karmem contributes to WASM development by offering solutions for data communication between modules in different programming languages. The innovations have practical implications for distributed applications based on WASM, making them more efficient and interoperable. With the implementation of a code generator that supports multiple programming languages, Karmem mitigates one of WASM's inefficiencies. This paves the way for a wide range of applications, from games to complex systems. The main limitation faced by the project is the inherent complexity of supporting different programming languages and the need for continuous optimization to maintain security and efficiency. Therefore, future research should focus on enhancing security and performance and investigating the possibility of expanding to more programming languages. The impact of Karmem on WASM application development is significant and marks an important step toward more efficient and secure data interaction in constrained execution environments.

However, there are additional aspects that require more in-depth study related to usability and testing. First, Karmem uses a custom IDL. Therefore, to improve the user experience, the Language Server Protocol (LSP) should be developed. It could greatly assist developers by providing features such as autocomplete, syntax highlighting, and error detection within Integrated Development Environments (IDEs). This would streamline the coding process and reduce errors, especially for those unfamiliar with KarmemIDL.

Second, Karmem already uses continuous integration/continuous deployment (CI/CD) processes with tests written for both the code generator and individual language implementations. However, improving test coverage is crucial, especially because Karmem uses unsafe features, bypasses type protection, and uses type punning, which depends on the specific memory layout of each language. These techniques are often employed to enhance performance and usability, but we acknowledge that they introduce potential security risks.

Continuous testing is not solely in response to updates in the Karmem source code, but it is also crucial for maintaining compatibility with evolving languages and compilers. Regular tests are conducted using known good, malicious, and malformed data to ensure that the system behaves as expected under various conditions. These tests are designed to verify that the guest does not read data out of bounds, modify read-only memory, or cause resource exhaustion. Additionally, they ensure that all modules are correctly initialized and mitigation techniques (timeouts, memory size limits) function as intended.

Continuous fuzzing must be implemented to identify potential security vulnerabilities, and the usage of limited memory and WASM runtime can help identify issues previously described. Additionally, Karmem runtime relies on an existing WASM runtime, Karmem runtime is responsible for memory management mapping and invoking WASM functions. This separation of responsibilities reduces the attack surface by delegating complex execution tasks to a well-tested runtime, thereby minimizing the potential for vulnerabilities within Karmem itself.

To facilitate quick experimentation with Karmem, a dedicated website will be developed. The existing code generator is already compatible with WASM, enabling users to generate code from IDL directly in the browser without the need for any additional software.

At present, Karmem is a prototype that does not have all of its features implemented due to time constraints and the need for viability validation. Therefore, it is still necessary to implement all the features described in this document and review the existing implementations.

The development of these future work lines, along with the implementation of a use-case scenario for the Karmen application, will play an important role in confirming performance improvements and assessing the generalization of the results.

Author Contributions: Conceptualization, L.S.; methodology, L.S.; software, L.S.; investigation, L.S.; supervision, J.M., F.R.; funding acquisition, J.M., F.R.; writing—original draft preparation L.S., J.M., F.R.; writing—review and editing L.S., J.M., F.R. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data is contained within the article.

Acknowledgments: This work was funded by National Funds through the Foundation for Science and Technology (FCT), I.P., within the scope of the project UIDB/05583/2020 and DOI identifier https://doi.org/10.54499/UIDB/05583/2020. Furthermore, we would like to thank the Research Centre in Digital Services (CISeD) and the Instituto Politécnico de Viseu for their support.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Dis Virtual Machine Specification. Available online: https://www.vitanuova.com/inferno/papers/dis.html (accessed on 16 October 2023).
2. Rose, J.R. Bytecodes meet Combinators: Invokedynamic on the JVM. In Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages, Orlando, FL, USA, 25–29 October 2009; pp. 1–11.
3. Hickey, P. Lucet Takes WebAssembly Beyond the Browser | Fastly | Fastly. Available online: https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime (accessed on 16 October 2023).
4. Varda, K. WebAssembly on Cloudflare Workers. Available online: https://blog.cloudflare.com/webassembly-on-cloudflare-workers/ (accessed on 16 October 2023).
5. Bytecodealliance/Wasm-Micro-Runtime: WebAssembly Micro Runtime (WAMR). Available online: https://github.com/bytecodealliance/wasm-micro-runtime (accessed on 16 October 2023).
6. Dale, B. Gavin Wood: WebAssembly Is the Future, EVM Is Right Now—CoinDesk. Available online: https://www.coindesk.com/tech/2021/05/25/polkadots-gavin-wood-webassembly-is-the-future-of-smart-contracts-but-legacy-evm-is-right-now/ (accessed on 16 October 2023).
7. Charriere, P. Give Super Powers to Java with WebAssembly by Philippe Charriere @ Wasm I/O 2023—YouTube. Available online: https://www.youtube.com/watch?v=5HBglrvHtWg (accessed on 16 October 2023).
8. Shuralyov, D. Syscall/js: Performance Considerations—Issue #32591—Golang/Go. Available online: https://github.com/golang/go/issues/32591 (accessed on 16 October 2023).
9. Denis. [Interface Types] Avoid Memory Copy—Issue #88—WebAssembly/Interface-Types. Available online: https://github.com/WebAssembly/interface-types/issues/88 (accessed on 16 October 2023).
10. Sikora, P. WebAssembly in Envoy. Available online: https://github.com/proxy-wasm/spec/blob/master/docs/WebAssembly-in-Envoy.md#drawbacks (accessed on 16 October 2023).
11. Fioretti, M. How and Why to Link WebAssembly Modules. Available online: https://training.linuxfoundation.org/blog/how-and-why-to-link-webassembly-modules/ (accessed on 16 October 2023).
12. WebAssembly | web.dev. Available online: https://web.dev/explore/webassembly?hl=pt-br (accessed on 16 October 2023).
13. Intel Confirms Arrow Lake-S & Lunar Lake CPUs will Support Instructions for AVX-VNNI, SHA512, SM3, SM4 and LAM—VideoCardz.com. Available online: https://videocardz.com/newz/intel-confirms-arrow-lake-s-lunar-lake-cpus-will-support-instructions-for-avx-vnni-int16-sha512-sm3-sm4-and-lam (accessed on 18 November 2023).
14. Design/Nondeterminism.md at Main—WebAssembly/Design—GitHub. Available online: https://github.com/WebAssembly/design/blob/main/Nondeterminism.md (accessed on 10 November 2023).
15. WIT—The WebAssembly Component Model. Available online: https://component-model.bytecodealliance.org/design/wit.html#options (accessed on 26 January 2023).
16. WASI/legacy/preview1/witx/wasi_snapshot_preview1.witx. Available online: https://github.com/WebAssembly/WASI/blob/41c4383548ba7a06df5df7232b68a6f0bbb93e2d/legacy/preview1/witx/wasi_snapshot_preview1.witx (accessed on 16 October 2023).
17. Denis, F. Jedisct1/Witx-Codegen: WITX Code and Documentation Generator for AssemblyScript, Zig, Rust and More. Available online: https://github.com/jedisct1/witx-codegen (accessed on 16 October 2023).
18. waPC.io | waPC.io. Available online: https://wapc.io/ (accessed on 16 October 2023).
19. Fukuda, T. knqyf263/go-plugin: Go Plugin System over WebAssembly. Available online: https://github.com/knqyf263/go-plugin (accessed on 16 October 2023).

20. Canonical ABI—The WebAssembly Component Model. Available online: https://component-model.bytecodealliance.org/design/canonical-abi.html (accessed on 26 January 2023).
21. WebAssembly Contributors. WebAssembly/Proposals: Tracking WebAssembly Proposals. Available online: https://github.com/WebAssembly/proposals (accessed on 17 October 2023).
22. Overview | Extism—Make all Software Programmable. Extend from within. Available online: https://extism.org/docs/overview/#example-in-action (accessed on 16 October 2023).
23. Dylibso. Host Functions | Extism—Make All Software Programmable. Extend from within. Available online: https://extism.org/docs/concepts/host-functions/ (accessed on 16 October 2023).
24. Wasmbus (Stable ABI) | WasmCloud. Available online: https://wasmcloud.com/docs/category/wasm-abis (accessed on 16 October 2023).
25. Kanev, S.; Darago, J.P.; Hazelwood, K.; Ranganathan, P.; Moseley, T.; Wei, G.Y.; Brooks, D. Profiling a warehouse-scale computer. In Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, 13–17 June 2015; ACM: New York, NY, USA, 2015; pp. 158–169. [CrossRef]
26. Raghavan, D.; Levis, P.; Zaharia, M.; Zhang, I. Breakfast of Champions: Towards Zero-Copy Serialization with NIC Scatter-Gather. In Proceedings of the Workshop on Hot Topics in Operating Systems, Providence, RI, USA, 22–24 June 2023; pp. 199–205. [CrossRef]
27. Sriraman, A.; Dhanotia, A. Accelerometer: Understanding acceleration opportunities for data center overheads at Hyperscale. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems—ASPLOS, Lausanne, Switzerland, 16–20 March 2020; pp. 733–750. [CrossRef]
28. Google. Tutorials | Protocol Buffers Documentation. Available online: https://protobuf.dev/getting-started/ (accessed on 22 January 2023).
29. Furuhashi, S. MessagePack: It's like JSON. But Fast and Small. Available online: https://msgpack.org/ (accessed on 22 January 2023).
30. Google. FlatBuffers: Use in Rust. Available online: https://flatbuffers.dev/flatbuffers_guide_use_rust.html (accessed on 22 January 2023).
31. Google. FlatBuffers: Platform/Language/Feature Support. Available online: https://flatbuffers.dev/flatbuffers_support.html (accessed on 22 January 2023).
32. Cap'n Proto: Encoding Spec. Available online: https://capnproto.org/encoding.html#security-considerations (accessed on 22 January 2023).
33. OSCON 2010: Rob Pike, 'Public Static Void'—YouTube. Available online: https://www.youtube.com/watch?v=5kj5ApnhPAE (accessed on 17 December 2023).
34. Llopis, N. Data-Oriented Design (Or Why You Might Be Shooting Yourself in the Foot with OOP)—Games from within. Available online: https://gamesfromwithin.com/data-oriented-design (accessed on 24 November 2023).
35. DICE. A Step Towards Data Orientation | PPT. Available online: https://pt.slideshare.net/DICEStudio/a-step-towards-data-orientation (accessed on 22 January 2024).
36. Acton, M. CppCon 2014: Mike Acton 'Data-Oriented Design and C++'—YouTube. Available online: https://www.youtube.com/watch?v=rX0ItVEVjHc (accessed on 22 January 2024).
37. DICE. Introduction to Data Oriented Design | PPT. Available online: https://pt.slideshare.net/DICEStudio/introduction-to-data-oriented-design (accessed on 22 January 2024).
38. DOTS—Unity's Data-Oriented Technology Stack. Available online: https://unity.com/dots (accessed on 26 November 2023).
39. C# Language Support | Burst | 1.8.11. Available online: https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual/csharp-language-support.html (accessed on 26 November 2023).
40. Odin Programming Language. Available online: https://odin-lang.org/ (accessed on 11 November 2023).
41. Jai Programming Language Resources and Information. Available online: https://inductive.no/jai/ (accessed on 26 November 2023).
42. Home ⚡ Zig Programming Language. Available online: https://ziglang.org/ (accessed on 11 November 2023).
43. Richard, F. *Data-Oriented Design: Software Engineering for Limited Resources and Short Schedules*, 1st ed.; Ingram International Inc.: La Vergne, TN, USA, 2018; Volume 1.
44. Straume, P.-M. Investigating Data-Oriented Design. 2019. Available online: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2677763 (accessed on 25 November 2023).
45. Karlsson, M. Evaluation of Object-Space Occlusion Culling with Occluder Fusion. 2011. Available online: https://www.semanticscholar.org/paper/Evaluation-of-Object-Space-Occlusion-Culling-with-Karlsson/a8e7758df9f651e90e532863766861328b30c199 (accessed on 11 November 2023).
46. Wells, A.M.; Prabha, A.M. Intel® HPC Developer Conference Fuel Your Insight Improve Vectorization Efficiency Using Intel Simd Data Layout Template (Intel Sdlt). 2016. Available online: https://www.intel.com/content/dam/www/public/us/en/documents/presentation/improving-vectorization-efficiency.pdf (accessed on 12 December 2023).
47. Albrecht, T. *Pitfalls of Object Oriented Programming*; M & T Books: Buffalo, NY, USA, 2014.
48. Homann, H.; Laenen, F. SoAx: A generic C++Structure of Arrays for handling particles in HPC codes. *Comput. Phys. Commun.* **2018**, *224*, 325–332. [CrossRef]

49. Data Alignment to Assist Vectorization. Available online: https://www.intel.com/content/www/us/en/developer/articles/technical/data-alignment-to-assist-vectorization.html (accessed on 13 January 2024).
50. Eberhardt, C. The State of WebAssembly 2023. Available online: https://blog.scottlogic.com/2023/10/18/the-state-of-webassembly-2023.html (accessed on 16 October 2023).
51. Assume_no_Moving_gc Package—Go4.org/unsafe/assume-no-moving-gc—Go Packages. Available online: https://pkg.go.dev/go4.org/unsafe/assume-no-moving-gc#section-readme (accessed on 17 January 2024).
52. A08 Software and Data Integrity Failures—OWASP Top 10:2021. Available online: https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/ (accessed on 25 November 2023).
53. Deserialization of Untrusted Data | OWASP Foundation. Available online: https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data (accessed on 25 November 2023).
54. Krill, P. Oracle Plans to Dump Risky Java Serialization | InfoWorld. Available online: https://news.ycombinator.com/item?id=17165800 (accessed on 25 November 2023).
55. HTML Standard. Available online: https://html.spec.whatwg.org/multipage/webstorage.html#sensitivity-of-data (accessed on 23 November 2023).
56. RabbitMQ tutorial—Routing—RabbitMQ. Available online: https://www.rabbitmq.com/tutorials/tutorial-four-spring-amqp.html (accessed on 31 January 2024).
57. Pankaj. Gangs of Four (GoF) Design Patterns | DigitalOcean. Available online: https://www.digitalocean.com/community/tutorials/gangs-of-four-gof-design-patterns (accessed on 31 January 2024).
58. Design Patterns in Dynamic Programming. Available online: https://kidneybone.com/c2/wiki/DesignPatternsInDynamicProgramming (accessed on 31 January 2024).
59. Are Design Patterns Missing Language Features. Available online: https://kidneybone.com/c2/wiki/AreDesignPatternsMissingLanguage (accessed on 31 January 2024).
60. Amazon. Restrictions on Lambda@Edge—Amazon CloudFront. Available online: https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/lambda-at-edge-function-restrictions.html (accessed on 31 January 2024).
61. Netlify. Functions Overview | Netlify Docs. Available online: https://docs.netlify.com/functions/overview/ (accessed on 31 January 2024).
62. Vercel. Edge Functions Limitations. Available online: https://vercel.com/docs/functions/edge-functions/limitations (accessed on 31 January 2024).
63. Surma. Bringing Javascript to WebAssembly for Shopify Functions. 2024. Available online: https://shopify.engineering/javascript-in-webassembly-for-shopify-functions (accessed on 31 January 2024).
64. Server Operating System Market Volume, Share | Analysis, 2030. Available online: https://www.fortunebusinessinsights.com/server-operating-system-market-106601 (accessed on 29 October 2023).
65. Nyberg, F. Investigating the Effect of Implementing Data-Oriented Design Principles on Performance and Cache Utilization, UMEÅ University. 2021. Available online: https://www.diva-portal.org/smash/get/diva2:1578616/FULLTEXT01.pdf (accessed on 29 October 2023).
66. Mytkowicz, T.; Diwan, A.; Hauswirth, M.; Sweeney, P.F. Producing Wrong Data without Doing Anything Obviously Wrong! *ACM Sigplan Not.* **2009**, *44*, 265–276. [CrossRef]
67. Série B expansível—Máquinas Virtuais do Azure—Azure Virtual Machines | Microsoft Learn. Available online: https://learn.microsoft.com/pt-pt/azure/virtual-machines/sizes-b-series-burstable (accessed on 26 January 2024).
68. Test Package—Cmd/go/internal/test—Go Packages. Available online: https://pkg.go.dev/cmd/go/internal/test (accessed on 24 January 2024).
69. Benchstat Command—Golang.org/x/perf/cmd/benchstat—Go Packages. Available online: https://pkg.go.dev/golang.org/x/perf/cmd/benchstat (accessed on 26 January 2024).
70. Cap'n Proto: Introduction. Available online: https://capnproto.org/ (accessed on 31 January 2024).
71. Neward, T. The Vietnam of Computer Science. 2006. Available online: https://www.odbms.org/wp-content/uploads/2013/11/031.01-Neward-The-Vietnam-of-Computer-Science-June-2006.pdf (accessed on 8 November 2023).