

# Enhancing Embedded Systems Development with TS-

**Xingzi Yu**

Shanghai Jiao Tong University

**Wei Tang**

Ant Group

**Tianlei Xiong**

Shanghai Jiao Tong University

**Wengang Chen**

Ant Group

**Jie He**

Ant Group

**Bin Yang**

Ant Group

**Zhengwei Qi** (✉ [qizhwei@sjtu.edu.cn](mailto:qizhwei@sjtu.edu.cn))

Shanghai Jiao Tong University

---

## Research Article

**Keywords:** keyword1, Keyword2, Keyword3, Keyword4

**Posted Date:** August 22nd, 2023

**DOI:** <https://doi.org/10.21203/rs.3.rs-3269068/v1>

**License:**   This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

**Additional Declarations:** No competing interests reported.

---

**Version of Record:** A version of this preprint was published at Automated Software Engineering on December 6th, 2023. See the published version at <https://doi.org/10.1007/s10515-023-00404-x>.

# Enhancing Embedded Systems Development with TS<sup>-</sup>

Xingzi Yu<sup>1</sup>, Wei Tang<sup>2</sup>, Tianlei Xiong<sup>1</sup>, Wengang Chen<sup>2</sup>,  
Jie He<sup>2</sup>, Bin Yang<sup>2</sup>, Zhengwei Qi<sup>1\*</sup>

<sup>1</sup>Shanghai Jiao Tong University, Shanghai, China.

<sup>2</sup>Ant Group, Shanghai, China.

\*Corresponding author(s). E-mail(s): [qizhwei@sjtu.edu.cn](mailto:qizhwei@sjtu.edu.cn);

Contributing authors: [editriendl@sjtu.edu.cn](mailto:editriendl@sjtu.edu.cn);

[tangwei.tang@antgroup.com](mailto:tangwei.tang@antgroup.com); [qmyyxtl@sjtu.edu.cn](mailto:qmyyxtl@sjtu.edu.cn);

[wengang.cwg@antgroup.com](mailto:wengang.cwg@antgroup.com); [hejie.he@antgroup.com](mailto:hejie.he@antgroup.com);

[yb261973@antgroup.com](mailto:yb261973@antgroup.com);

## Abstract

The lack of flexibility and safety in C language development has been criticized for a long time, causing detriments to the development cycle and software quality in the embedded systems domain. TypeScript, as an optionally-typed dynamic language, offers the flexibility and safety that developers desire. With the advancement of Ahead-of-Time (AOT) compilation technologies for TypeScript and JavaScript, it has become feasible to write embedded applications using TypeScript. Despite the availability of writing AOT compiled programs with TypeScript, implementing a compiler toolchain for this purpose requires substantial effort.

To simplify the design of languages and compilers, this paper presents a new compiler toolchain design methodology called TS<sup>-</sup>, which advocates the generation of target intermediate language code (such as C) from TypeScript rather than the construction of higher-level compiler tools and type systems on top of the intermediate language. TS<sup>-</sup> not only simplifies the design of the system but also provides developers with a quasi-native TypeScript development experience. This paper also presents Ts2WASM, a prototype that implements TS<sup>-</sup> and allows compiling a subset of language TypeScript to WebAssembly (WASM). The tests from the repository TypeScript show that Ts2WASM provides 3.8x as many features compared to the intermediate language (AssemblyScript). Regarding performance, Ts2WASM offers a significant speed-up of 1.4x to 19x. Meanwhile, it imposes over 65% less memory overhead compared to Node.js in most cases.

**Keywords:** keyword1, Keyword2, Keyword3, Keyword4

## 1 Introduction

In the domain of embedded system development, the debugging, compilation, and uploading of programs usually takes longer than in native environments. In this case, developers naturally write their code carefully to avoid errors. Although C is the most widely used programming language for constrained devices [1], recent studies [2, 3] suggest that it is prone to errors due to its use of pointers, manual memory management, and the potential for memory leaks.

In recent years, embedded developers have been increasingly turning to languages other than C for their development needs, such as JavaScript [4] and Python [5]. As a superset of JavaScript, TypeScript [6] has also gained increasing attention as a developing language for embedded systems [7, 8]. By incorporating TypeScript, developers can introduce a degree of type safety into their code, thus mitigating the risk of type errors. Additionally, developers can select to take advantage of dynamic typing in cases where it is necessary to maintain coding agility. The rich IDE support and third-party tools also boost the development process, e.g., code linting, formatting, and autocompletion.

There are currently two main approaches to TypeScript development in the embedded systems domain: (i) Generate JavaScript code from TypeScript, which is then executed by an Ahead-of-Time (AOT) compiler or interpreter [9–12]. And (ii) programming directly in a TypeScript-like language tailored to specific hardware and scenarios [7, 13, 14].

For approach (i), although Just-in-Time (JIT) compilers are powerful, they are not a suitable choice for embedded devices. Since embedded systems, particularly microcontroller units (MCUs), often lack the capability to support JIT compilers, as their memory is limited ( $\leq 1\text{MB}$ ). Additionally, a recent paper has demonstrated that Ahead-of-Time (AOT) compilation exhibits significantly superior performance compared to JavaScript interpreters in the context of embedded systems [7]. Therefore, approach (i) involves TypeScript and an available JavaScript AOT language variant.

TypeScript is intentionally designed with an unsound type system; Type annotations are optional. As such, approach (ii) similarly involves creating a variant for TypeScript that guarantees soundness, and the corresponding compiler framework, which is an arduous and time-consuming task. After investigating the AOT compilation for dynamic typing languages and the properties of various type systems, we observe an often overlooked insight that the gradual typing system of TypeScript can substantially aid the compilation progress, following which we can design new TypeScript-based languages for embedded devices in a trivial way. Instead of developing a type system for a new language, we build a compiler to transform the type system from TypeScript to a low-level language (e.g., C or C++). One key challenge is to preserve the dynamic features in TypeScript. Specifically, selective retention of

certain dynamic features is chosen to optimize the performance of the compiled code while sacrificing little flexibility.

In a nutshell, contrary to approaches (i) and (ii), we propose our approach, called  $\text{TS}^-$ , that does not require a dedicated type system but adheres to the original TypeScript compilation procedure, using `tsc` to check type errors and emit nominal typed intermediate language, which can eventually be AOT compiled into binaries.

Our contributions are listed as follows:

- Proposing  $\text{TS}^-$ , a language and compiler design methodology for embedded systems, and proving its feasibility from a design analysis (**Section 2**).
- Presenting a prototype Ts2WASM that uses TypeScript as the source language to emit AOT compiled binaries, which validates the proposed design TypeScript  $\text{TS}^-$  in practice (**Section 3**).
- Implementing five compiler passes to convert  $\text{TS}^-$  to a C++-like intermediate language (AssemblyScript) and emit WASM as the compilation target (**Section 4**).
- A series of comprehensive experiments to evaluate the performance of Ts2WASM and show that Ts2WASM is 1.4x to 19x faster than Node.js and imposes over 65% less memory overhead compared to Node.js in most cases (**Section 5**).

## 2 Motivation

This section presents the motivation for  $\text{TS}^-$ .

**Definition 1** ( $\sim$ ). Denote  $\sim X$  be the language with a syntax similar to  $X$ , which requires a different type system from that of  $X$ .

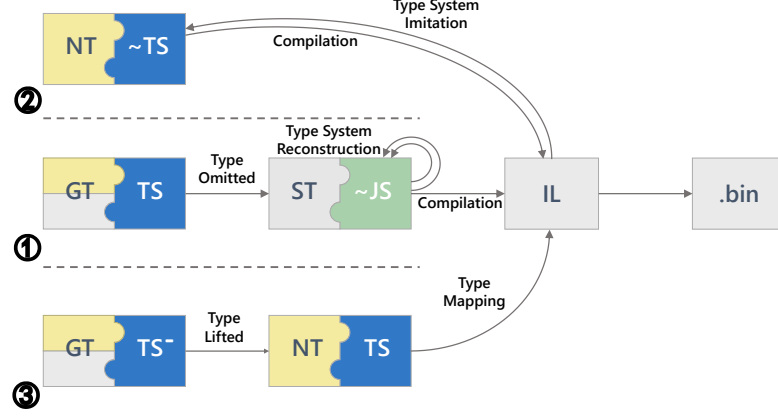
**Definition 2** (Gradual Typing, Structural Typing, and Nominal Typing). Gradual typing is proposed in [15] and adopted in languages such as TypeScript [16], GradualTalk [17], and C# [18]. Gradual typing allows the coexistence of dynamic typing (type checked at run-time) and static typing (type checked at compile time). Structural (property-based) typing and nominal (name-based) typing depend on how type compatibility is determined.

We use **GT**, **ST**, and **NT** as abbreviations for Gradual Typing, Structural Typing, and Nominal Typing, respectively.

Gradual typing allows developers to use type annotations for type safety, while also supporting unannotated code, allowing for increased efficiency and adaptability in software development. However, the gradual type system for TypeScript is intentionally unsound, leading to conflicts for AOT compilation.

**Definition 3** (AOT-Compatible Counterparts). Following Definition 1, we define  $\sim \text{TS}$  and  $\sim \text{JS}$  be the AOT-compatible counterparts of TypeScript and JavaScript. Languages compiled using AOT compilation typically employ nominal typing to achieve optimal performance.

In Definition 3, we make a general assumption for the type system design option that  $\sim \text{TS}$  and  $\sim \text{JS}$  compile to a nominal typing IL. Despite that a sound gradual type system for AOT compilation is feasible [19, 20], a recent research paper also [21] shows that such soundness comes at the cost of significant performance overhead. Subsequently, we refer to IL as being nominal typed for the remainder of this section.



**Fig. 1:**  $\sim$ TS,  $\sim$ JS and  $TS^-$ : 3 different compilation path for TypeScript development in embedded systems

## 2.1 Existing Methodology

The methodologies for implementing  $\sim$ TS and  $\sim$ JS are similar, that is, finding an IL, building a higher level language on top of it, and adding support for as many original language features as possible. Hop.js [9, 10] is an example of  $\sim$ JS that utilizes Scheme as its intermediate language, while StaticTS [7] as an example of  $\sim$ TS, uses C++. As shown in Figure 1, ① and ② are the compilation path for  $\sim$ JS and  $\sim$ TS, respectively. Type systems are distinguished by different colors, while GT is represented by a mixture of the colors for NT and ST.

For ①, developers write TypeScript and can use `tsc` to emit JavaScript, which is then fed to the compiler of  $\sim$ JS. To ensure compatibility with  $\sim$ JS, developers must modify the generated JavaScript code, which risks breaking the type safety provided by TypeScript. As  $\sim$ JS would be compiled into IL, type system reconstruction is needed. Furthermore, since all type information is discarded while generating JavaScript with `tsc`,  $\sim$ JS type system reconstruction involves redundant work for constructing discarded types.

A type system based on IL is designed for ②, and the type system imitation process is demonstrated in Figure 1.  $\sim$ TS utilizes an NT system similar to TypeScript while it is imitated from the IL’s NT. Given that TypeScript comes equipped with a powerful type system, it follows that  $\sim$ TS, which is a subset of TypeScript, only includes a relatively limited set of TypeScript features. As a result, a large portion of the language’s functionalities that depend on ST are missing.

## 2.2 Our proposed methodology

Prior to introducing  $TS^-$ , it is essential to understand that the process of compiling TypeScript AOT is not a straightforward one.

Taking into account the GT nature of TypeScript [15, 16], the code usually includes both ST and NT elements (as suggested in Figure 1). However, the nominal type systems for IL are non-trivial to represent the ST component, which typically involves dynamic typing code. As a result, many runtime functions are bundled into compilation at the performance cost or a maximum language subset is selected. The latter language subset is strictly not TypeScript, as it does not support a complete syntax or a standard compilation path of TypeScript.

In addition to ① and ②, we introduce a new approach called  $\text{TS}^-$ , which is illustrated in Figure 1 ③. Programmers write code in native TypeScript, use `tsc` to validate the type annotations, generate target code, and compile the target code into an executable binary. The differences are

- `tsc` is modified with additional rules to detect type errors and forbidden dynamic typing features.
- A part of ST is lifted to NT counterparts so that they can be AOT compiled.
- Runtime functions transform the commonly used features that are incompatible with type lifting.
- The NT-only Abstract Syntax Tree (AST) produced by type lifting can be easily mapped to an IL. However, in  $\sim\text{JS}$ , type reconstruction is necessary.
- The IL is not restricted to any particular language and can encompass class-based languages such as C++, Java, and C#.

$\text{TS}^-$  imitates not only a TypeScript source but also the compile path of TypeScript. Therefore, we coined the term  $\text{TS}^-$  to denote TypeScript itself with fewer features, different from languages that only write like TypeScript ( $\sim\text{TS}$ ,  $\sim\text{JS}$ ).

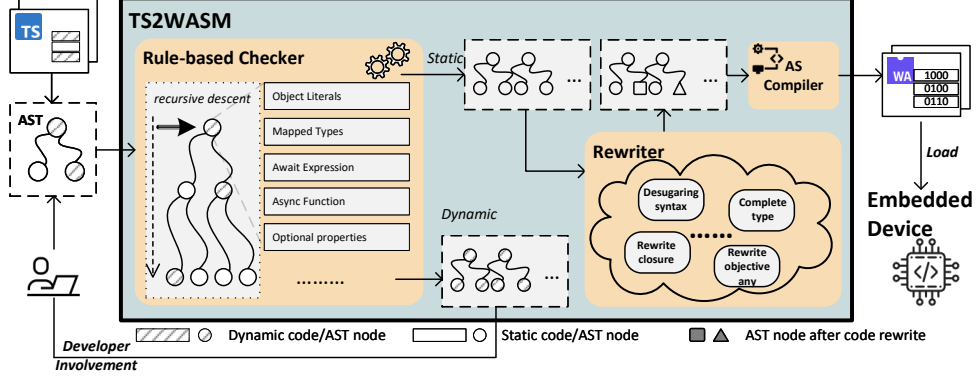
### 3 Implementation: Ts2Wasm

In this section, we present the design of Ts2WASM, a TypeScript to WASM compiler. We build Ts2WASM based on TypeScript, with AssemblyScript as the IL and Binaryen as the compiler from IL to binary. AssemblyScript is a class-based language similar to C++, and its source files can be compiled into WebAssembly (WASM) binaries. The selection of WASM as the AOT compilation target binary is primarily based on its portability across platforms, fast execution speed, and small code size - all desired features for developing embedded systems applications [22–24].

Ts2WASM is written in about 13000 lines of TypeScript. Figure 2 shows the architecture and basic execution workflow of Ts2WASM, which serves as a checker and compiler framework for TypeScript. The user-input source files are written in native TypeScript syntax while containing two types of code:

- **Static:** Code blocks suitable for AOT compilation with the TypeScript type system.
- **Dynamic:** Code blocks containing some highly dynamic features (mainly JavaScript) that *cannot* be AOT compiled.

Following the TypeScript compilation procedure, the user-input TypeScript files are first parsed into TypeScript ASTs with `tsc`, the TypeScript compiler. Then, Ts2WASM traverses the ASTs and picks out the nodes suitable for static compilation.



**Fig. 2:** Ts2WASM Design Overview: A compiler framework for TS<sup>-</sup> AOT compilation

Meanwhile, some AST transform operations are performed during the AST traversal, and for simplicity, we abstract the process into a **Checker** component. The Checker component (also a CLI tool for Ts2WASM) takes different actions depending on the type dynamic of code.

Dynamic nodes are represented by hatched circles in Figure 2 and are identified and collected by Checker. Subsequently, these dynamic nodes and the collected information (such as line number, node type, file name) are exported to the succeeding component. The nodes that have passed the check are classified as static and undergo AST rewriting in the **Rewriter** component of Ts2WASM. Additionally, since certain runtime functions are necessary to manage the selected dynamic features, a customized AssemblyScript compiler is utilized.

Also, note that in Figure 2, the rule-outed dynamic AST nodes are reported to the developer for manual review.

### 3.1 Nominal (Typing) Boundary Detection

We make an in-depth survey of the features unsupported by static compilation and divide the features into different categories, as Table 1 shows. The features are not fully listed for some categories due to space limitations.

When building the Ts2WASM Checker to filter the dynamic AST nodes, a complete version of Table 1 is applied to the AST visitor, as introduced in Section 4.1. Furthermore, we have categorized the features listed in Table 1 based on the reason for their exclusion.

- **Dynamic Typing:** The complexity of implementing dynamic semantics in a nominal type system excludes most of the features. For instance, the categories of types such as Type Manipulation, Narrowing, (dynamic) Expressions, and Interface are excluded.
- **JavaScript Compatibility:** In TypeScript, certain features are kept for JavaScript compatibility that may not be essential for AOT compilation.
- **Asynchronous Programming:** Asynchronous is vital in JavaScript programming to handle blocking operations. However, asynchronous is not supported in WASM.

**Table 1:** Ts2WASM Checker Feature Filter Table

| <i>Category</i>            | <b>Feature (Partially Displayed)</b>       |
|----------------------------|--|
| <i>Primitive Types</i>     | BigInt Types                               |
|                            | Unknown Types                              |
| <i>Advanced Types</i>      | Type Assertions                            |
|                            | Intersection Types                         |
| <i>Type Manipulation</i>   | Conditional Types                          |
|                            | Mapped Types                               |
| <i>Narrowing</i>           | Typeof Type Guards                         |
|                            | Equality Narrowing                         |
| <i>Expressions</i>         | Object Literals                            |
|                            | Computed Property                          |
|                            | Operator 'in'                              |
| <i>Statements</i>          | For-Await-Of Statement                     |
| <i>Functions / Methods</i> | Function Overload                          |
| <i>Interface</i>           | Interface Extending Class                  |
|                            | Interface Optional Property                |
| <i>Class</i>               | Class Optional Properties                  |
| <i>Enums</i>               | String Enums                               |
| <i>Async</i>               | Await Expression                           |
|                            | Asynchronous Function                      |
|                            | Promise                                    |
| <i>Generators</i>          | Yield Expression                           |
|                            | Generator Function                         |
| <i>Decorators</i>          | Accessor / Property / Parameter Decorators |
| <i>Namespaces</i>          | Namespace Merging                          |
|                            | Nested Namespace                           |
| <i>Modules</i>             | Importing Types                            |

- **Variable Scope:** This includes modules and namespaces, also absent in WASM.

By categorizing the language features, Ts2WASM can provide developers with more valuable hints than just reporting the lines that the Checker failed on.

### 3.2 Structural Typing Lifting

Recall in Section 2 that TS<sup>-</sup> uses type lifting technique to transform some code components from ST (Structural Typing) into NT (Nominal Typing). However, in the implementation of Ts2WASM, structural type lifting is accomplished by several methods executed during various compilation passes. For simplicity, we represent the implementation of structural type lifting by using the Rewriter pass in which the type lifting methods are primarily executed.

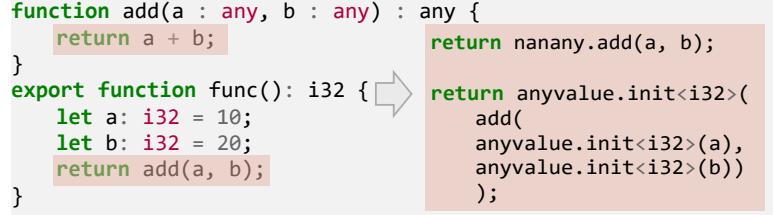
Roughly speaking, structural type lifting involves explicit rewriting and implicit inference. Implicit inference happens before explicit rewriting and endeavors to push the boundary between nominal and structural types toward the structural type side



as far as possible. The details of implicit inference are introduced in Section 4.3, and in this section, we focus on explicit rewriting.

## Primitive Types

Depending on the runtime type of a TypeScript variable, different strategies are used to rewrite the related AST node. Figure 3 shows an example of a simple `add` function in which the input and output arguments are all of `any` type, and a module-exported function `func` calls `add` with a primitive type `i32` for the input arguments. This scenario is frequently encountered when legacy JavaScript code is transitioned to TypeScript.



```

function add(a : any, b : any) : any {
  return a + b;
}
export function func(): i32 {
  let a: i32 = 10;
  let b: i32 = 20;
  return add(a, b);
}

function add(a : any, b : any) : any {
  return nanany.add(a, b);
}
export function func(): i32 {
  let a: i32 = 10;
  let b: i32 = 20;
  return anyvalue.init<i32>(
    add(
      anyvalue.init<i32>(a),
      anyvalue.init<i32>(b)
    )
  );
}

```

**Fig. 3:** Primitive Types: Simple Add function rewriting in Structural Typing Lifting

After the compilation passes, the original code inside the color box in Figure 3 is rewritten to the right-hand side counterpart. The original `any` is replaced with a runtime type `_any`, defined in our customized AssemblyScript compiler. To support the basic operations of `_any`, we introduce namespace `nanany` (NaN boxing any), which overrides the default operators with namespace methods, e.g., `nanany.add` in Figure 3. Type casts between built-in primitive types and runtime types are performed by inserting explicit calls to the casting functions, e.g., `anyvalue.init<T>`.

## Objects

Any keyword is also crucial for features associated with *Object Literals*. When an object is declared as `any`<sup>1</sup>, or an `any` parameter accepts an object as input, straight-forward type casting does not work in such cases.

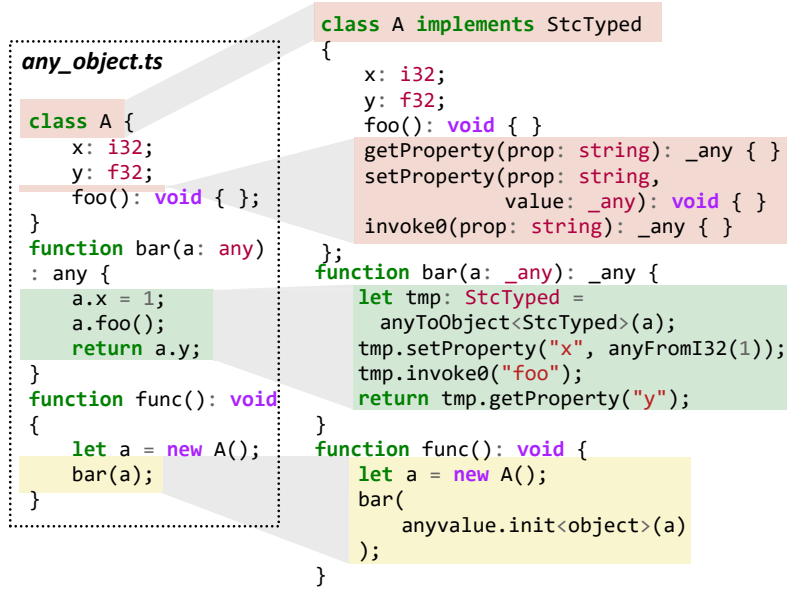
As the example in Figure 4 shows, the function `bar` attempts to access properties `y` and `x` and invoke method `foo` of parameter `a`. Because of TypeScript’s structural typing nature, any object with the same structure as class `A` can serve as a valid input for parameter `a`. Structural typing lifting involves converting an object into a nominal typing object, which requires designing a runtime type (interface) called `StcTyped`. `StcTyped` stands for *static* objects that maintain a static layout throughout their life-cycle. Additionally, it serves as a common interface for property access and method call of statically typed objects. The lifting process rewrites static object classes into

<sup>1</sup>Implicit `any` is forbidden in ‘strict’ TypeScript

classes implementing `StcTyped`, as is shown inside red boxes in Figure 4. The class definition now includes dedicated implementations of the `getProperty` and `setProperty` methods defined in `StcTyped`, which includes invalid property access checking.

Property access of parameter `a: any` in function `bar` are type checked to ensure runtime type safety. Initially, variable `a` is typecast to the object implementing `StcTyped`. Subsequently, the corresponding property accesses and method invokes are replaced with the aforementioned safe implementation.

Static objects are distinguished from *dynamic* ones with have changing dynamic memory layouts. The type-lifting strategy for dynamic objects is a runtime class `DynTyped` that implements `StcTyped` while maintaining a map to store object properties. Prior to accessing a property or invoking a method, the class searches the map using the given property name. Hence, property insertion/deletion can be implemented based on manipulating the map data structure.



**Fig. 4:** Objects: Method invoke and property access on a class object (*any* is used as an object)

## Closures

Prior research has extensively investigated the compilation of closures [25, 26]. Building on the work of others, and in order to design a strategy for lifting types on closures, it is necessary to determine and optimize the scope that each closure is allocated.

The Rewriter pass accepts metadata for closure rewriting collected in previous compilation passes and uses this metadata to explicitly construct the lexical scope of each closure. For example, in Figure 5, the return type of `func` is closure, and the

statement `let f1 = func(1)` assigns variable `f1` a closure value. Then a call is made to closure function `f1`.

In Figure 5, the nested scopes are represented by the boxes with varying greyscale colors, with higher greyscale colors used for inner scopes. Closure `f1` is defined in scope ③, invoked in scope ①, and dependent on variable `a` that lives in scope ②. When `f1` is assigned, only the pointer to the closure function is stored, while the input parameter `a` is lost because it is defined in scope ②, which is the outer scope of ③.

As painted in Figure 5, the code inside color boxes is transformed accordingly:

- The `return` statement in `func` is modified, and an object of class `Closure` is returned instead.
- `Context#func` object explicitly represents the lexical scope. The `Context` object is created with the closure and stored in the `Closure` object.
- `Closure` class is like a wrapper around the original closure function, which includes additional context information. Ultimately, a call to the function closure variable `f1` turns into invoking method `execute` of the `Closure` class.
- The class `Context` and `Closure` declarations are generated for each use case of a closure.

## 4 Compiler

In this section, we present the compiler implementation overview of Ts2WASM.

### 4.1 Compilation Passes

Generally, Ts2WASM consists of 5 compilation passes – *Resolver*, *Completer*, *Rewriter*, *Checker*, and *Emitter*, as shown in Figure 6. Before entering the compilation passes, Ts2WASM will use `tsc`, the TypeScript compiler, to perform pre-compilation

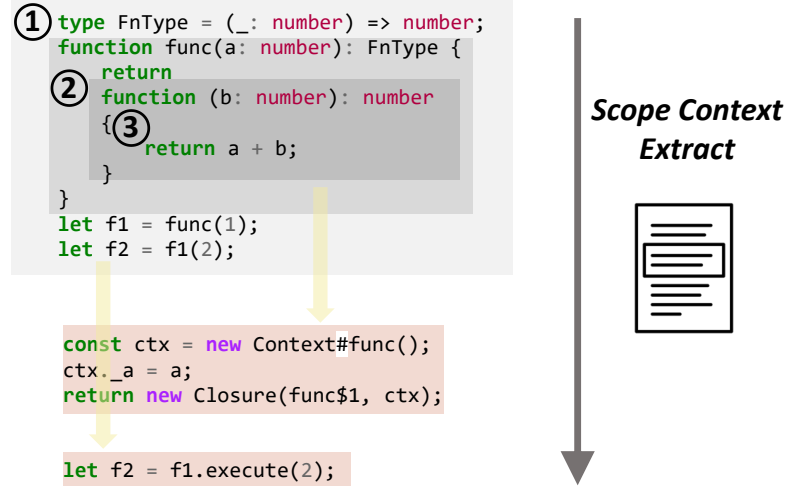
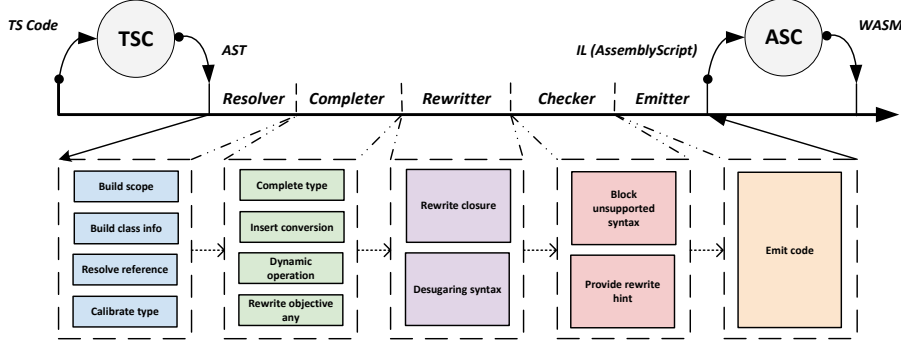


Fig. 5: Closures: Context extract and closure rewriting



**Fig. 6:** Ts2WASM Compiler: 5 Compilation Passes (ASC stands for AssemblyScript Compiler asc)

steps like lexical analysis, semantic analysis, type checking, and the AST (Abstract Syntax Tree) emitting. The ASTs will be later manipulated using TypeScript compiler API [27].

### ***Resolver***

During the *Resolver* pass, class information is extracted for subsequent compilation passes, incorporating the class hierarchies, members, and methods. Additionally, the type references are linked with the type declarations, the *Resolve Reference* block in Figure 6. For closures, Ts2WASM explicitly constructs a compile-time object **Scope** to identify the lexical scope. Additionally, to handle nested scopes, pointers to the inner and outer scope are also recorded. Resolver also infer types on variables declared as *any* based on the inference rules in Section 4.3. *Calibrate Type* process is performed to infer the types of variables and mark the inference-failed nodes as *any*.

### ***Completer***

The *Completer* pass generally addresses the issue of variable types, especially for *any*. Ts2WASM completes the type of each AST node based on the type inference results, and the originally untyped nodes will either have a concrete type or *any* type. Once some node was assigned *any* type, the *Completer* would insert type conversions to the relevant operations. For example, if

- An node with *any* type is inferred as a concrete type, then the corresponding conversion function from *any* to this concrete type is automatically inserted.
- Type inference failed on a node with *any* type, then a forced conversion from the concrete type to *any* would be inserted.

For objects with *any* type, the operations on this object are refactored to some built-in function calls, similar to reflection (See Figure 4 for example). We will explain the runtime implementation of *any* in Section 4.2.

## Rewriter

Ts2WASM will rewrite closures in this pass by leveraging the previously constructed **Scope** objects. As Ts2WASM aims to emit AssemblyScript code as intermediate products, which does not support some syntax sugar in TypeScript, desugaring techniques are used to rewrite the code into equivalent semantics.

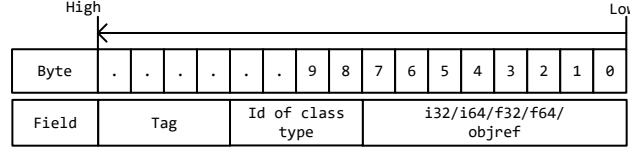
## Checker

For the dynamic features Ts2WASM currently does not support, *Checker* will capture these invalid uses of features, report check failures, and provide code rewrite hints accordingly.

## Emitter

The last pass will emit the refined AST to the AssemblyScript code. Note that in Figure 6, we put a customized AssemblyScript compiler after *Emitter*.

## 4.2 JavaScript *any*



**Fig. 7:** AnyValue: The memory layout

We extend **asc**, the AssemblyScript compiler, to support *any* with a built-in object, namely **AnyValue**. **AnyValue** wraps number types (**i32**, **i64**, **f32**, and **f64**) and reference types (**objref**), and the memory layout of **AnyValue** is shown in Figure 7. The 16-byte **AnyValue** object is allocated in the stack, bitwise-copied, and passed/returned in value. The first four-byte field, **Tag**, stores the type of each runtime object, and all the types represented by the **Tag** field are listed in Table 2. If the variable is a class object, the following four bytes are used to store the class ID of the object. Otherwise, these bytes are ignored. The actual value of *any* is represented by the remaining 8 bytes. Among these types, the 32-bit numbers, strings, and **objref**<sup>2</sup> only use the lower 4 bytes. **AnyValue** is implemented as a tagged union in AssemblyScript’s built-in definition and compiled using Binaryen IR tuple type. Tagged union is a special type to hold multiple values that does not exist in WASM.

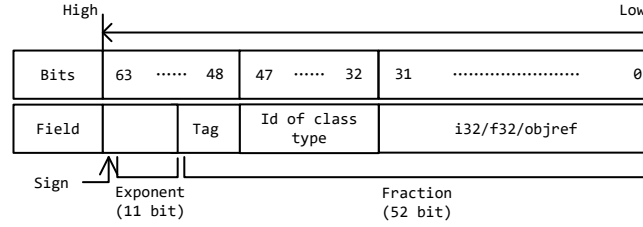
The compiler implementation of **AnyValue** is similar to the implementation of **JSValue** [28] in quickjs [29], a small embeddable JavaScript engine that supports the static compilation.

<sup>2</sup>Currently, WASM only supports 32-bit addressing

**Table 2: AnyValue: Runtime type represented by Tag field**

| Tag               | Type   |
|-------------------|--|
| TAG_NONE          | AnyValue with no concrete type, i.e., void, null and undefined |
| TAG_I32/I64       | i32/i64  |
| TAG_U32/U64       | u32/u64  |
| TAG_F32/F64       | f32/f64  |
| TAG_STRING        | builtin string   |
| TAG_OBJ           | GC-managed objects   |
| TAG_UNMANAGED_OBJ | Not GC-managed objects   |

NaN Boxing (or NaN Tagging) [30] plays an essential role in encoding JavaScript Numbers, which is the approach adopted by SpiderMonkey [31, 32] and JavaScript-Core [33]. Compared with the built-in object implementation, we also implement NaN Boxing in AssemblyScript with a built-in type **NanAny**. According to the IEEE-754 specification [30], there are  $2^{52} - 1$  different NaN values that can be encoded, but only sNaN (signal NaN) and qNaN (quiet NaN) are used in processors.

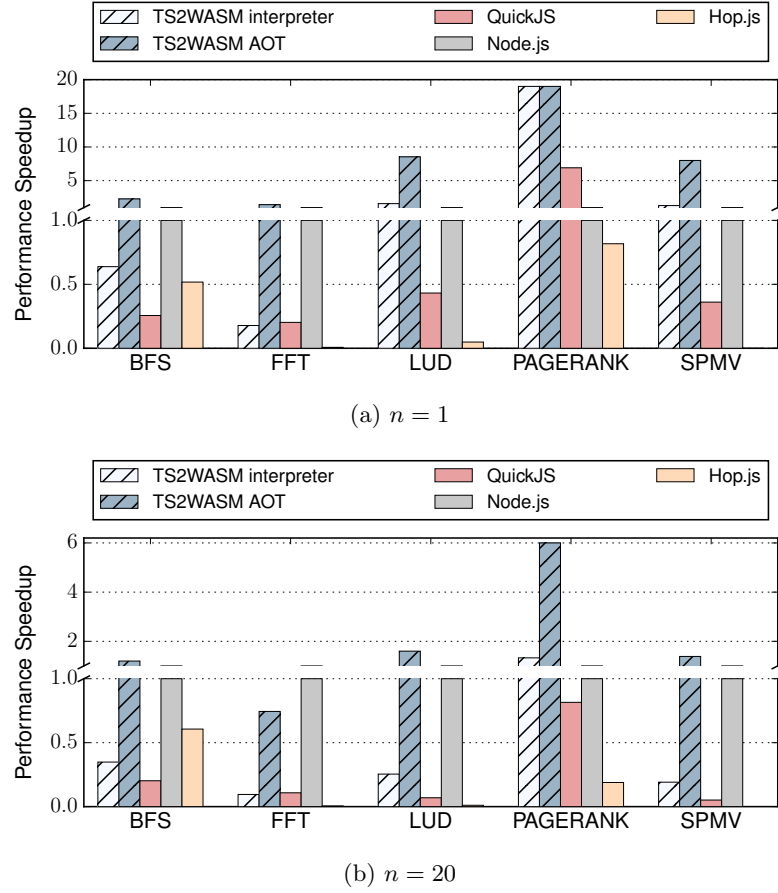


**Fig. 8: NaN-Boxing-Any (NanAny): The memory layout**

The memory layout of **NanAny** is shown in Figure 8. Compared with the memory layout of **AnyValue** in Figure 7, the data fields of **NanAny** are not strictly byte-aligned, so we use *Bits* instead of *Bytes* on the top as indices. One major difference between **AnyValue** and **NanAny** is that NaN boxing can encode *any* in a Binaryen primitive type (F64 or I64), which takes 64 bits of memory. At the same time, this cost doubles to 128 bits when we use **AnyValue**. The leftmost empty 12-bit field defaults to **0x7ff**, representing a NaN value in IEEE-754 spec, while other values for this field are treated as the sign bit (1 bit) + exponent part (11 bits) of the double floating point number representation.

Meanwhile, the Tag field is compressed into 4 bits, compared with the original 4 bytes in **AnyValue**. Considering only seven values are used in Table 2, integers from 0 to 16 are sufficient to encode the currently supported tags, including sNaN, qNaN, and  $\infty$ . The remaining bits encode the class ID and the data value. Since the runtime class ID is a 32-bit unsigned integer, the workaround is keeping the lower 16 bits of the class IDs, as they are incremented from 0.

### 4.3 Type Inference



**Fig. 9:** AS-Benchmarks Performance: Speedup compared with Node.js (JavaScript). The Ts2WASM result stands for the best performance among interpreters / AOT compilers. (Wasmtime JIT mode is taken into the account of AOT compilers)

Although TypeScript does not recommend using *any*, it is still commonly used in some cases, such as when incorporating third-party JavaScript libraries or for quick bug fixes when avoiding TypeScript’s type checking is preferable. Ts2WASM pursues minimizing the usage of *any* as much as possible. Given that in Ts2WASM, each use of *any* incurs the cost of runtime type casts. We use the following type inference rules to eliminate unnecessary *any*s.

*Rule 1:* Types of operands and results derived on the basis of operator rules.

1. The operands and results of arithmetic operators can only be **Number**, except for **+**, which also accepts **String**.
2. The results of relational operators (**>**, **<**, **==**, etc.) must be **Boolean**.

*Rule 2:* Infer the types of operands and results based on the argument types and the return types of built-in functions.

1. The argument of `console.log` function must be **String** type.
2. Most built-in functions in **Math** library accept/return only **Number**.
3. Some type cast functions have fixed typed arguments and return values, e.g., `parseInt`, `parseFloat`, `toString`, etc.

*Rule 3:* Assignment operations should have consistent types of expressions on the right side and on the left side.

1. Infer types of expressions on the left-hand side based on the type of expression on the right-hand side and vice versa.
2. The input arguments must be type-consistent with the function parameters.
3. The type of expression **Return** must be consistent with the return type of the function.
4. The preceding rules are also applied to literals and the **new** operator.

*Rule 4:* For the dot (**.**) operator, the left-hand side expression must be an object. For example, an expression `A.B.C.D = X`, variable `A`, `A.B`, and `A.B.C` should be an object.

## 5 Evaluation

In this section, we evaluate Ts2WASM in the following aspects:

1. **Feature Support:** As mentioned in Section 2, the static compilation of TypeScript is non-trivial. We aim to support the most widely used features for TypeScript developers such that they can write and deploy AOT-compiled programs efficiently. We use the tests from the official repository of TypeScript and AssemblyScript to check the feature support status Ts2WASM. Besides, we also design unit tests for Ts2WASM to verify the correctness of feature support.
2. **AOT Performance:** The performance of TypeScript relies on JavaScript compiler toolchain implementations. While the execution speed of JavaScript is slower than some AOT-compiled languages (C/C++) [34], the AOT performance of JavaScript [10] often falls behind the optimized JIT runtimes. For those concerned about the performance of statically compiled code, we choose AS-Benchmarks [35], a compute-intensive benchmark suite provided by AssemblyScript, to compare the performance of AOT-compiled and JIT-compiled programs.
3. **Memory Usage:** Generally, AOT-compiled binaries are more memory-efficient than programs running in JIT runtimes. The preceding statement also holds for embedded systems with constrained storage, where AOT is the preferred option for memory-saving purposes. To demonstrate the memory efficiency of Ts2WASM



compared with existing works, we tested the maximum allocated memory size in benchmark programs for various runtimes and compilers.

4. **Runtime Implementation Overheads:** The implementation of `AnyValue` relies on the assistance of built-in runtime functions, which can introduce some performance overhead. The third experiment quantifies this overhead using four computation-intensive benchmarks with different input parameters. The results can provide users with hints to adjust the use frequency of *any* to mitigate the performance slowdown.

**Table 3:** Ts2WASM Running Tests from Ts2WASM, AssemblyScript, and TypeScript, respectively. *Skipped* are the tests that currently unsupported by Ts2WASM checker

|                | <i>Total</i> | <i>Passed</i> | <i>Failed</i> | <i>Skipped</i> |
|----------------|--------------|---------------|---------------|----------------|
| Ts2WASM        | 77           | 77            | 0             | 0              |
| AssemblyScript | 106          | 106           | 0             | 0              |
| TypeScript     | 5409         | 1160          | 1396          | 2853           |

## 5.1 Feature Tests

As shown in Table 3, Ts2WASM passes all the tests from AssemblyScript and Ts2WASM, covering about 47% of the tests from TypeScript. Among the covered tests from TypeScript, 1160 test cases (45%) passed cases, i.e., these 1160 test cases can be AOT compiled to WASM. In AssemblyScript, only 306 of the 1160 tests are passed, indicating that Ts2WASM provides 3.8x as many features.

The *Failed* cases stand for the cases that are rule-outed by the Checker, such as `abstract` keywords, accessors, `BigInt`, and some decorators (e.g., `accessor`, `property`, and `parameter`), which are mentioned in Section 3. As Ts2WASM includes a superset of AssemblyScript, the tests mentioned above also fail during `asc` compilation. Many *Failed* tests use JavaScript-compatible features and syntactic sugar, which are not the primary focus of Ts2WASM implementation.

The remaining *Skipped* cases are the features the Ts2WASM checker can not detect. Hence, they might cause unexpected behaviors (such as runtime errors) when used with Ts2WASM. In theory, it is feasible to classify the *Skipped* test cases, which can be achieved by maintaining a logically correct table to eliminate all language features that are not supported by AOT compilation. However, creating such a filtering table would require significant manual effort due to the complexity of TypeScript and the complex cases that require considering the types of multiple AST nodes. Therefore, we leave the implementation of these test cases for future work.

To validate that our modification to AssemblyScript does not incur errors, we introduce the AssemblyScript tests, all of which passed. The outcome indicates that AssemblyScript programs can run without modification with Ts2WASM.

## 5.2 AOT Performance

### Methodology

We run the performance tests on a Linux server, which has a 2.6 GHz Intel Xeon Gold 6240 32-core CPU and 190 GiB memory with hyper-threading disabled. For compilers, `-O3` flag is set for the best performance. The benchmarks are selected from the AS-Benchmarks [35], which contain five individual tests: **BFS** (Breadth First Search), **FFT** (Fast Fourier Transform), **LUD** (Lower Upper Decomposition), **PageRank**, and **SPMV** (Sparse Matrix Vector Multiplication).

We use nine different language runtimes/compilers for testing:

- **WAMR (Wasm Micro Runtime)** interpreter/AOT: WASM runtime written in C, optimized for both small source code and runtime size.
- **Wasmtime** JIT/AOT: WASM runtime written in Rust that has rich support for the WASM features.
- **Wasm3**: A fast WASM interpreter that supports a wide range of embedded system architectures.
- **Node.js**: JavaScript framework powered by V8 JavaScript engine.
- **QuickJS**: An embeddable JavaScript interpreter and runtime [29] that can compile JavaScript programs into small native binaries.
- **Hop.js**: A JavaScript AOT compiler [9, 10].

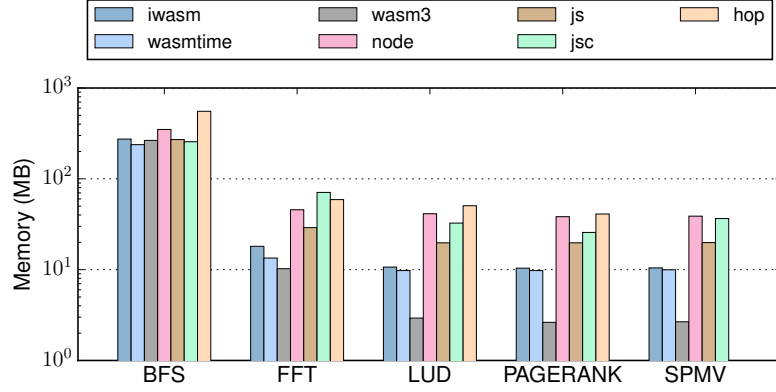
As the performance of JIT compilers is optimized through repeated executions, we use two experiment settings to compare the performance between Ts2WASM compiled WASM binaries and JavaScript on other JavaScript runtimes. The performance of JavaScript can, to a large extent, represent the performance of JavaScript since TypeScript is first compiled into JavaScript and then executed. We utilize the parameter  $n = 1$  to indicate the number of iterations for each test run. Specifically, when  $n$  is set to 1, we analyze the cold-start performance of JavaScript JIT compilers. When  $n$  is set to 20, we evaluate their warm-up performance.

### Results

The experiment results are shown in Figure 9. We chose the best performance data from all WASM runtimes to represent the performance of Ts2WASM and distinguished the execution policy adopted with *interpreter/AOT* suffix. The vertical axis stands for the related performance speedup of each tested compiler, normalized to the performance of Node.js. Specifically, this is computed by

$$R_{X-speedup} = \frac{T_{node}}{T_X} \quad (1)$$

When  $n = 1$ , as Figure 9 (a) shows, the platform with the best performance is Ts2WASM AOT, which has a speedup ranging from 1.4x to 19x. Ts2WASM interpreters also outperform Node.js in LUD, PageRank, and SPMV tests and even approximate the performance of the Ts2WASM AOT compiler in the PageRank test. BFS and FFT tests involve the operations on high-dimensional data, such as matrix



**Fig. 10:** Maximum Memory Footprints: compare the memory overhead of running TS2WASM AOT binaries with JIT-compiled JavaScript programs

convolution and graph walking, for which WASM is not optimized. Despite WASM interpreters being slower than Node.js in some cases, they perform better than Hop.js, exhibiting up to an order of magnitude speedups in LUD and PageRank tests.

The  $n = 20$  case in Figure 9 (b) gives us a different result, where the speedup becomes less pronounced. Moreover, for JavaScript programs, the speed of Node.js is dominant after the warm-up phase. However, Ts2WASM AOT compilation still outperforms Node.js in all tests except FFT, with an average 6x speedup. In the worst case of AOT compilation, Ts2WASM still reaches 74% of Node.js’s execution speed. Moreover, the performance of Hop.js does not improve but shows heavier performance degradation in LUD and PageRank tests.

There are some exceptional cases, for example, that Hop.js failed in FFT and SPMV tests. In these two tests, runtime errors are thrown, while the compilation from JavaScript to the native code of Hopc works fine. Additionally, the performance of LUD in Hop.js is terrible, with no errors generated during runtime. Another point not mentioned in the figure is that when running WASM with Node.js, the speedup in the BFS test is minor (1.78x), and the speedups in the rest tests are not significant, from which we can infer that standalone WASM runtimes are better than V8 in a native environment.

## 5.3 Memory Size

### Methodology

A process’s memory usage is hard to precisely defined, and the most frequently referred metrics are heap/stack usage, virtual memory size (VSZ), and resident set size (RSS). VSZ should be larger than RSS since the mapped pages can not exceed the total virtual memory size given to a process. However, the memory test results presented in Hop.js [10] demonstrate that the RSS is greater than the VSZ, which is unconvincing.

We adopt the Linux *time* [36] command to measure benchmark programs’ memory usage (RSS). Besides Node.js (V8), we have incorporated two other significant

JavaScript engines: js and jsc. Jsc and js are built locally in the test environment, using WebKit 2.36 and the latest release of Mercurial repository (SpiderMonkey sources).

## Results

The maximum memory footprints are shown in Figure 10, where each color stands for a different runtime or compiler. In the BFS test, the difference between maximum memory footprints for JavaScript runtimes and WASM is small, and the JavaScript program’s memory footprint is about 10% larger. After comparing the minimum memory footprint data of JavaScript runtimes with that of WASM, we found that in the FFT test, WASM saves 65% of memory. Additionally, in the rest tests, WASM can save more than 85% of memory usage.

## 5.4 Runtime Overhead

### Methodology

We use three tests from AS-Benchmark: FFT, LUD, and PageRank, along with a matrix multiplication test, GEMM, to evaluate the runtime overhead of **AnyValue** implementation. The horizontal axis represents the input parameter set for each test, increasing from *A* to *E*. And the vertical axis stands for the performance loss compared with the baseline programs that use **f64** variables instead of **AnyValue**, which is computed by

$$Loss_X = \frac{T_{X(any)}}{T_{X(f64)}} - 1 \quad (2)$$

### Results

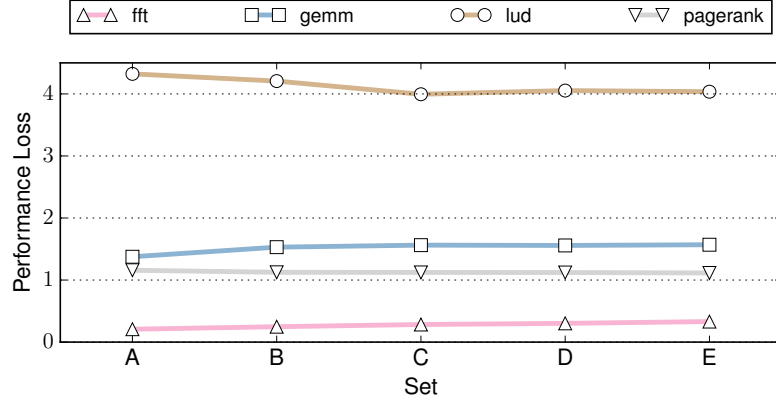
As is shown in Figure 11, the slowdowns of *any* tests are not increasing when the workload increases, meaning a more frequent use of *any*.

On average, the performance loss is 1.7x for all benchmark tests. For FFT, the average performance loss is 0.3x, the minimum among the tests. The worst case of performance loss happens in the LUD test, having an average value of 4.0x. Additionally, the performance loss for GEMM and PageRank falls between 1.0x to 1.5x. While the performance loss is moderate in some cases (0.3x), the abuse of *any* (PageRank) would cause a heavy performance downgrade.

## 6 Related Work

### 6.1 Structural Typing vs. Nominal Typing

There has long been a debate about the superiority of dynamic typing over static typing or vice versa. [37] conducts a large-scale survey to study the effect of language features. The results from their work show that strong typing is slightly better than weak typing, and for functional languages, static typing is better than dynamic typing in code quality. Moreover, an empirical study [38] shows that static typing has a positive impact on the maintenance of software. However, the dynamic characteristic of structural typing languages allow writing succinct code flexibly and efficiently.



**Fig. 11: AnyValue Overhead**

Gradual typing [15] offers a way to emerge the two type systems: structural typing and nominal typing into a language, and TypeScript is a famous one of gradual typing languages. Even though TypeScript is intentionally unsound [17], the type system in TypeScript brings many positive outcomes. The evidence can be found in a study in 2017 [39] that runs a series of quantifying tests in TypeScript and FLOW. The test results in this paper show that adding type annotations to JavaScript can help avoid 15% of the reported bugs.

## 6.2 JavaScript / TypeScript in Embedded Systems

The works with similar purpose of this paper are Espruino and Static TypeScript, which are both designed for embedded systems. Espruino [14] offers a comprehensive hardware and software solution that enables JavaScript-like embedded software development. Meanwhile, Static TypeScript [7] is a TypeScript-like language for course teaching with embedded devices [40]. However, these approaches are limited to some specific embedded hardware and preserves few structural typing language features.

## 6.3 AOT Compilation of Structural / Gradual Typing Languages

Researchers have shown that AOT compilation can bring significant performance improvements [41, 42]. While for JavaScript, the performance of AOT compilers is not usually comparable to JIT compilers (see in 6.2), some optimization techniques follow an AOT-like nature.

To support AOT compilation, some existing works proposed to extend or modify TypeScript, which includes [7, 19, 20]. An alternative approach is to generate statically compiled targets with some JavaScript AOT compilation workarounds, typically, building a language subset of JavaScript, as thoroughly investigated in many research papers during the past decade [9, 11–13].

### 6.3.1 ~JS Related Works

The research on the AOT compilation of JavaScript started long ago before we proposed this work, and continues till recently. As JavaScript is a dynamically typed language, the AOT compilation would suffer from the runtime path penalty for determining the types of variables and expressions. An early dynamic language [43], SELF, invoked the succeeding type inference works (mainly for JavaScript). For instance, TAJIS [44] proposes a whole program analysis framework for inferring the sound type information for JavaScript programs. After that, researchers from Samsung Research proposes SJS [11], a static type system for a subset of JavaScript, which supports high-level features such as prototype inheritance, structural subtyping, and closures. SJS also presents a proof-of-concept AOT compiler implementation, compiling JavaScript to C, and then to machine code. Based on the type system of SJS, researchers proposed a rich subset of JavaScript for static compilation in [12], where they further extend SJS to a more comprehensive type inference framework. Hop.js [9, 10] is another AOT compiler and language subset of JavaScript, while Hopc compiles JavaScript to Scheme and uses another compiler, Bigloo [45], to compile Scheme. The authors of Hop.js further investigate the performance of JavaScript AOT compilation (Hop.js) in [10] and claim that the performance (execution time) of Hopc is mostly within  $2\times$  compared to V8. They also introduce the detailed design of Hopc in [9, 46], which gives valuable insights into the future AOT compiler design of dynamic languages. Type inference techniques play an important role in preceding JavaScript AOT compilation frameworks, and their effects and accuracy are thoroughly studied in [47].

### 6.3.2 ~TS Related Works

Another line of research is introducing type annotations to JavaScript to support efficient type checking, and the mainstream is Microsoft TypeScript [16]. Besides, Facebook also proposes a fast type checking system for JavaScript called FLOW. [13]. A common trait of these languages is that they use their type system for type checking instead of compilation. And due to pragmatic reasons, the type system for TypeScript is unsound, which means building an AOT compiler directly on top of TypeScript is infeasible. Hence, researchers devote their efforts to proposing some variants of TypeScript language and the corresponding AOT compiler implementations. For example, Static TypeScript [7] (STS) is a subset of TypeScript developed by Microsoft, with an easy-to-learn syntax and compact code size of static compilation. The distinct features of STS are utilized to promote in-class programming for teenage students. StrongScript [20] is a superset of TypeScript which extends TypeScript with syntax ! to denote concrete types and provides correctness guarantees offered by the language runtime for concretely typed code. Safe TypeScript [19], also a subset of TypeScript, adds a “Safe” compilation mode for TypeScript, that enforces additional type checks to confirm type soundness. A most recent paper [48] proposes their AOT compiler framework STSC (Static TypeScript Compiler), which is implemented based on `tsc` and V8.

## 6.4 WebAssembly

First proposed in 2015 [22], and released MVP in 2017, WebAssembly (WASM) [49] is a binary format with compact code size and near-native execution speed. Despite initially being designed for web applications, WASM is a light-weight sandbox independent of the source language, platform, and architecture, which is suitable for embedded systems applications [50–56]. The release and standardization of WASI (WebAssembly System Interface) [57, 58] provides official community support for native WASM applications. With the evolution of WASI, more embedded system applications will be able to migrate to WASM. Engaged with the WASM, the applications developed in TS<sup>-</sup> can be portable among various embedded devices.

## 7 Conclusion

In this paper, we present TS<sup>-</sup>, a design methodology for embedded systems programming language. A prototype implementation of TS<sup>-</sup> is Ts2WASM, a flexible compiler framework for the static compilation of TypeScript, that provides a checker tool to split the code into dynamic and static parts. We show that Ts2WASM generated targets achieve close-to-JavaScript and even better performance on compute-intensive benchmarks, with optimized memory usage.

## References

- [1] Foundation, E.: 2022 IoT & Edge Developer Survey Report (2022). <https://accounts.eclipse.org/documents>
- [2] Ray, B., Posnett, D., Devanbu, P.T., Filkov, V.: A large-scale study of programming languages and code quality in github. *Commun. ACM* **60**(10), 91–100 (2017) <https://doi.org/10.1145/3126905>
- [3] Berger, E.D., Hollenbeck, C., Maj, P., Vitek, O., Vitek, J.: On the impact of programming languages on code quality: A reproduction study. *ACM Trans. Program. Lang. Syst.* **41**(4), 21–12124 (2019) <https://doi.org/10.1145/3340571>
- [4] Vaarala, S.: DukTape (2023). <https://duktape.org/>
- [5] George, D.: MicroPython (2023). <http://www.micropython.org/>
- [6] Bierman, G.M., Abadi, M., Torgersen, M.: Understanding typescript. In: Jones, R.E. (ed.) ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. *Proceedings. Lecture Notes in Computer Science*, vol. 8586, pp. 257–281. Springer, ??? (2014). [https://doi.org/10.1007/978-3-662-44202-9\\_11](https://doi.org/10.1007/978-3-662-44202-9_11) . [https://doi.org/10.1007/978-3-662-44202-9\\_11](https://doi.org/10.1007/978-3-662-44202-9_11)
- [7] Ball, T., Halleux, P., Moskal, M.: Static typescript: an implementation of a static compiler for the typescript language. In: Hosking, A.L., Finocchi, I. (eds.) *Proceedings of the 16th ACM SIGPLAN International Conference on Managed*

- Programming Languages and Runtimes, MPLR 2019, Athens, Greece, October 21-22, 2019, pp. 105–116. ACM, ??? (2019). <https://doi.org/10.1145/3357390.3361032> . <https://doi.org/10.1145/3357390.3361032>
- [8] Microsoft: DeviceScript: TypeScript for Tiny IoT Devices (2023). <https://microsoft.github.io/devicescript/>
  - [9] Serrano, M.: Javascript AOT compilation. In: Felgentreff, T. (ed.) Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2018, Boston, MA, USA, November 6, 2018, pp. 50–63. ACM, ??? (2018). <https://doi.org/10.1145/3276945.3276950> . <https://doi.org/10.1145/3276945.3276950>
  - [10] Serrano, M.: Of javascript AOT compilation performance. Proc. ACM Program. Lang. **5**(ICFP), 1–30 (2021) <https://doi.org/10.1145/3473575>
  - [11] Choi, W., Chandra, S., Necula, G.C., Sen, K.: SJS: A type system for javascript with fixed object layout. In: Blazy, S., Jensen, T.P. (eds.) Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9291, pp. 181–198. Springer, ??? (2015). [https://doi.org/10.1007/978-3-662-48288-9\\_11](https://doi.org/10.1007/978-3-662-48288-9_11) . [https://doi.org/10.1007/978-3-662-48288-9\\_11](https://doi.org/10.1007/978-3-662-48288-9_11)
  - [12] Chandra, S., Gordon, C.S., Jeannin, J., Schlesinger, C., Sridharan, M., Tip, F., Choi, Y.: Type inference for static compilation of javascript. In: Visser, E., Smaragdakis, Y. (eds.) Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, Part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016, pp. 410–429. ACM, ??? (2016). <https://doi.org/10.1145/2983990.2984017> . <https://doi.org/10.1145/2983990.2984017>
  - [13] Chaudhuri, A., Vekris, P., Goldman, S., Roch, M., Levi, G.: Fast and precise type checking for javascript. Proc. ACM Program. Lang. **1**(OOPSLA), 48–14830 (2017) <https://doi.org/10.1145/3133872>
  - [14] Espruino: Espruino - JavaScript for Microcontrollers. <https://www.espruino.com/> Accessed 2023-03-20
  - [15] Siek, J.G., Taha, W.: Gradual typing for objects. In: Ernst, E. (ed.) ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4609, pp. 2–27. Springer, ??? (2007). [https://doi.org/10.1007/978-3-540-73589-2\\_2](https://doi.org/10.1007/978-3-540-73589-2_2) . [https://doi.org/10.1007/978-3-540-73589-2\\_2](https://doi.org/10.1007/978-3-540-73589-2_2)
  - [16] Microsoft: TypeScript Specification (2014). <https://www.typescriptlang.org/>
  - [17] Allende, E., Callaú, O., Fabry, J., Tanter, E., Denker, M.: Gradual typing for smalltalk. Sci. Comput. Program. **96**(P1), 52–69 (2014) <https://doi.org/10.1016/>



- [18] Bierman, G.M., Meijer, E., Torgersen, M.: Adding Dynamic Types to C#. In: ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings, pp. 76–100 (2010). [https://doi.org/10.1007/978-3-642-14107-2\\_5](https://doi.org/10.1007/978-3-642-14107-2_5) . [https://doi.org/10.1007/978-3-642-14107-2\\_5](https://doi.org/10.1007/978-3-642-14107-2_5)
- [19] Rastogi, A., Swamy, N., Fournet, C., Bierman, G.M., Vekris, P.: Safe & efficient gradual typing for typescript. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015, pp. 167–180. ACM, ??? (2015). <https://doi.org/10.1145/2676726.2676971> . <https://doi.org/10.1145/2676726.2676971>
- [20] Richards, G., Nardelli, F.Z., Vitek, J.: Concrete types for typescript. In: Boyland, J.T. (ed.) 29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic. LIPIcs, vol. 37, pp. 76–100. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, ??? (2015). <https://doi.org/10.4230/LIPIcs.ECOOP.2015.76> . <https://doi.org/10.4230/LIPIcs.ECOOP.2015.76>
- [21] Takikawa, A., Feltey, D., Greenman, B., New, M.S., Vitek, J., Felleisen, M.: Is sound gradual typing dead? In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, pp. 456–468 (2016). <https://doi.org/10.1145/2837614.2837630> . <https://doi.org/10.1145/2837614.2837630>
- [22] Bastien, J.F.: WebAssembly – Going public launch bug (2015). <https://github.com/WebAssembly/design/issues/150>
- [23] Alliance, B.: WebAssembly Micro Runtime. Bytecode Alliance. original-date: 2019-05-02T21:32:09Z (2023). <https://github.com/bytecodealliance/wasm-micro-runtime> Accessed 2023-03-22
- [24] Wasm3: wasm3/wasm3: A fast WebAssembly interpreter, and the most universal WASM runtime. <https://github.com/wasm3/wasm3> Accessed 2023-03-22
- [25] Kranz, D.A., Kelsey, R., Rees, J., Hudak, P., Philbin, J., Adams, N.: Orbit: an optimizing compiler for scheme (with retrospective). In: 20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection, pp. 175–191 (1986). <https://doi.org/10.1145/989393.989414> . <https://doi.org/10.1145/989393.989414>
- [26] Shao, Z., Appel, A.W.: Space-Efficient Closure Representations. In: Proceedings of the 1994 ACM Conference on LISP and Functional Programming, Orlando, Florida, USA, 27-29 June 1994., pp. 150–161 (1994). <https://doi.org/10.1145/182409.156783> . <https://doi.org/10.1145/182409.156783>

- [27] Microsoft: Using the compiler API · Microsoft/typescript wiki (2021). <https://github.com/Microsoft/TypeScript/wiki/Using-the-Compiler-API>
- [28] Bellard, F.: QuickJS Documentation - JSValue (2021). <https://bellard.org/quickjs/quickjs.html#JSValue-1>
- [29] Bellard, F.: QuickJS Javascript Engine (2021). <https://bellard.org/quickjs/>
- [30] Wikipedia, T.F.E.: Double-precision floating-point format (2022). [https://en.wikipedia.org/w/index.php?title=Double-precision\\_floating-point\\_format&oldid=1104943899](https://en.wikipedia.org/w/index.php?title=Double-precision_floating-point_format&oldid=1104943899)
- [31] Mozilla: SpiderMonkey - Mozilla’s JavaScript and WebAssembly Engine, (2022). <https://spidermonkey.dev/>
- [32] MDN: SpiderMonkey Internals (2019). <https://firefox-source-docs.mozilla.org/js/index.html>
- [33] Apple: WebKit: A fast, open source web browser engine. (2018). <https://webkit.org/>
- [34] Cheng, L., Ilbeyi, B., Bolz-Tereick, C.F., Batten, C.: Type freezing: exploiting attribute type monomorphism in tracing JIT compilers. In: CGO ’20: 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, February, 2020, pp. 16–29. ACM, ??? (2020). <https://doi.org/10.1145/3368826.3377907> . <https://doi.org/10.1145/3368826.3377907>
- [35] nischayv: Github - AS-Benchmarks (2020). <https://github.com/nischayv/as-benchmarks>
- [36] page, L.: time(1) - Linux manual page. <https://man7.org/linux/man-pages/man1/time.1.html> Accessed 2023-03-24
- [37] Ray, B., Posnett, D., Filkov, V., Devanbu, P.T.: A large scale study of programming languages and code quality in github. In: Cheung, S., Orso, A., Storey, M.D. (eds.) Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014, pp. 155–165. ACM, ??? (2014). <https://doi.org/10.1145/2635868.2635922> . <https://doi.org/10.1145/2635868.2635922>
- [38] Hanenberg, S., Kleinschmager, S., Robbes, R., Tanter, É., Stefik, A.: An empirical study on the impact of static typing on software maintainability. *Empir. Softw. Eng.* **19**(5), 1335–1382 (2014) <https://doi.org/10.1007/s10664-013-9289-1>
- [39] Gao, Z., Bird, C., Barr, E.T.: To type or not to type: quantifying detectable bugs in javascript. In: Uchitel, S., Orso, A., Robillard, M.P. (eds.) Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos

- Aires, Argentina, May 20-28, 2017, pp. 758–769. IEEE / ACM, ??? (2017). <https://doi.org/10.1109/ICSE.2017.75> . <https://doi.org/10.1109/ICSE.2017.75>
- [40] Devine, J., Finney, J., Halleux, P.d., Moskal, M., Ball, T., Hodges, S.: Make-Code and CODAL: intuitive and efficient embedded systems programming for education. In: Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2018, Philadelphia, PA, USA, June 19-20, 2018, pp. 19–30 (2018). <https://doi.org/10.1145/3211332.3211335> . <https://doi.org/10.1145/3211332.3211335>
  - [41] Wimmer, C., Stancu, C., Hofer, P., Jovanovic, V., Wögerer, P., Kessler, P.B., Pliss, O., Würthinger, T.: Initialize once, start fast: application initialization at build time. Proc. ACM Program. Lang. **3**(OOPSLA), 184–118429 (2019) <https://doi.org/10.1145/3360610>
  - [42] Wade, A.W., Kulkarni, P.A., Jantz, M.R.: AOT vs. JIT: impact of profile data on code quality. In: Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2017, Barcelona, Spain, June 21-22, 2017, pp. 1–10 (2017). <https://doi.org/10.1145/3078633.3081037> . <https://doi.org/10.1145/3078633.3081037>
  - [43] Agesen, O., Palsberg, J., Schwartzbach, M.I.: Type inference of SELF. In: Nierstrasz, O. (ed.) ECOOP’93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26-30, 1993, Proceedings. Lecture Notes in Computer Science, vol. 707, pp. 247–267. Springer, ??? (1993). [https://doi.org/10.1007/3-540-47910-4\\_14](https://doi.org/10.1007/3-540-47910-4_14) . [https://doi.org/10.1007/3-540-47910-4\\_14](https://doi.org/10.1007/3-540-47910-4_14)
  - [44] Jensen, S.H., Möller, A., Thiemann, P.: Type analysis for javascript. In: Palsberg, J., Su, Z. (eds.) Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5673, pp. 238–255. Springer, ??? (2009). [https://doi.org/10.1007/978-3-642-03237-0\\_17](https://doi.org/10.1007/978-3-642-03237-0_17) . [https://doi.org/10.1007/978-3-642-03237-0\\_17](https://doi.org/10.1007/978-3-642-03237-0_17)
  - [45] Serrano, M., Weis, P.: Bigloo: A portable and optimizing compiler for strict functional languages. In: Mycroft, A. (ed.) Static Analysis, Second International Symposium, SAS’95, Glasgow, UK, September 25-27, 1995, Proceedings. Lecture Notes in Computer Science, vol. 983, pp. 366–381. Springer, ??? (1995). [https://doi.org/10.1007/3-540-60360-3\\_50](https://doi.org/10.1007/3-540-60360-3_50) . [https://doi.org/10.1007/3-540-60360-3\\_50](https://doi.org/10.1007/3-540-60360-3_50)
  - [46] Serrano, M., Feeley, M.: Property caches revisited. In: Amaral, J.N., Kulkarni, M. (eds.) Proceedings of the 28th International Conference on Compiler Construction, CC 2019, Washington, DC, USA, February 16-17, 2019, pp. 99–110. ACM, ??? (2019). <https://doi.org/10.1145/3302516.3307344> . <https://doi.org/10.1145/3302516.3307344>

- [47] Saifullah, C.M.K., Asaduzzaman, M., Roy, C.K.: Exploring type inference techniques of dynamically typed languages. In: Kontogiannis, K., Khomh, F., Chatzigeorgiou, A., Fokaefs, M., Zhou, M. (eds.) 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020, pp. 70–80. IEEE, ??? (2020). <https://doi.org/10.1109/SANER48275.2020.9054814> . <https://doi.org/10.1109/SANER48275.2020.9054814>
- [48] Wu, Z., Sun, Z., Gong, K., Chen, L., Liao, B., Jin, Y.: Hidden inheritance: an inline caching design for typescript performance. *Proc. ACM Program. Lang.* 4(OOPSLA), 174–117429 (2020) <https://doi.org/10.1145/3428242>
- [49] Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.F.: Bringing the web up to speed with webassembly. In: Cohen, A., Vechev, M.T. (eds.) Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017, pp. 185–200. ACM, ??? (2017). <https://doi.org/10.1145/3062341.3062363> . <https://doi.org/10.1145/3062341.3062363>
- [50] Wen, E., Weber, G.: Wasmachine: Bring iot up to speed with A webassembly OS. In: 2020 IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2020, Austin, TX, USA, March 23-27, 2020, pp. 1–4. IEEE, ??? (2020). <https://doi.org/10.1109/PerComWorkshops48775.2020.9156135> . <https://doi.org/10.1109/PerComWorkshops48775.2020.9156135>
- [51] Zheng, S., Wang, H., Wu, L., Huang, G., Liu, X.: VM matters: A comparison of WASM vms and evms in the performance of blockchain smart contracts. *CoRR abs/2012.01032* (2020) [2012.01032](https://arxiv.org/abs/2012.01032)
- [52] Ménétrey, J., Pasin, M., Felber, P., Schiavoni, V.: Twine: An embedded trusted runtime for webassembly. In: 37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021, pp. 205–216. IEEE, ??? (2021). <https://doi.org/10.1109/ICDE51399.2021.00025> . <https://doi.org/10.1109/ICDE51399.2021.00025>
- [53] Narayan, S., Garfinkel, T., Lerner, S., Shacham, H., Stefan, D.: Gobi: Webassembly as a practical path to library sandboxing. *CoRR abs/1912.02285* (2019) [1912.02285](https://arxiv.org/abs/1912.02285)
- [54] Shillaker, S., Pietzuch, P.R.: Faasm: Lightweight isolation for efficient stateful serverless computing. In: Gavrilovska, A., Zadok, E. (eds.) 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020, pp. 419–433. USENIX Association, ??? (2020). <https://www.usenix.org/conference/atc20/presentation/shillaker>

- [55] McCallum, T.: AI on a cloud native WebAssembly runtime (WasmEdge) — Part I (2021). <https://medium.com/wasm/ai-on-a-cloud-native-webassembly-runtime-wasmedge-part-i-3bf3714a64ea>
- [56] Wen, E., Weber, G., Nanayakkara, S.: WasmAndroid: a cross-platform runtime for native programming languages on Android (WIP paper). In: LCTES '21: 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, Virtual Event, Canada, 22 June, 2021, pp. 80–84 (2021). <https://doi.org/10.1145/3461648.3463849> . <https://doi.org/10.1145/3461648.3463849>
- [57] bytecodealliance: WASI: The WebAssembly System Interface (2020). <https://wasi.dev/>
- [58] Clark, L.: Standardizing Wasi: A system interface to run webassembly outside the web – mozilla hacks - the web developer blog. Mozilla Hacks (2019). <https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/>