WebAssembly in Avionics

Decoupling Software from Hardware

Wanja Zaeske*, Sven Friedrich†, Tim Schubert‡ and Umut Durak§
Safety Critical Systems & Systems Engineering
German Aerospace Center (DLR)
Brunswick, Germany
*wanja.zaeske@dlr.de †sven.friedrich@dlr.de ‡tim.schubert@dlr.de §umut.durak@dlr.de

***Abstract*—Avionics software development is expensive and release cadence is slow. Therefore, reusable software components and applications are particularly attractive to enable faster system development, increase quality and decrease costs. A common Application Programming Interface (API) can avoid strong coupling between software and its execution environment and enable re-use even when the execution environment changes. ARINC 653 describes such an API for avionics applications. However, orthogonal to the use of this API, some coupling to the hardware and the execution environment remains. Usually, an avionics application still has to be adapted to the OS implementing ARINC 653 and compiled for the target hardware architecture. In this paper, we outline a concept for removing this restriction, while increasing portability and re-usability, using WebAssembly as a common Application Binary Interface and object code format. Since WebAssembly is a universal byte-code written for a Virtual Machine, this also enables advantages towards certification and fault isolation. In this work we integrate a WebAssembly (Wasm) interpreter onto an ARINC 653 Hypervisor to demonstrate feasibility of the approach, and to assess runtime impact on binary size and performance. Moreover we argue that certification according to DO-178C is achievable for a Wasm interpreter based avionic software stack.**

***Index Terms*—Avionics, Wasm, ARINC 653, Software**

I. Introduction

Safety critical software demands exceptional rigor during implementation, leading to high costs. Regulative bodies further increase the cost by requiring comprehensive steps in the software development life-cycle: planning process, development process and integral process [1]. These measures ensure safe flight, but the high costs slow down innovation and form an economic barrier to entry, limiting competition. Because of the high cost of development and certification, an incentive towards reuse exists. ARINC 653 was developed to enable software reuse, reducing development costs for avionics by standardizing an Application Programming Interface (API) for avionics applications [2]. Now, vendors like FentISS [1], SYSGO [2], Lynx [3], WindRiver [4] and GreenHills [5] all offer hypervisors that provide the same ARINC 653 API, thus software can be reused by only recompiling it for the next target system.

However, the development process is only a part of the bill. A rigorous integral process still must assure the outputs of the development process. This is due to the nature of DO-178C, the widely applied standard for software assurance in avionics. It mandates that compliance with requirements is demonstrated on the Executable Object Code (EOC). Recompilation to another target generates new EOC, and therefore compliance with requirements has to be demonstrated again, even if the application's source code did not change.

If EOC could be reused, certification costs would be reduced just like the cost of development. However, for that to be feasible, a common Application Binary Interface (ABI) has to be defined. The ABI dictates low-level interfaces such as in-memory representation of data types but also calling convention and the instruction set available on the Central Processing Unit (CPU). Provided with that, a piece of EOC could be executed on different target systems of the same Hardware (HW) architecture. Other HW architectures however use different instruction sets, thus the EOC would not execute on them. The ultimate goal would be an ABI which overarches HW and instruction sets, allowing reuse of EOC on *any* HW.

EOC with these properties is also referred to as universal byte-code. An example for this would be Java byte-code, which together with the Java Virtual Machine (JVM) can be executed on almost any target system. However, some Java features like the Garbage Collector (GC) and its strong reliance on heap memory make it hard to use in real-time, safety-critical systems. A universal byte-code which evolved in the last six years is WebAssembly (Wasm). In this paper we explore the approach of using Wasm as EOC for avionics applications. Both integration with ARINC 653 as well as certification of Wasm in avionics are covered. Our assumptions are put to the test in a small demonstrator, proving the ease of retrofitting Wasm to existing hypervisors.

II. Background

Some foundations of both the Wasm technology and on the discipline of avionics software engineering are required to explain the ideas laid out in the remainder of this paper. This section provides an overview over both.

[1] https://www.fentiss.com/lithos/
[2] https://www.sysgo.com/pikeos
[3] https://www.lynx.com/products/lynxos-178-do-178c-certified-posix-rtos
[4] https://www.windriver.com/resource/vxworks-653-product-overview
[5] https://www.ghs.com/products/safety_critical/integrity_178_extensions.html

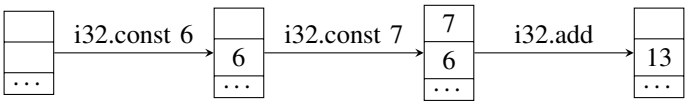

Fig. 1. Example of the Wasm stack machine

A. Wasm, a Universal Bytecode

Wasm is a low-level code format aiming to provide efficient execution and compact representation of program code [3]. While the name implicates a focus on web technologies, Wasm has many areas of application unrelated to the web, a fact both recognized by the Wasm specification [3] and highlighted through other publications [4, 5, 6]. The design goals of Wasm include it being safe, fast, well-defined and open [3]. Further on, Wasm aims to be independent of the Hardware, programming language and software platform/execution environment [3]. Before highlighting how the aforementioned goals are achieved, we make a brief description of the Wasm execution model.

At its heart, Wasm is a virtual stack machine [3, 6, 7]. The instructions of this machine may have arguments, but no real register's exist, instead operands are popped from the stack, results are pushed to it. An example is shown in Figure 1 [3]. In addition to the stack, the runtime provides the so-called linear memory. This memory manifests as an array of bytes, which is addressable only using indices. The index based access to linear memory is beneficial in two ways: it is verifiable that an index is valid, and the physical memory address of the linear memory is opaque to the Wasm program. The linear memory can only grow in well-defined steps ($64\,\mathrm{KiB}$), and it cannot shrink. As the linear memory is indexed using $32\,\mathrm{bit}$ integers, it is constrained to $4\,\mathrm{GiB}$ of size. All memory accesses are indeed checked at runtime, and any invalid access is guaranteed to cause a trap, halting the execution [3]. These semantics of the runtime enable Wasm to be safe: an unsound program can corrupt its own data, but it cannot corrupt a specification conforming Wasm interpreter or its surrounding environment [3, 6]. Other than basic type checking of operands, function calls are also checked: when calling a function whose signature does not fit to the values on the stack, a trap occurs as well [6].

The Wasm format is a compact binary representation — Spies and Mock estimate, that Wasm modules are approximately 2.5 times bigger than native code (not including the additional size of the Wasm interpreter) [5]. Allowing fast single-pass parsing of Wasm, the format enables streaming compilation and validation [3]. Therefore, startup times of Wasm interpreters can be kept small. Because of the low-level nature, Wasm is similar to Intermediate Representation (IR) used in compilers, such as LLVM [6] IR [6]. This simplifies compilation to native code, enabling near native performance.

[6]https://llvm.org/

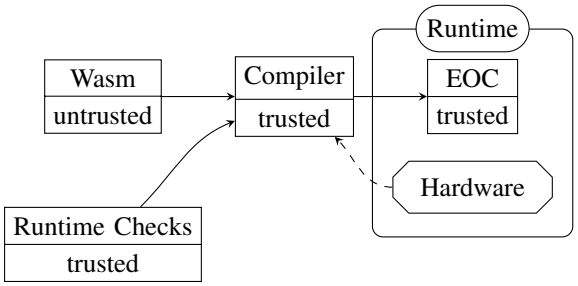

Fig. 2. Compilation of untrustworthy Wasm into sand-boxed object code. Trust means that the final object code is subject to the same sand-box rules as if it was run in a Wasm interpreter

But, when compiling Wasm into a native executable code, checks must be inserted in order to uphold the safety guarantees mentioned above. Once code from an unknown source is executed, it is advisable to not entrust the code with excess privileges. By first compiling to Wasm and then compiling the Wasm to native code combined with checks (see Figure 2) effective sandboxing can be achieved [6, 8]. For small embedded systems such as Cortex-M Microcontroller Units (MCUs), the bounds checks are reported to slow down the execution by a factor of approximately 2 [6]. A deviation from the Wasm specification decreases the slow-down to an approximate factor of 1.5, by wrapping invalid linear memory indices. This does not break the sandbox provided by Wasm, but it violates the Wasm specification and results in Undefined Behavior (UB) within the confines of the sandbox, as the interpreter would normally trap the offending instruction [6]. On bigger CPUs that benefit from speculative and out-of-order execution average slow-downs as low as 1.1 are reported [5]. All the aforementioned results were achieved using some form of compilation, while pure interpretation is orders of magnitudes slower [6]. It is however not necessary to directly compile Wasm to native code before deployment, to benefit from improved performance; instead an interpreter on the target system can employ techniques such as Just in Time (JIT) or Ahead of Time (AOT) compilation to enable fast execution of Wasm [5].

The Wasm specification is an open document providing an authoritative definition of Wasm and its behavior. Only three sources of in-determinism are allowed by the specification: the bit-wise representation of Not a Number (NaN), when growth of the linear memory fails and the behavior of host functions [3, 7]. Due to variance in the implementation of floating point units, the specification allows multiple representations for NaN, of which all may be used non-deterministically. This ambiguity is foreseen in IEEE 754 [9], and modern programming languages such as Rust explicitly declare comparisons between floats as partial; since two instances of NaN are neither equal nor unequal, they simply don't compare. Since the ability of a Wasm interpreter to grow its linear memory is bounded by the amount of physical memory available, the

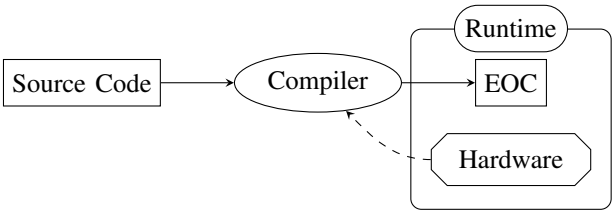

Fig. 3. Traditional compilation: source code is compiled to target dependent object code

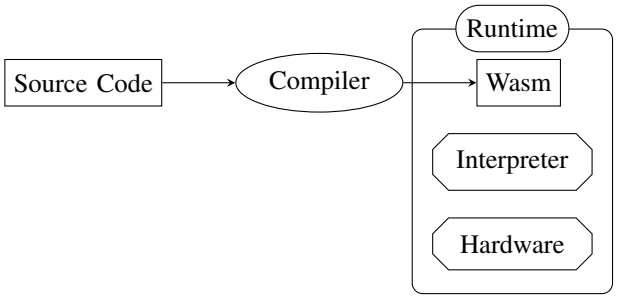

Fig. 4. Compilation for Wasm: source code is compiled to target independent Wasm bytecode

point at which an allocation fails is not deterministic but dependent on the execution environment and its state. Lastly, the exposure of host functions to a Wasm module allows the module to perform arbitrary actions on the outer environment, only constrained by the implementation of said host functions. Apart from these, a mechanized formalization of the Wasm core language has been used to produce a proof of soundness of the Wasm type system [7]. Despite its young age, a lot is already known about Wasm, and existing work proves the preservation and the progress theorem formulated in the Wasm specification [3, ch. 7.5.6][7]. The proofs show that "[...] every thread in a valid [interpreter state] either runs forever, traps, or terminates with a result that has the expected type" [3]. This is not self-evident, bugs in the specification of instruction sets do occur. Wegner discovered soundness issues in documentation provided by Intel for its x86 processors when feeding the examples provided in the manual to a SAT-solver [10].

Finally, Wasm strives for independence from hardware, platforms, and programming languages. Traditionally, when compiling software, the compiler must know the HW architecture of the target environment (see Figure 3). Being a byte-code executed by a Virtual Machine (VM), running on any HW platform is feasible as long as an interpreter executable on said platform is present (see Figure 4). Examples include bare-metal applications on MCUs such as the Cortex-M used in [6] to x86 microprocessors found in servers running full Operating Systems (OSs) as described in [8, 5]. This also illustrates that Wasm can be executed in any software environment, as a plugin in a process running on an OS or as a standalone bare-metal program. Further on, many programming languages can be compiled to Wasm, for example C, C++ and Rust [6, 8, 4, 5]. A software vendor can develop software that then is compiled to Wasm, without having any specific information about the target environment, such as the HW architecture. That software can then be deployed by a system integrator without them knowing in what programming language it was written. In effect, the coupling between development, execution, and execution environment is severely loosened. The caveat of looser coupling is an increase in code size.

B. Software in Avionics

Over the past two decades, two generations of Integrated Modular Avionics (IMA) evolved; the latest form of which foresees multiple software applications to be hosted on a single HW module [11]. The applications are supervised by and executed on top of a hypervisor. Due to the strong emphasis on safety, avionics software is often executed in segregation as partitions [1, 11]. The partitions are isolated spatially and temporally from each other, increasing robustness by containing the effect of failures, preventing transient and cascading faults. Temporal isolation is achieved by allocating CPU time based on a static schedule, each allocation of which forms a partition window. These are enforced through interrupts, preempting one partition at the end of their partition window. Thus, the time consumable by each partition is known a priori, and a deadlock or infinite loop inside one partition is prevented from harming the temporal behavior of other partitions. Spatial isolation is achieved by allowing a process only to access memory and state of its own partition, often using HW mechanisms such as a Memory Management Unit (MMU). Therefore, a faulty process can only harm the state and memory of other processes in the same partition, but any access to memory outside is blocked.

Most meaningful partition compositions require some information to flow between them, e.g. if one partition is functionally dependent on data provided by another. For these cases well-defined channels can be configured [12, 13]; resulting in a system design similar to a separation kernel. In order to enable code re-use and to establish common practices, ARINC 653 [12, 2, 14] provides a standardized API for hypervisors in use by the aviation industry. Today, multiple vendors offer hypervisors featuring the API described in ARINC 653.

III. Related Work

None of the principal ideas described in this paper are novel on their own. The use of interpreted stack machines in avionics dates back at least to the late 1980s, while the significant growth of Java yielded some interesting efforts in the late 1990s and early twenty-first century, for example Real-time Specification for Java (RTSJ). The following subsections

highlight some previous efforts, while differentiating them from the ideas found in this paper.

A. The Pegasus Air-Launched Space Booster

When comparing Wasm to existing programming languages, a striking resemblance is found with the Forth family of languages [7]. Forth is also based on a stack machine, and existing work aiming to formalize some Forth dialects is strikingly similar to the work presented in [7]. Steffy mentions the use of Forth in their avionics design of the Pegasus Air-Launched Space Booster back in 1988. The embedded firmware of telemetry multiplexer, pyro driver units and thruster driver modules were implemented in Forth running on a M68HC11 MCU. The design was eventually qualified to MIL-STD-1540B and MIL-STD-810C. Despite the use of a Wasm like language, software written in Forth was integrated for a non-technical reason: "Personal experience was a selection criteria". [15]

B. The ScanEagle Unmanned Aerial Vehicle

A high-level comparison of Wasm to the JVM seems appropriate, since both technologies share many features. Utilization of a universal byte-code enabling *compile once run everywhere* would be one of them. Based on the RTSJ, Boeing and Insitu developed an avionics suite in the early 2000s for the ScanEagle [16]. Armbruster et al. described the challenges of this project in great detail, while reporting an overall positive experience. Performance was reported to be on-par or even faster than a native C++ version of the onboard software, which is of little surprise considering that the Java code was compiled to native machine code before deployment. Further on, portability is claimed to be better than with traditional languages such as C++ [16]. Implementing the semantics of the RTSJ memory management model proved to be the most challenging, in particular the implementation of scoped memory areas. These areas are not subject to the normal GC, thus all threads using only scoped memory areas are immune to GC induced delays. Other than that, the priority based scheduling is also implemented by the specialized JVM, further complicating matters since the JVM is executed on a real-time Linux kernel, resulting in nested scheduling. The project heavily relied on AOT compilation, converting all Java code to C++, which then was compiled to native code [16] (comparable to Figure 2).

The comprehensive article dives deep into technical matters, focusing primarily on implementing scoped memory and priority scheduling. Both are of no concern for Wasm: Wasm neither implements scheduling nor does it feature a GC — instead memory management lays within the hands of the user. Further on, the work was based on running the Ovm Real-time JVM framework [7] on a real-time Linux. The added complexity from managing a full OS as the foundation for the execution environment is again no concern for Wasm, since Wasm was already proven to work in bare-metal environments. The authors of [16] found multiple ambiguities in the RTSJ specification, which fortunately were addressed in later revisions. Still, they conclude: "The main concern expressed was about the level of maturity of tools and vendor support" [16]. While qualification aspects where discussed, only internal processes of the involved parties are mentioned. The complexity related to GC can be ignored for Wasm, because it does not feature GC. Scheduling on the other hand is not addressed by Wasm but often necessary, thus considerations of scheduling for byte-code VMs are valuable. While Wasm **is** a programming language, the emphasis lies on it being a compilation target. In ScanEagle instead, Java is used as a programming language, but Java code is transpiled to C++ which then in turn is compiled to native code [16].

[7] http://janvitek.org/soft/ovm/index.html

C. DoD Report: Use of Virtual Machines in Avionics

Andersson et al. compiled a comprehensive literature study concerning VMs in avionics. They recite multiple works that consider pure interpreters not to fall under the category of VMs [17]. This stands in stark contrast to DO-332, which explicitly names interpreters as technique to implement virtualization [18]. Unfortunately no further detailing regarding the use of interpreter based VMs is provided in [17], instead the work focuses on the area of HW virtualization which forms the foundation of most established hypervisors. Nonetheless, [17] names multiple examples of abstract interpretation for Worst-Case Execution Time (WCET) analysis. This has some similarity to the concept introduced in subsection IV-C, where instead of abstract interpretation, instructions used during actual interpretation are measured.

IV. Integrating a Wasm Interpreter in Avionics

As explained in the section II, there are two ways of using Wasm. It can be either executed by an interpreter (see Figure 4), or it is compiled further to native code together with runtime bounds checking to uphold the sandbox guarantees (see Figure 2). The utilization of AOT and JIT compilation still belongs to the former, the distinctive factor is that a Wasm module is passed to an interpreter during runtime for interpretation. In the case of the latter, there is no interpreter at runtime, and Wasm is not treated as EOC, but as IR in the compiler toolchain. When comparing the two, interpretation results in loose coupling, while compilation enables higher performance. This paper focuses on the former; thus for the execution of Wasm we integrate an interpreter to the avionics software stack.

A. Embedding the Interpreter

The stack often consists of a hypervisor hosting multiple applications. The partitioning mentioned in subsection II-B does not only improve robustness. It also eases certification, as it enables mixed criticality systems: one hypervisor can host applications of varying criticality, and each application can be certified to its respective level. If there was no partitioning, all applications running on the same HW would have to be certified to the level of the most critical application [1]. As described in Figure 3, each application would be compiled

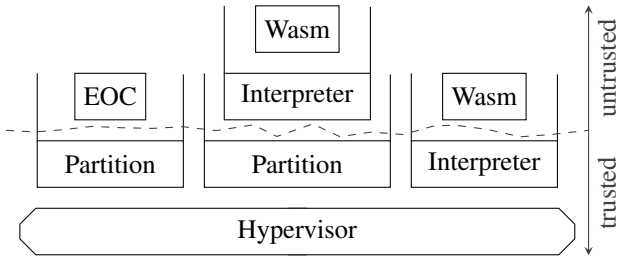

Fig. 5. Approaches towards integration of Wasm interpreters into IMA. An untrusted components' failure cannot harm other components. A trusted component must not fail in order to uphold temporal and spatial isolation

from source to EOC, which could directly run on the hardware. But, instead, the Executable Object Code is allocated to a partition in the hypervisor, as illustrated in the left section of Figure 5.

Now, to integrate a Wasm interpreter, the interpreter can be compiled to EOC, which is allocated to a partition. This scenario, depicted in the mid-section of Figure 5, is very useful to retrofit Wasm to an existing hypervisor, because there is no apparent difference to executing any other EOC in a partition. The platform's hypervisor and partitioning remain unchanged, and the interpreter needs certification only to the same level as the most critical application running inside it. However, Wasm offers substantial sandboxing features, the strict runtime checks on memory access implement spatial isolation, and the fuel concept introduced in subsection IV-C enables a form of temporal isolation.

Thus, a third option lends itself, a hypervisor dedicated to Wasm applications. In this case, the spatial partitioning is provided by the Wasm interpreter itself, while the hypervisor only orchestrates and schedules the interpreter instances. This case, outlined in the right section of Figure 5, moves the Wasm interpreter into the platform, requiring its implementation to be trusted to the highest level. Until now, avionic hypervisors often relied on HW mechanisms such as MMU or Memory Protection Unit (MPU) to enforce the spatial isolation between partitions. With Wasm this is not required, as similar guarantees can be provided by the Wasm sandbox purely in software.

B. Exposing the ARINC 653 API

Provided with some options to integrate a Wasm interpreter in the avionics stack, the next question is the integration of software running inside said interpreter with the rest of the system. First, we consider a typical API offered by a hypervisor to applications running inside partitions. The aforementioned ARINC 653 describes an API for applications in partitions to interact with the hypervisor and other partitions [14]. ARINC 653 is separated into parts, each covering different aspects. P0 introduces the basics for the other parts [14], P1 defines a base API [13] while P2 extends P1 API [19]. P4 on the other hand defines a reduced subset of the P1 API [20]. All the aforementioned parts can be expressed as sets of functions and data types, and ARINC offers C header files for the P1 and the P2 APIs [8].

On the other hand, Wasm allows exposing functions from the outside to the execution environment, allowing Wasm to call native code, and vice versa, allowing native code to call functions defined in Wasm. At first glance, it looks like this already enables porting of the ARINC 653 API to Wasm; one simply has to expose all the functions to the Wasm code. While this holds true for many of the aforementioned functions, it does not hold for all of them. Namely, ARINC 653 allows for multiple processes in a given partition, but Wasm does not yet provide a mechanism for spawning a new process or thread. The `CREATE_PROCESS` call requires a memory address which is interpreted as a function pointer. Wasm however does not share an address space with the native code environment, thus one cannot pass a Wasm function table index as a function pointer to native code. When retrofitting an existing hypervisor, a scheme providing function pointer translation is needed, and new interpreter instances need to be spawned ad-hoc. Such a scheme requires either compromise due to additional state management, or reliance on hypervisor specific behavior. For a new Wasm native hypervisor these detours can be avoided. Interesting to note is, that ARINC 653 P4 restricts the process model to exactly two processes, one background aperiodic and one high priority periodic process. We estimate that most avionics applications work with a handful of processes, which most often are spawned directly at partition warm-up, eluding the performance penalty of process creation in many cases. These cases can work well with interpreter instances manually spawned per process, however resulting in a stronger coupling to the Wasm code. Besides the process management functions, exposing the API in Wasm is as straightforward as laid out above.

A key ingredient left to do is the definition of an ABI. Wasm only supports a handful of data types, and the layout of more complex types is dependent on the source language. For example, two structs in Rust and C++ might have a different layout when compiled to Wasm, despite seeming to be semantically equivalent. To enable true independence, the ABI needs to be well-defined for all ARINC 653 data types down to the level of individual bytes. Wasm Interface Type (WIT) is a work-in-progress proposal aiming to define a programming language agnostic interface description language for Wasm, easing the handling of complex types [9].

C. Measuring Algorithmic Cost Independent of HW

Another aspect of Wasm is the fuel concept, sometimes also referred to as gas or metering. While not yet formalized in the specification, many Wasm interpreters such as Wasmtime [10], Wasmer [11] and wasmi [12] allow measuring and limiting the

[8] P1 https://www.aviation-ia.com/support-files/arinc653h; P2 https://www.aviation-ia.com/support-files/arinc653p2h

[9] https://github.com/WebAssembly/component-model/blob/main/design/mvp/WIT.md

[10] https://wasmtime.dev/

[11] https://wasmer.io/

[12] https://github.com/paritytech/wasmi

execution time of Wasm. The concept works as follows: if fuel is enabled, an interpreter instance has to be *filled up* with fuel. Each Wasm instruction consumes a discrete amount of fuel. If the interpreter runs out of fuel, the execution is trapped and a respective error is yielded by the interpreter. Most instructions like the `i32.add` in Figure 1 consume only a single unit of fuel, theoretically allowing progress as long as there is any fuel left. An example use-case for fuel in Wasm is provided by Watt, where it is used to avoid solving the halting problem during the formulation of a mechanized proof [7]. Internally, the fuel tracking is often implemented per code block, not per instruction, to avoid excessive overhead. The code is divided into basic blocks, separated by branches, and for each block the total fuel consumption is summed up. Because there are neither locks [13] nor branches *in* any block, it is sufficient to check at every branch if the amount of fuel in the interpreter is bigger than the fuel consumed by the next block.

This enables a couple of interesting properties. As the check for sufficient fuel is made before executing the next block, an out-of-fuel situation can be handled *before* it occurs. Compare that to a preemptive scheduler, which observes a deadline violation only when it occurs, having to conduct the appropriate fault handling *after* the fact. For a deterministic program the fuel cost is deterministic with regard to the inputs. If an expected fuel consumption for an algorithm is known, the percentage of completion can be approximated from the amount of fuel already consumed. While all of the above can be achieved with instrumentation of code or hardware tracing as well, a striking difference remains: the existence of the Wasm interpreter with its information about the program state makes this possible without tracing hardware and without insertion of instrumentation code into the EOC.

The fuel consumption of an algorithm with provided inputs can be measured in an interpreter on a development machine, independent of the target system. Then, the consumption can be compared to the typical execution speed in fuel per time of a target HW, to get a first estimate on the required execution time without ever running the algorithm on the target HW.

For HW with a highly deterministic temporal behavior, the fuel concept can assist in a divide-and-conquer approach towards WCET analysis of any algorithm a with a known fuel cost: first the slowest Wasm operation o_s is determined.

$$o_s : \forall o \in \text{WASM operations} : \frac{\text{wcet}(o)}{\text{fuel}(o)} \leq \frac{\text{wcet}(o_s)}{\text{fuel}(o_s)}$$

Then, an upper bound in fuel cost for a certain algorithm is determined, either experimentally or by static code flow analysis. Now the WCET of the algorithm can be calculated as

$$\text{wcet}(a) \leq \text{fuel}(a) \cdot \text{wcet}(o_s)$$

This analysis is very pessimistic, assuming that any consumed fuel will yield the maximal WCET per fuel *possible*. To provide less pessimistic WCETs, the algorithm can be applied per block, delivering a per block WCET. For a block that only contains relatively fast operations (small $\text{wcet}(o)/\,\text{fuel}(o)$), this leads to smaller bounds. The so determined per block WCET can be derived while loading a Wasm module, allowing to detect future deadline violations at least one basic block ahead. Both approaches must be enhanced with the cost of checking for fuel on every branch and the fuel consumption of any native code callable from the Wasm needs to be modeled properly to yield robust results. The typical interference problems must be accounted for as well, and due to the overhead of interpretation, interference may be more pronounced when compared to executing native code directly.

It remains to be seen whether the fuel concept can be exploited to replace traditional scheduling and temporal isolation. Irrespective of that outcome, it may be a useful tool for development and runtime monitoring.

V. Evaluation

In order to put our claims from section IV to the test, we have developed a small demonstrator [14]. For this, our open source Terrain Awareness and Warning System (TAWS) [15] was compiled to Wasm. The wasmi Wasm interpreter was integrated into a partition using our Rust wrapper a653rs [16] [21]. Similarly to this partition, another one is developed for comparison, utilizing our TAWS natively without Wasm. A third partition provides simulated flight data to the aforementioned partitions and receives the alerts emitted by the TAWS.

With this setup we were able to validate several of our assumptions. While the integration was completed in a couple of days, some clear limitations (i.e. regarding processes, see subsection IV-B) as well as the extra cost of this approach became apparent. The first immediate drawback we noticed was the loss of Rust's guarantees regarding its type system when exposing functions and variables to or from a Wasm module. Even though Wasm's native types are checked, if non-native types are used, pointers to linear memory are necessary. These do not carry any type information, comparable to void pointers in C. This effectively renders any programming languages type system powerless in regard to the Foreign Function Interface (FFI) as soon as types not directly representable in Wasm are used. Another drawback is the larger binary size of the partitions. The natively compiled TAWS partition image has a size of $489\,\text{kB}$, but the Wasm based partition's image is $4.2\,\text{MB}$ large, of which $1.1\,\text{MB}$ are for the embedded Wasm file and another $1.6\,\text{MB}$ for a memory allocator pool that is necessary for wasmi. Inclusion of the memory allocator pool in the partition image can be avoided depending on the specific process of building partition images. This leads us to believe that the wasmi Wasm interpreter itself inflates the partition images by approximately $1.5\,\text{MB}$, resulting in an overall 8 times larger image size for the Wasm based TAWS. The Wasm module itself is 2.2 times bigger than the native code, an

[13] the Wasm instructions do not include locks or atomics [3]

[14] https://github.com/DLR-FT/a653rs-wasmi-taws-demonstrator

[15] https://github.com/aeronautical-informatics/openTAWS

[16] https://github.com/DLR-FT/a653rs

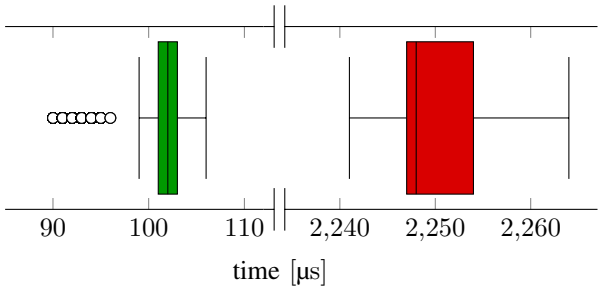

Fig. 6. Performance evaluation. For our example, interpreted Wasm is approximately 20 times slower than native code

increase close to the factor of 2.5 measured by Spies and Mock.

Concerning the computational overhead introduced by Wasm, we conducted a benchmark measuring the duration it takes for new flight data to be processed in the TAWS. Our results, as illustrated in Figure 6, show that the identical TAWS algorithm takes approximately 20 times longer when ran inside the wasmi Wasm interpreter. The jitter for both cases is in similar bounds, approximately $\pm 7.5\,\mu s$ for native code and approximately $\pm 11.5\,\mu s$ for Wasm.

VI. Certification

Avionics software is subject to rigorous certification and validation. A document recognized by aviation regulation agencies worldwide is DO-178C. It provides guidelines in the form of commentary and objectives for the development of software in avionics. The set of objectives to be carried out on a software component is dependent on its safety impact; a big impact results in a comprehensive set of objectives, while software with no impact on safety may be exempt from all objectives. The impact is grouped in levels ranging from A to E, where A mandates all objectives and E mandates none. DO-178C is extended by supplementary standards such as DO-330 (Tool Qualification) [22], DO-331 (Model Based Development and Verification) [23] and DO-332 (Object Oriented Technology) [18]. It makes sense to review some definitions provided by the standard to understand how the 71 objectives of DO-178C apply to Wasm.

- *Code*: "The implementation of particular data or a particular computer program in a symbolic form, such as Source Code, object code, or machine code." [1]
- *Object Code*: "A low-level representation of the computer program not usually in a form directly usable by the target computer but in a form which includes relocation information in addition to the processor instruction information." [1]
- *Executable Object Code*: "A form of code that is directly usable by the processing unit of the target computer and is, therefore, a compiled, assembled, and linked binary image that is loaded into the target computing hardware." [1]
- *Parameter data item*: "A set of data that, when in the form of a Parameter Data Item File, influence the behavior of the software without modifying the Executable Object Code and that is managed as a separate configuration item. Examples include databases and configuration tables." [1]

While Wasm matches the definition of *Code*, it seemingly does not fit the definitions of Object Code and/or EOC. The EOC definition presumes that the code is directly usable by the target computer's processing unit, but Wasm would instead be interpreted by a virtual processing unit that is implemented as software running on the physical processing unit. Similarly, the *Object Code* definition mentions processor instruction information to be present, but again Wasm is not a format compatible with the physical processing unit. This distinction however is important: DO-178C demands evidence that the EOC complies with the requirements [1, ch. 6.4].

Instead, Wasm fulfills all features described in the definition of *parameter data item*: it does influence the behavior of EOC, but it does not change the EOC. Despite that, treating Wasm as data instead of software is a semantic hack prohibited by DO-332: "When verifying such programs [interpreters], the data processed as instructions by the program should be verified as executable code (not data)." [18, ch. OO.4.2m, OO.D.1.7, OO.D.1.7.3].

In addition, DO-332 names interpreters as a virtualization technique while proclaiming that, if virtualization is used, the term *target computer* means *target environment*. The target environment is further detailed as a "combination of virtualization software and a target computer" [18, ch. OO.4.0]. Taking these additions into consideration, a clearer picture forms: Wasm **is** either Object Code or EOC. The respective terms used by the Wasm tool conventions are *WebAssembly object file* and *executable WebAssembly module* [24]. A Wasm object file is an executable module only if it does not contain a `linking` or a `relocation` section, similar to the DO-178C definitions for (Executable) Object Code.

Now that we established that Wasm modules can be treated like EOC, it is subject to the same objectives as an executable binary derived from compiling source code [18, ch. OO.D.1.7.3]. Still, there are a few details to keep in mind. First, DO-178C allows simulation and emulation both for testing and integration, explicitly allowing certification credit to be granted for testing done on a target computer [1, ch. 4.4.3, 5.4.2b]. Since DO-332 declares that interpreters provide emulation [18, ch. OO.D.1.7], a Wasm interpreter running on the host (not target) computer can be used to gain certification credit. Due to possible differences in behavior, such as timing, some tests still need to be carried out with the interpreter on the target computer [1, ch. 4.4.3b, 6.4.1a]. Use of virtualization and the associated executable program have to be described in the software planning process [18, ch. OO.4.2m, OO.4.4g] and must be considered for the software life cycle environment plan. Verification activities on the EOC for a virtualization technique need to be considered in the context of the technique itself [18, ch. OO.D.1.7.3], for example because

of the potential impact of scheduling inside the technique [18, ch. OO.D.2.4.3.3]. When considering Wasm, some of these demands have little to no impact: there is no inherit scheduling to the execution of Wasm. The formal specification of a Wasm interpreter's behavior lays a solid foundation to find and resolve any differences between the functional behavior of interpreters. Indeed, aside from some degrees of freedom related to the handling of floating point numbers any deviating behavior of two Wasm interpreters can be attributed to a bug in either one, and the Wasm specification provides authoritative information on which behavior is correct. But especially considering the performance penalty for pure interpretation of Wasm, special attention is due in view of differences in non-functional behavior of interpreters, such as time consumed to perform a certain algorithm. An open question remains: how to approach AOT and JIT compilation on the target computer. Both techniques generate or modify the EOC present on the target computer, violating or at least endangering the compliance with various objectives. We estimate that further clarification from regulators would be required to enable these techniques.

VII. Discussion

This paper introduces the idea of deploying avionics applications as Wasm. Numerous advantages are highlighted: Wasm creates independence between execution environment (especially HW) and software. It does so while being language agnostic, i.e., many languages including C, C++ and Rust can be compiled to Wasm. As Wasm is a unified compilation target, no specific knowledge of the target HW is necessary to compile for it. This does not touch interface design, e.g. in order to be useful a common API and ABI between the Wasm code and the target environment must be agreed upon. There are no specific blockers for integrating Wasm into a typical avionics software stack, and ARINC 653 could be used as the agreed API between applications and target environment. The inherent inspectability of a Wasm interpreter enables analysis capabilities, such as EOC coverage independent of the target HW. Further on, the fuel concept found in some Wasm interpreters can be used to enhance the temporal predictability and ease the development of real-time systems. From a certification perspective the utilization of interpreters is foreseen, and specific requirements are described in DO-332. Specific formulations exist in DO-178C and DO-332 that allow the exploitation of a portable execution environment, i.e. tests performed on an interpreter simulating the target environment can yield certification credit.

The open nature of Wasm combined with multiple API **and** ABI compatible hypervisors might have an impact on economics as well. Competition is increased: smaller companies can develop software on one hypervisor, but since the software is adaptable to other hypervisors with little effort, they can compete in a bigger market without having to buy more hypervisors. Most importantly, ABI compatibility enables development for a target system without requiring possession of said target system, lowering the barrier to entry. Still, access to the target system remains necessary during the integral process [1].

Unfortunately, there are a couple of significant drawbacks associated with our approach: The performance penalty for interpretation is hefty. For small applications, like state machines or traditional control algorithms this may be tolerable, but many algorithms of higher computation complexity will be too slow on a Wasm interpreter. The reliance of the civil aviation sector on rather dated HW, such as sub-GHz single core CPUs, can be interpreted in favor or against this: either there is no pressing need for faster HW, or because of the limitations in HW performance, efficient execution is of even higher importance.

While implementing our demonstrator, we ran into multiple occasions of ABI incompatibility with complex types (i.e. structures). These form am Achilles heel for Wasm: while there is a stable ABI covering primitives, these are not sufficient for many use-cases. Thus, one is likely to observe bugs caused by differing memory layouts of complex types. Especially when mixing programming languages that are compiled to Wasm or when defining Wasm to host environment interfaces caution is advisable. We eagerly await WIT to stabilize, as we expect it to address these issues.

While we established a foundation, more work remains. The performance issue could be resolved using AOT or JIT compilation, which allow near native performance for Wasm. However, both techniques rely on runtime generation of EOC, a behavior which is not foreseen by current regulations. Bosamiya et al. demonstrated that formally proven Wasm to native code generation is possible [8], which greatly improves the safety case towards AOT and JIT compilation. Further clarification of the regulating bodies would therefore be welcome.

Another interesting aspect is the implementation of drivers in Wasm. Most drivers rely on memory mapped configuration registers, thus implementation of drivers requires raw memory access. This conflicts with the interface specified by Wasm, where pointers can only reference addresses in the linear memory. Two potential solutions seem feasible: the respective raw memory regions are mapped to a specific part of the linear memory. This however is error-prone, as normal use of these memory sub-slices by the Wasm interpreter may result in arbitrary side effects on the HW. A better approach is the utilization of *multiple* linear memories, a feature which is not yet specified but already prepared for in the Wasm specification. Further investigation is required, once this feature lands in more Wasm interpreters. The answer to the driver question enables performance improvements through the use of accelerators: The principle of mapped linear memory can be used to efficiently interface with accelerator HW.

In subsection II-A we briefly introduced the possibility of using Wasm as an IR for further compilation. As an IR, Wasm is particular handy to introduce checks that guarantee sandboxing, i.e. temporal- and spatial isolation. This method of sandboxing deserves its own investigation with further consideration of the formal-methods specific DO-333 standard [25].

VIII. Conclusion

This paper discusses various aspects of using Wasm in avionics to achieve a common ABI. We describe how Wasm interpreters can be integrated into the avionics software stack found in current IMA architectures. Multiple integration scenarios are discussed, one of which enables retrofitting of Wasm with little effort. To make use of current platforms, integration of established standardized APIs into Wasm is discussed. Namely, ARINC 653 is mostly compatible with Wasm, but ARINC 653's process management API. This is followed by an introduction to the fuel concept in Wasm, an interpreter instruction count based metric that may be used in the context of IMA to both implement some form of temporal isolation and enable WCET estimations independent of the target HW. A small demonstrator asserts our initial assumptions about retrofitting Wasm to an existing hypervisor, while also providing a sample of the performance and memory impact when interpreting Wasm without AOT and JIT compilation. Lastly, an analysis of DO-178C and DO-332 outlines the relevant details for certification of a system comprising a Wasm interpreter.

We are positive that Wasm could enable a unified ABI for avionics software. Such an ABI allows the reuse of already compiled software across vastly different HW. Because Wasm can be treated as EOC, the required certification effort is reduced. On the technical site, two problematic areas should be alleviated: the error prone FFI, and the big performance penalty of interpretation. For the former WIT might become a viable solution, the latter might be addressable by AOT compilation. This however requires clarification by regulative bodies as well as further engineering. A wider discussion among the aviation community would be compulsory for agreement on a Wasm based ABI. Finally, certification agencies need to decide how much certification evidence they agree to be transferable when reusing a Wasm module.

References

[1] RTCA, "Do-178c software considerations in airborne systems and equipment certification," RTCA, Standard, 2011.

[2] A. Cook and K. Hunt, "Arinc 653 — achieving software re-use," *Microprocessors and Microsystems*, vol. 20, no. 8, pp. 479–483, 1997. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933197011137

[3] W. C. Group, "WebAssembly Specification release 2.0 (draft 2023-06-06)," World Wide Web Consortium, Standard, Jun. 2023.

[4] S. Wallentowitz, B. Kersting, and D. M. Dumitriu, "Potential of webassembly for embedded systems," in *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, 2022, pp. 1–4.

[5] B. Spies and M. Mock, "An evaluation of webassembly in non-web environments," in *2021 XLVII Latin American Computing Conference (CLEI)*, 2021, pp. 1–10.

[6] G. Peach, R. Pan, Z. Wu, G. Parmer, C. Haster, and L. Cherkasova, "ewasm: Practical software fault isolation for reliable embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3492–3505, 2020.

[7] C. Watt, "Mechanising and verifying the webassembly specification," *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2018.

[8] J. Bosamiya, W. S. Lim, and B. Parno, "Provably-Safe multilingual software sandboxing using WebAssembly," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1975–1992. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/bosamiya

[9] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[10] Z. Wegner, "x86-sat," https://github.com/zwegner/x86-sat, accessed: 2023-06-09.

[11] RTCA, "Do-297 integrated modular avionics (ima) development guidance and certification considerations," RTCA, Standard, 2005.

[12] A. Cook, "Arinc 653 — challenges of the present and future," *Microprocessors and Microsystems*, vol. 19, no. 10, pp. 575–579, 1995. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0141933196841585

[13] ARINC, "Avionics Application Software Standard Interface part 1 required services," Aeronautical Radio Incorporated, Bowie, MD, USA, Standard, Dec. 2019.

[14] ——, "Avionics Application Software Standard Interface part 0 overview of arinc 653," Aeronautical Radio Incorporated, Bowie, MD, USA, Standard, Aug. 2019.

[15] D. Steffy, "The pegasus avionics system design," in *9th IEEE/AIAA/NASA Conference on Digital Avionics Systems*, 1990, pp. 326–332.

[16] A. Armbruster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek, "A real-time java virtual machine with applications in avionics," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 1, dec 2007. [Online]. Available: https://doi.org/10.1145/1324969.1324974

[17] B. Andersson, S. Chaki, and D. de Niz, "Use of virtual machines in avionics systems and assurance concerns," Carnegie Mellon University, Tech. Rep. DOT/FAA/TC-19/22, oct 2019.

[18] RTCA, "Do-332 object-oriented technology and related techniques supplement to do-178c and do-278a," RTCA, Standard, 2011.

[19] ARINC, "Avionics Application Software Standard Interface part 2 extended services," Aeronautical Radio Incorporated, Bowie, MD, USA, Standard, Dec. 2019.

[20] ——, "Avionics Application Software Standard Interface part 4 subset services," Aeronautical Radio Incorporated, Bowie, MD, USA, Standard, Dec. 2012.

[21] S. Friedrich, E. Engler, T. Schubert, W. Zaeske, and

U. Durak, “Assuring apex with a versatile rust api,” in *embedded world 2023 Conference Proceedings*. WEKA FACHMEDIEN GmbH, Mar. 2023, pp. 298–305.

[22] RTCA, “Do-330 software tool qualification considerations,” RTCA, Standard, 2011.

[23] ——, “Do-331 model-based development and verification supplement to do-178c and do-278a,” RTCA, Standard, 2011.

[24] “Webassembly object file linking,” https://github.com/WebAssembly/tool-conventions/blob/main/Linking.md, accessed: 2023-06-07.

[25] RTCA, “Do-333 formal methods supplement to do-178c and do-278a,” RTCA, Standard, 2011.