WEBASSEMBLY FOR SAFETY-CRITICAL SYSTEMS

From Sandbox to Flight Deck

Umut Durak

DLR

German Aerospace Center (DLR)

DLR is the national aeronautics and space research centre of the Federal Republic of Germany.
It has 30 locations and almost 12 000 employees.

German Aerospace Center (DLR)

Institute of Flight Systems

The Institute of Flight Systems is active in the topics of flight mechanics, control and system technology of all flying systems at the DLR site Braunschweig since 1953.

A REAL-TIME SYSTEMS PROBLEM

The hardware moves underneath us

Hardware changes for reasons that have nothing to do with the software it runs.

1

Components age out faster than certifications.

Typical avionics certification lifecycle: 15–20 years. Typical semiconductor product lifecycle: 5–7. The mismatch is structural.

2

New functions need new silicon.

Sensor fusion, integrated modular avionics, vision-based perception. The workload pulls toward multicore, GPUs, accelerators. Legacy single-core sub-GHz CPUs cannot host the next decade.

3

Sovereignty redraws the supply chain.

RISC-V is no longer hypothetical for aerospace. Vendors are now shipping e.g. Microchip's PolarFire SoC. ISA choice is increasingly a sovereignty and supply-chain decision.

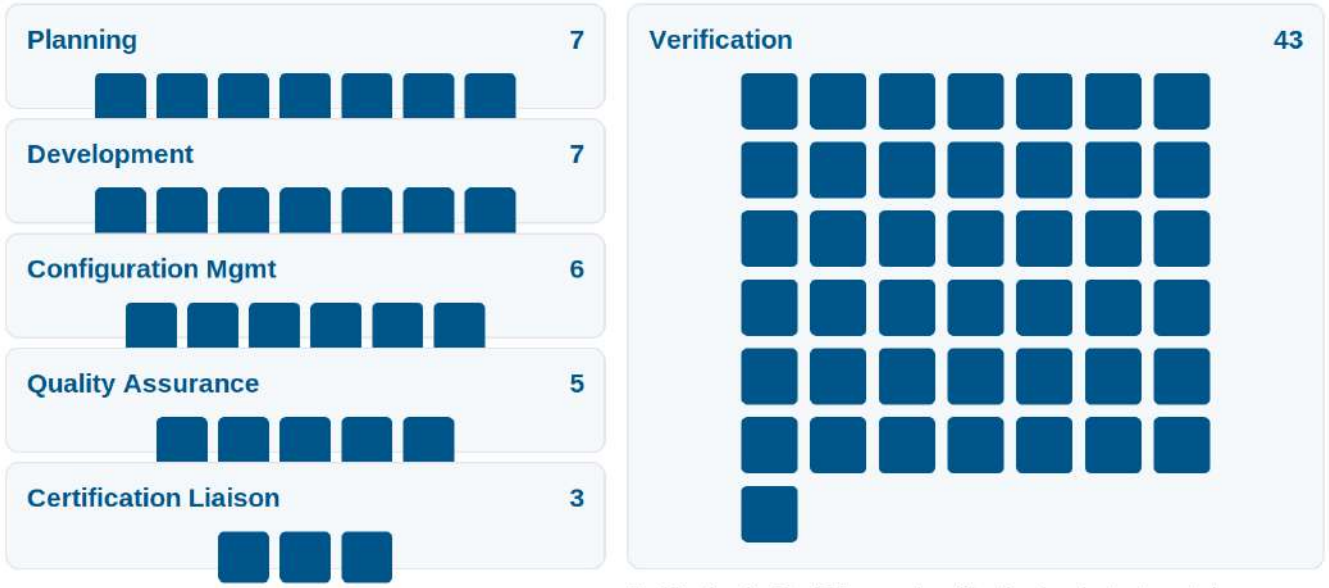
None of these are software decisions. Yet each one re-triggers certification of unchanged software.

What is this thing called Certification?

Anatomy of an Objective

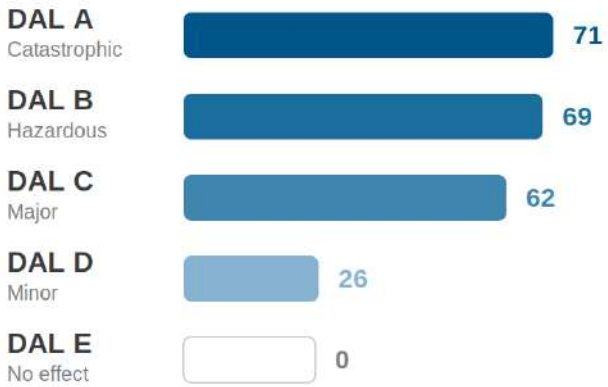
	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	High-level requirements comply with system requirements.	6.3.1.a	6.3.1	●	●	O	O	Software Verification Results	11.14	②	②	②	②

DAL A — all 71 objectives apply



Verification is 43 of 71 — and verification is what a target change re-opens.

Criticality sets the count



Rigour scales too

structural coverage: C statement · B decision · A MC/DC

The re-certification bill, for software that did not change

Photo: Christopher Doyle / CC BY-SA 2.0.

door_monitor.c — cabin door monitor · DAL A · 20 ms period

```
if (weight_on_wheels && cabin_pressure_kpa < 85.0f)  return SAFE;
if (!door_closed || !door_locked)                    return ALERT;
```

PowerPC e500mc — big-endian · hard FPU · 252 B

```
stwu    r1,-32(r1)
stw     r31,28(r1)
mr      r9,r3          ; door_closed
stb     r9,15(r31)
...
lbz     r9,13(r31)   ; weight_on_wheels
cmpwi   r9,0
beq     58
lfs     f12,8(r31)    ; cabin_pressure
fcmpu   cr0,f12,f0     ; HW float compare
bge     58
li      r9,0           ; SAFE
```

Arm Cortex-R5 (no FPU) — little-endian · soft float · 112 B

```
push    {r7, lr}
sub     sp, #8
mov     r3, r0         ; door_closed
strb    r3, [r7, #7]
...
ldrb    r3, [r7, #5] ; weight_on_wheels
cmp     r3, #0
beq.n   32
mov.w   r1, #0         ; 85.0f lo
bl __aeabi_fcmplt    ; SW float (libcall)
cmp     r0, #0
beq.n   32             ; SAFE
```

DO-178C COST

Twelve of seventy-one DO-178C objectives should be revisited at full rigour.

NOT AN AVIATION PECULIARITY

The same coupling, every safety-critical domain

Aviation	`DO-178C`	✓ compliance shown on the Executable Object Code
Automotive	`ISO 26262`	✓ software-unit verification on the object code
Industrial	`IEC 61508`	✓ verified safety function tied to the target
Space	`ECSS-Q-ST-80`	✓ object-code-level verification evidence

Wherever a standard pushes verification onto the binary, a hardware change re-triggers certification.

THESIS

Unweld the software from the hardware

WebAssembly is the first mainstream byte-code with the unsafe constructs carved out.
So, the idea is to re-certify the interpreter in hardware update, not the application.
Making that hold under hard real-time is what brings us to ECRTS.

ACT 1
Sandbox
formal semantics · no GC

ACT 2
Flight Deck
DO-178C · ARINC 653

ACT 3
Real-Time Lens
fuel · WCET · interference

ACT 4
The Stack
architecture → model-based

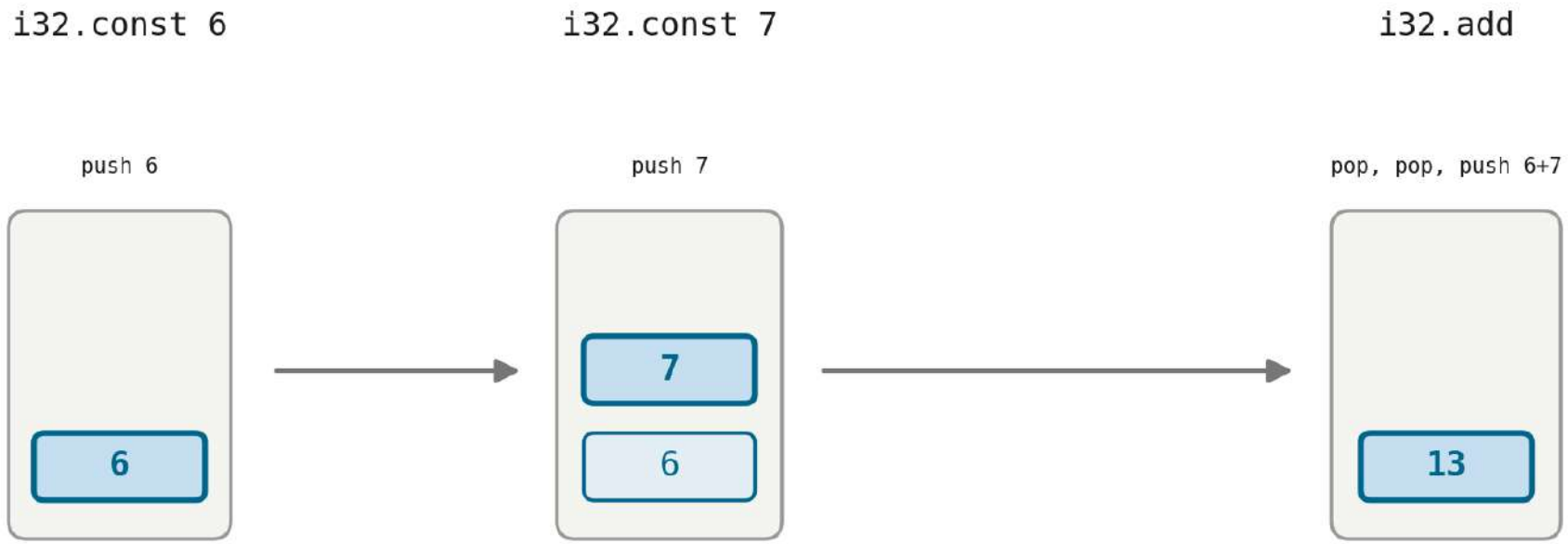
Three claims about the runtime, then we follow the consequences into architecture and tooling.

WebAssembly, and why its origin matters

- A compact byte-code for a virtual stack machine shipped in all major browsers 2017, W3C standard 2019, born in the browser for safe, fast, portable execution.
- A compilation target: C, C++, Rust, Ada (via LLVM / GNAT) all compile to it.
- Designed to be independent of hardware, language, and execution environment. The exact independence avionics needs.
- Executed by an interpreter or AOT/JIT-compiled with inserted sandbox checks. We focus on interpretation: loose coupling valued over peak performance.

The browser people built it for security and performance. **Those turn out to be our constraints too.**

A virtual stack machine



Operands are popped, results pushed — the machine has no registers.

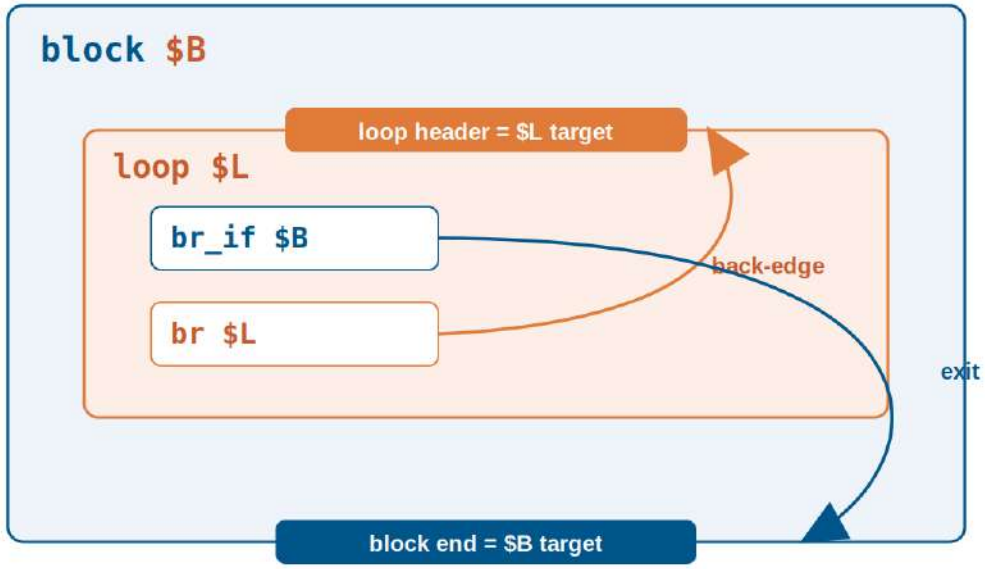
No calculated jumps, structured and typed control flow.

Structured control flow

WASM BYTECODE

```
func $f
  block $B
    local.get $x
    loop $L
      (body…)
      br_if $B          ;; exit block
      (body…)
      br $L             ;; back to header
    end                 ;; loop
  end                   ;; block
```

STRUCTURED CONTROL FLOW



Every branch targets an enclosing label — no goto, no arbitrary address.

Inside a Wasm instance

Two stacks and one byte array

EXECUTION STATE

Operand stack

the value stack

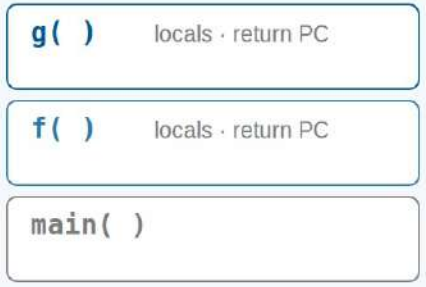
push / pop at the top

holds: operands mid-computation

depth and types known at each point

Call stack

frames



each call pushes a frame; return pops it

holds: locals + return point, per call

grows / shrinks with call depth

DATA

Linear memory

the program's data

every access bounds-checked → out of range traps

holds: a flat byte array it manages itself

separate from code and the stacks

Two stacks hold the execution state. One byte array holds the program's data.

Linear memory & the Harvard model

- Linear memory is a single indexed byte array. Pointers become indices; every access is bounds-checked at runtime, and an out-of-bounds access is guaranteed to trap.
- Harvard-style separation makes injected data non-executable; structured control flow with no writable return-address stack leaves return-oriented programming nothing to hijack.
- No garbage collector. Memory management stays with the application.
- The sandbox is outbound-only spatial isolation in software: an unsound module corrupts only its own data, never the interpreter or its neighbours.

Is that spatial isolation without an MMU? Revisited in Act 2.

Three sources of non-determinism

1	**NaN bit patterns**	IEEE-754 leaves some NaN payloads open; the spec permits the variation explicitly.
2	**Memory-growth failure**	whether grow() succeeds depends on host resources.
3	**Host-function behaviour**	whatever the embedder exposes across the boundary.

Everything else is fully determined by the specification.

Formally grounded

- * Watt (2018) : a machine-checked proof of Wasm's type soundness. It was rigorous enough that it found and fixed real bugs in the official spec.
- Preservation & progress: every thread in a valid interpreter state continues, traps, or returns a value of the expected type, nothing else.
- **Rao et al. (2025): the verified semantics is the interpreter. Proof and implementation in one definition, efficient enough to run, no longer maintained as separate artifacts.

A clear specification of semantics **is exactly what DO-333 asks for to certify an interpreter.**

* Watt — Mechanising and Verifying the WebAssembly Specification. CPP 2018.
** Rao, Radziuk, Watt, Gardner — "Progressful Interpreters for Efficient WebAssembly Mechanisation" , POPL 2025·

Accidental fitness

If you had asked this room in 2007 to design a portable byte-code for safety-critical real-time systems…

formally specified semantics	**Harvard memory, no GC**	**typed stack machine**
bounds-checked sandbox	**only few to no non-determinisms**	**structured control flow**

…you would have written down something that looks a lot like WebAssembly. The browser got there first, for entirely different reasons.

The constraints imposed for web security and speed converge to the constraints of certifiability.

Why didn't earlier byte-codes survive avionics?

Pegasus (1988)	`FORTH stack machine`	worked but chosen for personal familiarity, not portability, used as embedded firmware language
ScanEagle (2000s)	`Real-Time Java / RTSJ`	scoped memory + GC + real-time threading on an RT-JVM. It never generalised
WebAssembly	`virtual stack machine`	no scoped memory, no GC, no threading. The features that hurt are simply absent

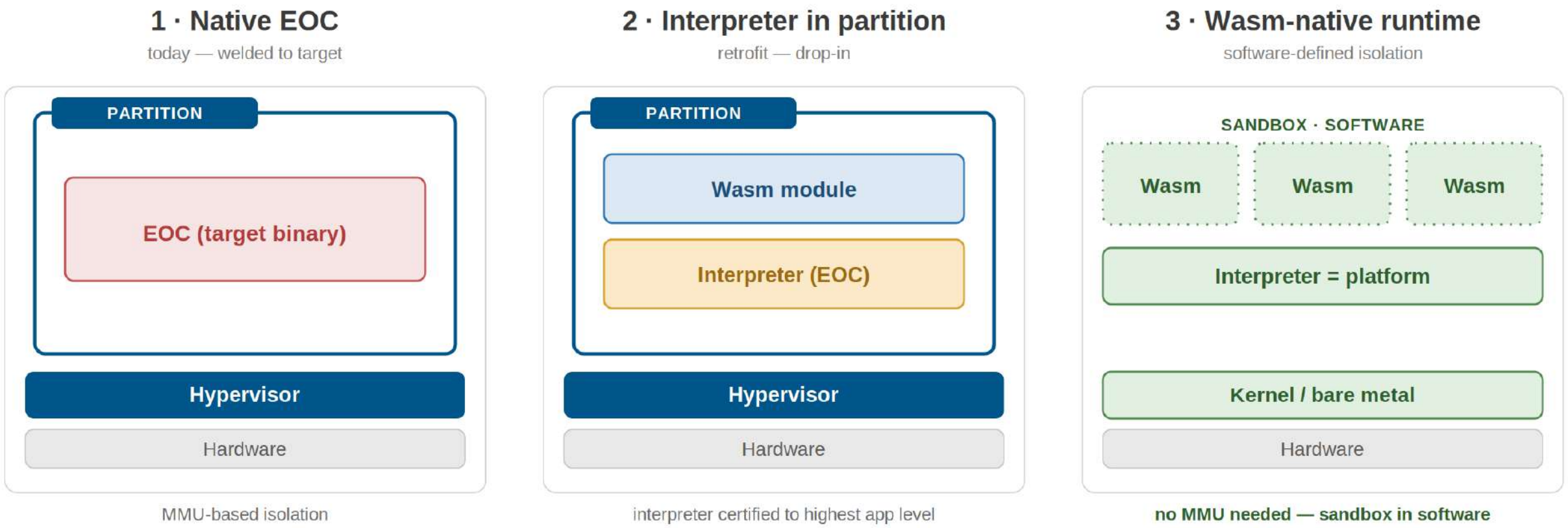
Wasm omits exactly the features that made its predecessors un-certifiable.

The recompilation tax

- ARINC 653 already promises you to reuse source across hypervisors. FentISS, SYSGO, Lynx, Wind River, Green Hills all expose the same API.
- But you still recompile for each target. New EOC means DO-178C compliance must be demonstrated again, even with unchanged source.
- The development cost is reused/reduced; the verification cost is not. That is where the budget goes.
- A universal byte-code, a common ABI that overarches ISAs, would let the EOC itself be reused. That is the goal.

ARINC 653 loosened the coupling to the OS. **Wasm targets the coupling to the ISA.**

Three ways to integrate Wasm



Pattern 2 retrofits onto any existing hypervisor. Pattern 3 is the interesting one for ECRTS: spatial isolation done in software, no MMU.

Can Wasm be Executable Object Code?

- DO-332 names interpreters as a virtualisation technique, and says code processed by them is [executable] code, not data.
- “Target computer” may be read as “target environment” = virtualisation software + target computer. Under that reading, Wasm bytecode is EOC.
- Consequence: certification credit earned on a host interpreter can transfer. The re-certification burden moves to the interpreter, once per architecture.

The open question: Is the certification evidence on the byte-code reusable across hardware? It remains as a question for the authority.

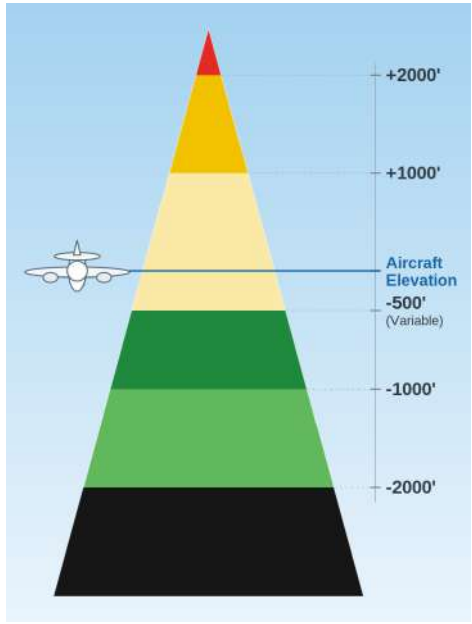
The cost of using Wasm — our avionics demonstrator, TAWS

~20× slower
interpreted vs. native, same TAWS algorithm

8.6× larger image
489 kB → 4.2 MB partition (interpreter ~1.5 MB)

2.2× bigger module
Wasm vs. native code, matching *Spies & Mock (2021)

±11.5 µs jitter
vs. ±7.5 µs native — same order of magnitude



For state machines and control laws this is tolerable; for heavy compute it is not.

*Spies, Mock — An Evaluation of WebAssembly in Non-Web Environments. CLEI 2021.

What shall a Wasm interpreter provide us?

1. **Certifiability**
   Certification effort scales super-linearly with code. Bounded/deterministic resource usage at run-time.
   ≤ 100 kLOC · no GC · no recursion

2. **Evidence-reuse**
   The Act 2 thesis made concrete: if the byte-code is the EOC, certification credit transfers across hardware.
   direct EOC · no AOT / JIT / IR

3. **Self-containment**
   No reliance on a host OS, portable to bare metal, and able to schedule and isolate applications itself.
   OS-independent · self-scheduling

4. **Observability**
   Coverage and health-monitoring without instrumenting the application. Evidence comes from the interpreter.
   instrumented per instruction

These four become the columns of the next slide and not one existing interpreter meets all of them.

No existing interpreter fits

Interpreter	LoC	<100 kLOC	Direct EOC	No GC	OS-indep.	No recursion
wasmtime	310k	!	!	✓	!	✓
wasmi	45k	✓	!	✓	✓	✓
wizard	26k	✓	✓	!	!	✓
reference	11k	✓	✓	!	✓	!
wasmer	193k	!	!	✓	!	✓
WAMR · classic	92k	✓	✓	✓	✓	!*
wasm3	10k	✓	!	✓	✓	!
WasmEdge	36k	!	!	✓	!	✓
WAVM	32k	!	!	✓	!	✓

✓ meets requirement ! fails it.

Even the closest fit fails on a specific, nameable point. Therefore, we built a new one.

DLR wasm-interpreter

- In-place interpretation, no IR rewrite. A side table (*Titzer, 2022) precomputes branch targets during validation.
- no_std Rust, two dependencies (log, libm). Bare-metal capable; passes the official spec test suite (stable features).
- Stack-less execution. Guest state lives on the heap, so a task can be preempted and resumed. This is what makes software scheduling possible.
- Fuel + resumable execution. Run until fuel is exhausted, return to the host, refill and resume. No recursion, deliberately for DO-178C.

Designed for certifiability first, performance second. It is the engine for everything in Act 3.

*Titzer — A Fast In-Place Interpreter for WebAssembly. OOPSLA 2022.

The fuel concept

- Each Wasm instruction is assigned a fuel cost. The interpreter is filled with fuel; execution consumes it; at zero, it traps.
- Accounting can be batched per basic block: blocks are branch-delimited, so the cost is known and the check happens once per block, at the branch.
- The check happens before the block runs. You handle an out-of-fuel condition before it occurs.
- Fuel is an artefact of the interpreter knowing the program state. No hardware tracing, no instrumentation of the EOC.

One mechanism, three jobs in this act: WCET abstraction, scheduling primitive, interference monitor.

Can fuel give a hardware-independent WCET bound?

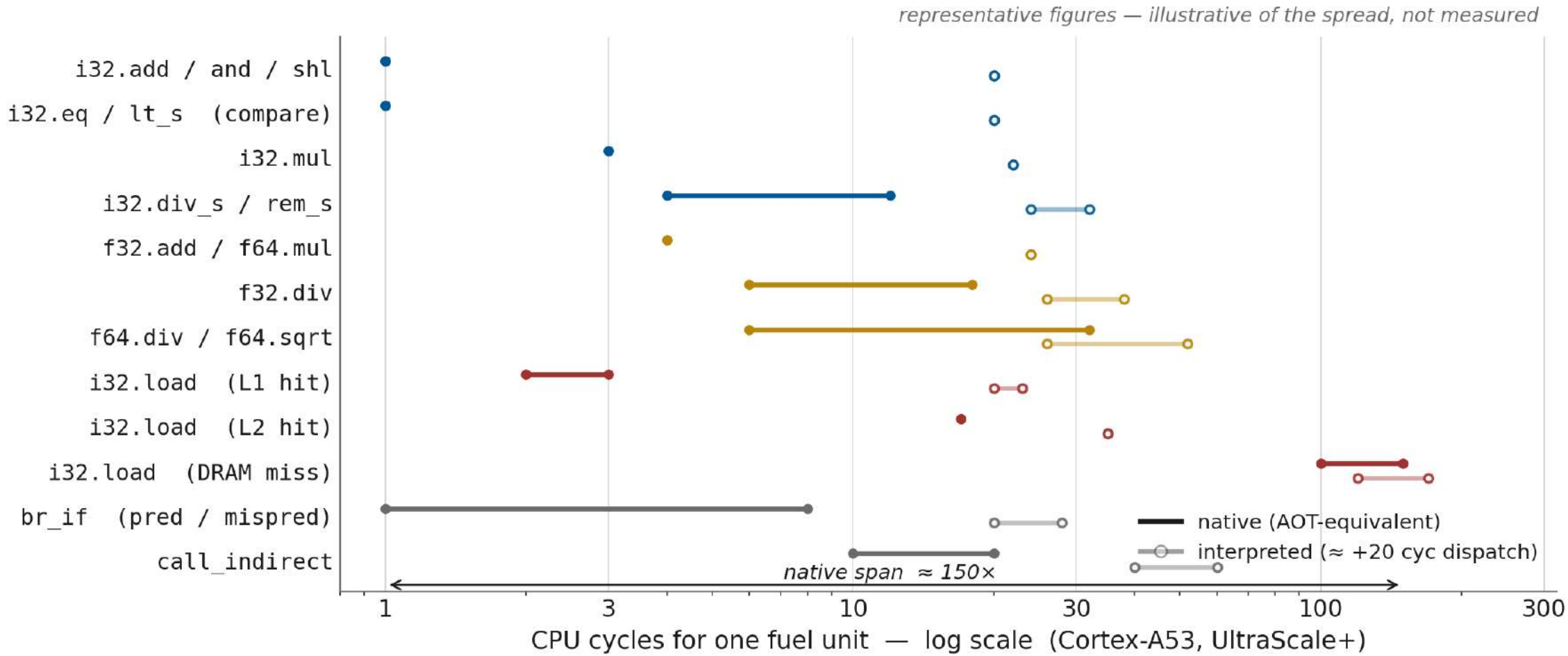
Pick the worst per-fuel operation on the target, then bound the algorithm:

$$o_s : \forall\ o \in \text{Wasm ops},\quad \text{WCET}(o) / \text{fuel}(o) \leq \text{WCET}(o_s) / \text{fuel}(o_s)$$

$$\textbf{WCET}(a) \leq \textbf{fuel}(a) \cdot \textbf{WCET}(o_s) / \textbf{fuel}(o_s)$$

- fuel(a) is measured on any development machine. It is deterministic and identical across hardware for given inputs.
- Our paper is explicit: this bound is very pessimistic. It can be tightened per basic block, and a violation is detected one block ahead.
- It needs no hardware tracing and no instrumentation of the EOC. The interpreter already knows the program state.

Why the bound is pessimistic — an illustration



One fuel unit buys wildly different amounts of time. The spread is dominated by the memory hierarchy.

Fuel measures work, not time and that is the point

time = work x rate

computational work

byte-code instructions = fuel(a)

HARDWARE-INDEPENDENT

Same module + same input
→ same fuel, on every CPU

Survives a hardware change untouched.

time per fuel

how fast this silicon runs a fuel unit

PER-TARGET

two ways to obtain it:

- static bound (DASC 2023)
  sound, pessimistic — for offline WCET
- live measurement dt/df
  sampled at runtime — for monitoring

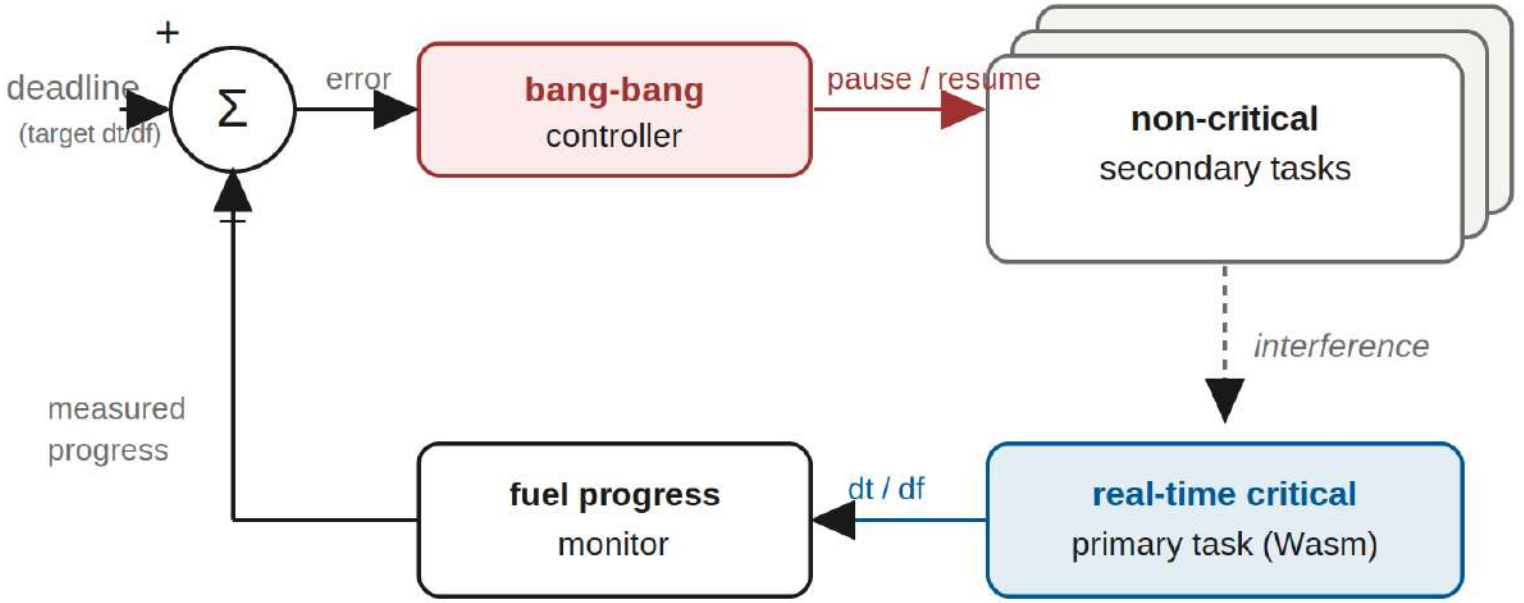
The work is portable; the timing variance is resolved into a per-target rate.
This is what lets us go from the offline bound to the live measurement for multicore targets.

Multicore interference, our open wound

Shared cache	one core evicts another's lines
Cache coherency	temporary unavailability of cache lines
Interconnect	congestion, unfair arbitration, possible starvation
Memory bus / controller	bandwidth saturation, row-buffer contention

Hundreds of mitigation techniques in the literature, nearly all channel-specific and static. On COTS hardware the channels are many, poorly documented, and partitioning costs performance.

Mitigate the effect, not the channel

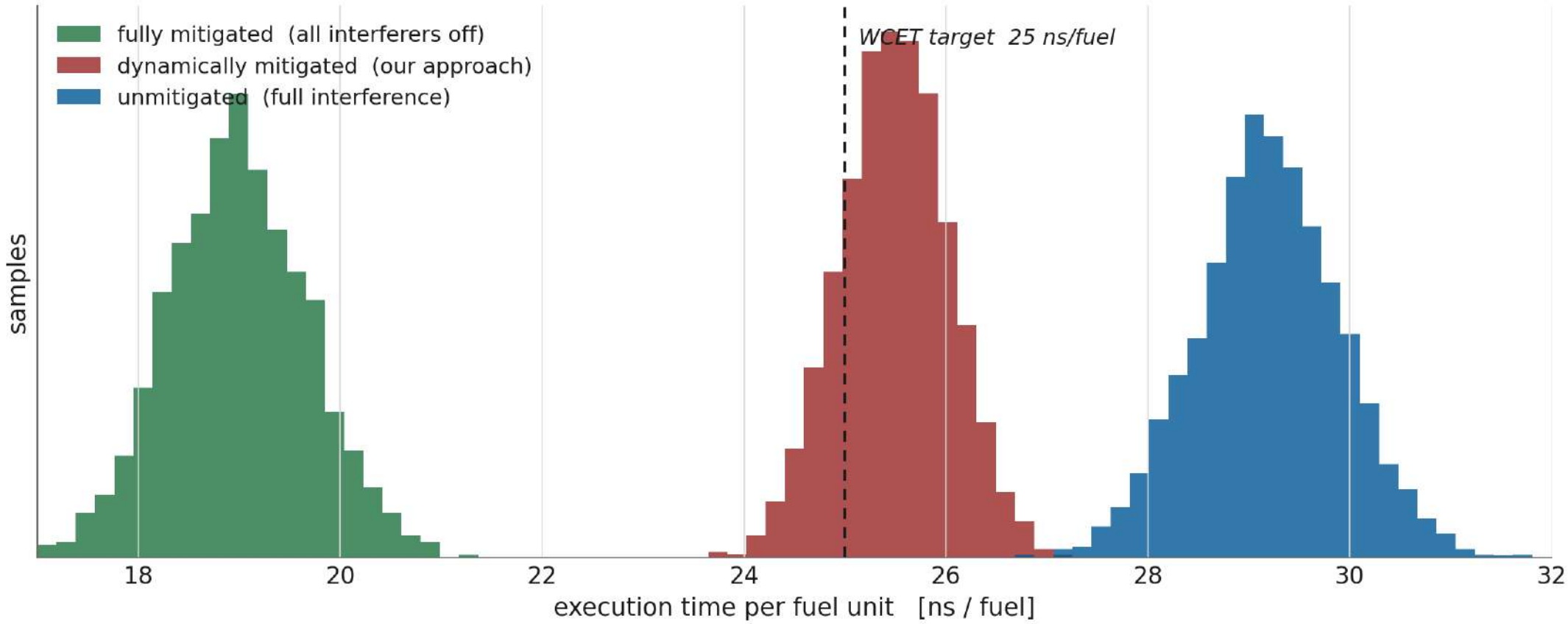


Mitigate the effect, not the channel.

One loop covers every interference source caused by software.

Fuel makes per-task progress observable. When time-per-fuel drifts above target, pause the secondary tasks. One loop, every known or unknown software-induced channel.

It seems working…



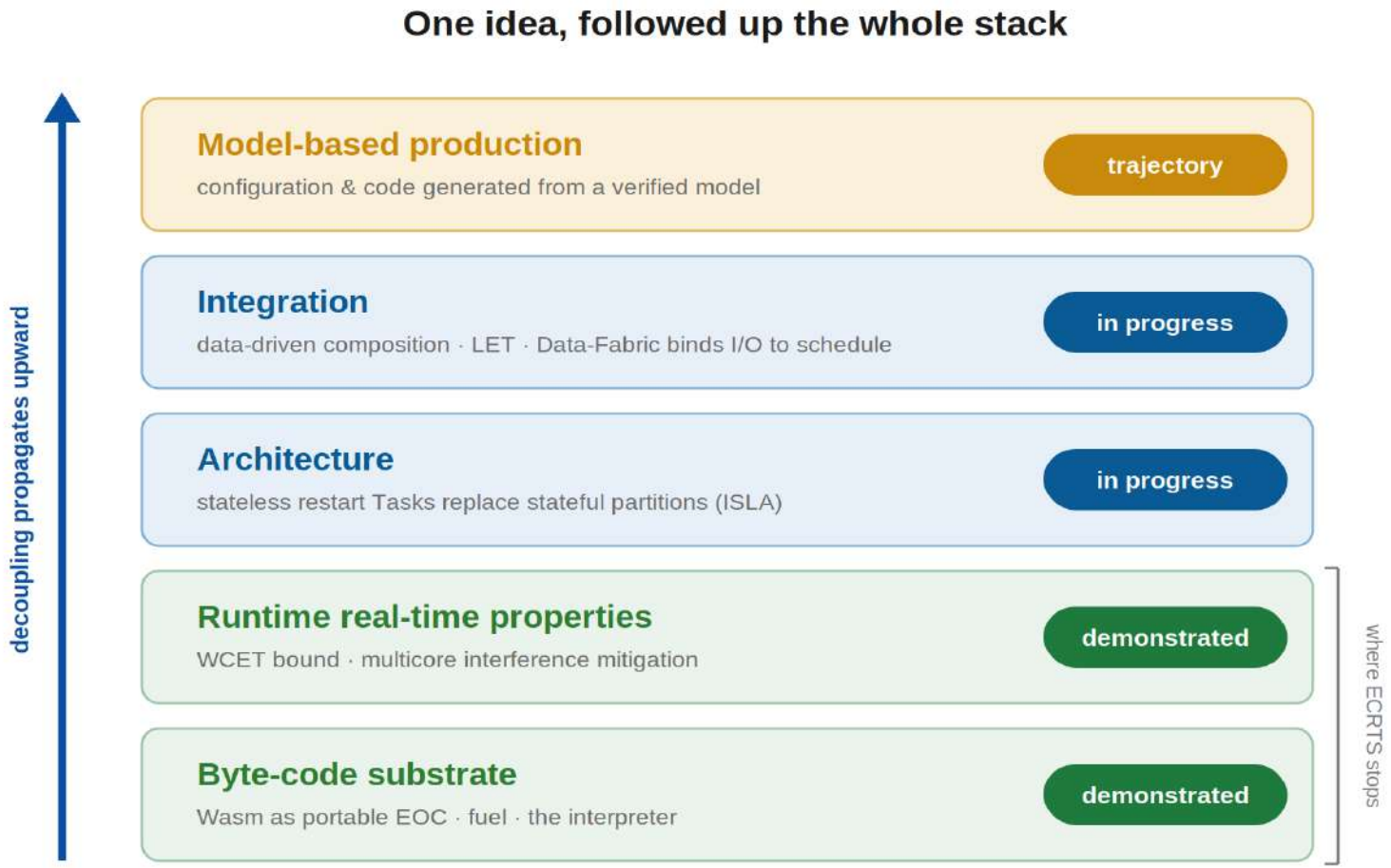
29 → 25.5 ns/fuel against a 25 ns/fuel target. Detection, mitigation, and convergence all demonstrated.

Where do these claims stop?

- Interpretation is slow, even an order of magnitude. AOT recovers it but reopens the certification gap; fuel survives AOT, the control loop needs re-tuning.
- Not every instruction is O(1): e.g memory.copy(n) scales with n. Per instruction fuel breaks here — the cost model must charge it proportionally.
- Only one critical task at a time. The mitigation throttles everyone else. Independent hardware interference (DMA from a NIC) is outside its reach.
- The demonstrator is on Linux with a worst-case StressNG adversary. Bare-metal porting and realistic workloads are needed before viability claims.

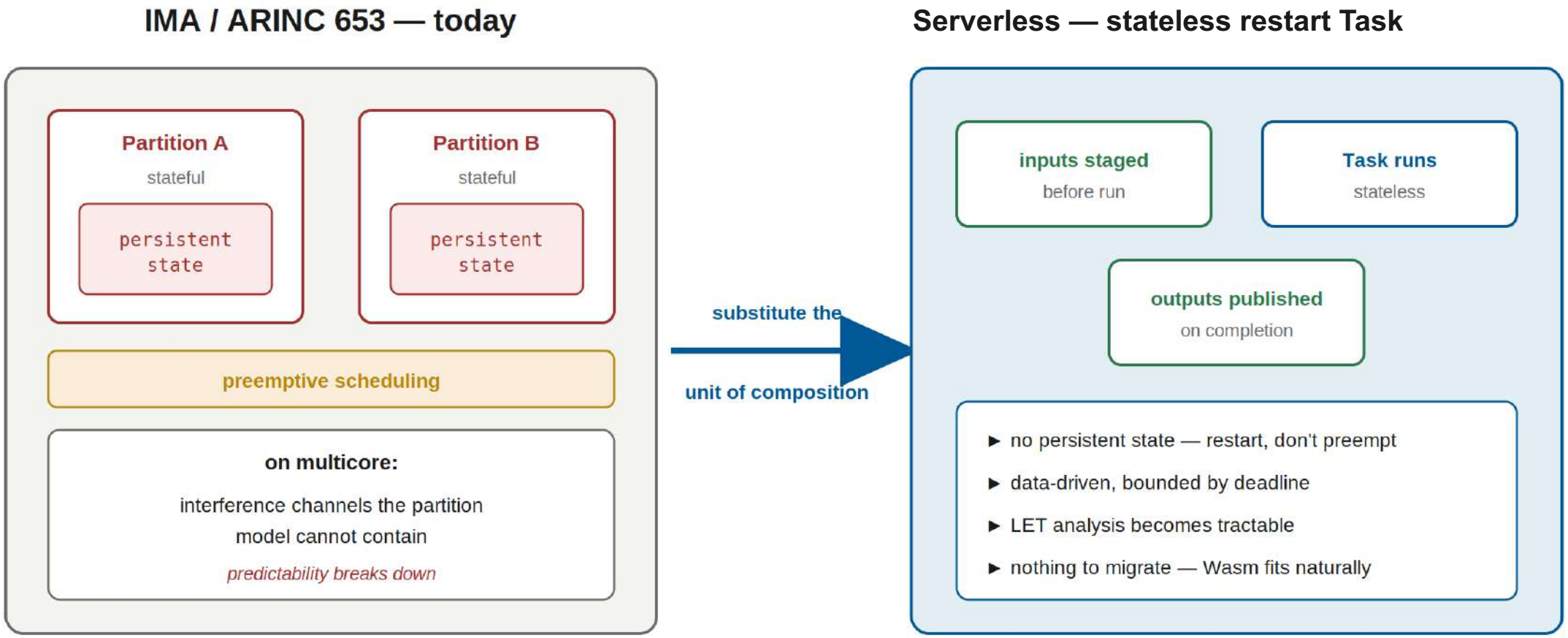
These are the boundaries of **early prototyping and experimentation**.

We don't stop at the runtime



Decouple software from hardware at the byte-code, and the consequence propagates up into architecture, integration, and how the system is produced.

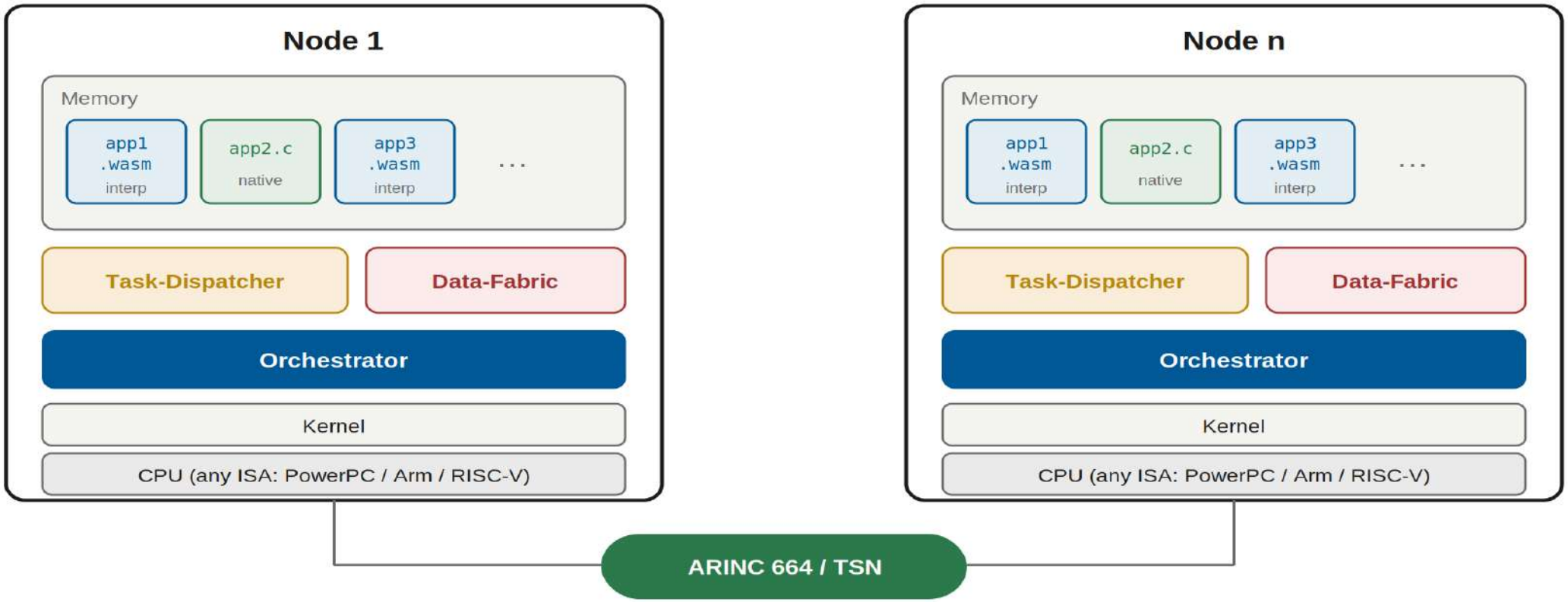
Architecture: partitions give way to stateless Tasks



The restart model is a different unit of composition. Stateless tasks make LET tractable, slack absorption coherent, and Wasm a natural fit.

Serverless: the architecture that substitution forces

Stateless Tasks · data-driven · one Orchestrator per node



A stateless Task carries no state between invocations, and that one property determines the rest for the architecture.

Integration becomes configuration of data-flow

- Tasks never touch the data store directly. The Data-Fabric stages inputs before a Task runs and collects outputs after it finishes. Implicit and safe communication.
- Inputs read at release, outputs visible at termination: this is Logical Execution Time (LET) (*Henzinger et al., Giotto), enforced at the runtime layer.
- Data-flow is bound to the static schedule. Queues are statically checked; integration is configured.
- Allocation + scheduling can be modelled as a Flexible Cyclic Jobshop Problem. It is solved offline. And interference is answered twice: opportunistic slack (architecture) and the fuel loop (runtime).

Statelessness changes how tasks compose, as well as how they run.

*Henzinger, Kirsch et al. — Giotto / the Logical Execution Time paradigm.

… and where the stack is going

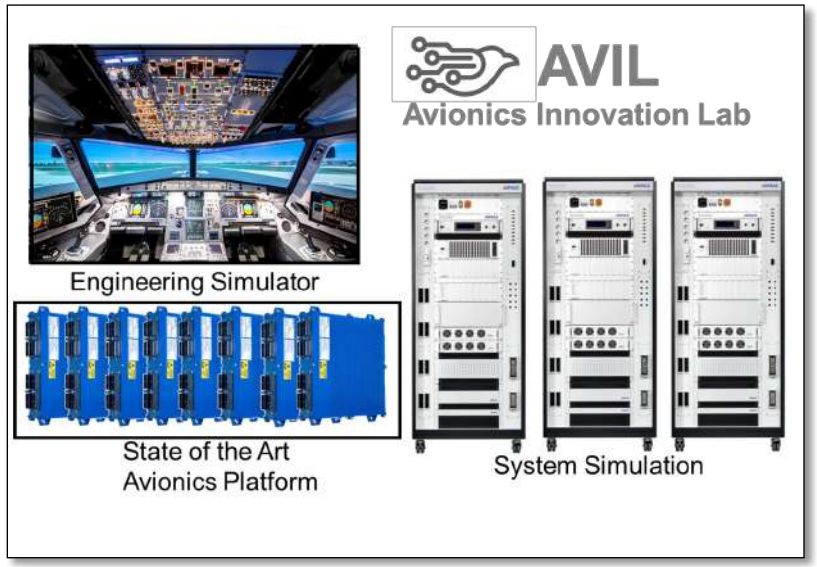
Trajectory. Architecture and integration are now structured: stateless tasks, explicit data-flow, static schedule. Because the structure is explicit, it can be modeled and the model can generate the configuration..

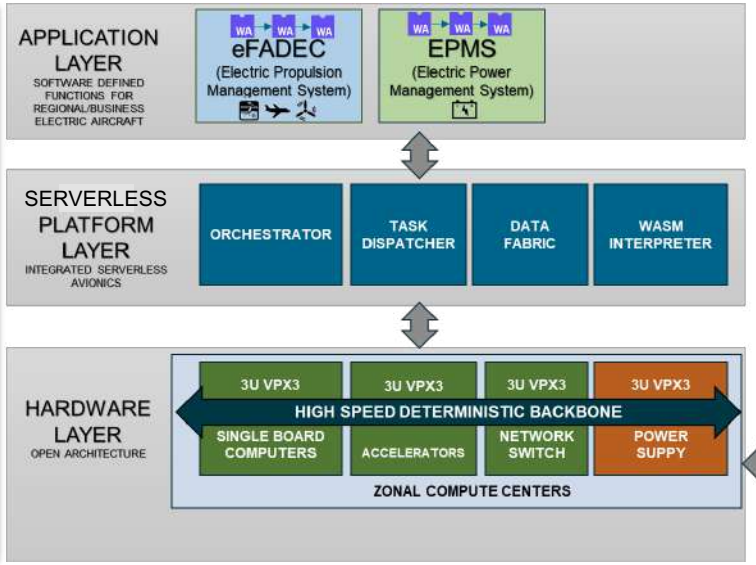
- Configuration of integration generated from a verified model. Not hand-built or hand-checked.
- Application code generated to the same byte-code substrate, deployable on any target.
- Verify once, at the model: the qualification burden concentrates where it is cheapest.

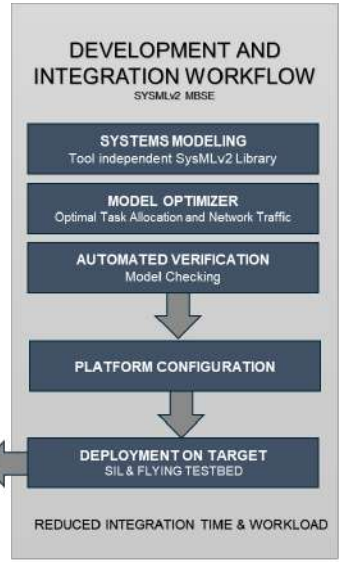
We are building toward this. The runtime is demonstrated, the upper stack is in progress. The serverless work lays the foundation; full validation is ahead.

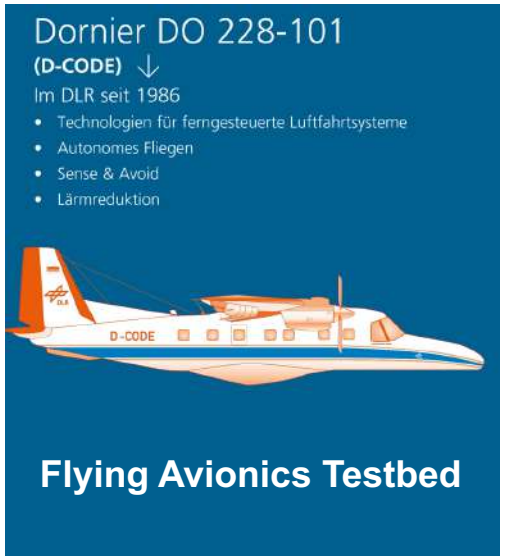
Validating the stack: from lab to flight

One stack· two validation targets: AVIL and the Flying Avionics Testbed









The stack will be validated first in the AVIL with a representative use-case, then flown on the DO 228 Flying Avionics Testbed.

A research agenda for this room

1	**A qualified Wasm, JIT/AOT compilers**	the CompCert-shaped hole; without it JIT/AOT is locked out of DAL-A/B
2	**Tighter fuel-based WCET**	per-path, cache- and pipeline-aware, at the byte-code level
3	**A statistical model of dynamic mitigation**	to support soft-real-time certification arguments
4	**Formal verification of an avionics interpreter**	the seL4 lineage, applied to Wasm
5	**Mixed-criticality without partitions**	can fuel + sandbox replace temporal/spatial partitioning?

We follow the byte-code further up through the architecture it forces and the integration model it enables, toward a development stack that generates the system from a verified model.

From sandbox, to flight deck, to the model that builds it.

Thank you

Questions · discussion · collaboration welcome

Key references

Zaeske, Friedrich, Schubert, Durak — WebAssembly in Avionics: Decoupling Software from Hardware. DASC 2023.

Zaeske, Önem, Hartung, Durak — On the Design of a WebAssembly Interpreter for Safety-Critical Avionics. DASC 2025.

Lukić et al. — Serverless Avionics: A New Architectural Approach (ISLA). AIAA SciTech 2026.

Meier, Zaeske, Durak — Dynamic Mitigation of Multi-Core Interference in Safety-Critical Systems. SE 2026.

DLR-FT/wasm-interpreter · github.com/DLR-FT