Journal homepage: https://itce.vn.ua/en

Information Technologies and Computer Engineering **Vol. 23, No. 2. 2026**

Article's History: Received: 02.01.2026 Revised: 13.05.2026
Accepted: 25.06.2026 Published: 03.08.2026

ISSN 1999-9941; e-ISSN 2078-6387
UDC 004.75
DOI: 10.31649/vitce/2.2026.121

Parallelism and WebAssembly memory management in microinterface architecture

Oleksandr Stepanov*
Postgraduate Student
Lviv Polytechnic National University
79000, 12 S. Bandera Str., Lviv, Ukraine
https://orcid.org/0009-0004-2068-6318

Halyna Klym
Doctor of Technical Sciences, Professor
Lviv Polytechnic National University
79000, 12 S. Bandera Str., Lviv, Ukraine
https://orcid.org/0000-0001-9927-0649

Abstract. The decentralisation of monolithic front-ends into micro-front-end architectures, combined with WebAssembly (Wasm), imposes limitations on the communication overhead associated with 'bridge crossing' between JavaScript and Wasm's linear memory, an area that remains under-researched in quantitative terms. The aim of the study was to quantitatively assess the overhead of bridge crossing, analyse the risks of race conditions in shared-memory micro-frontend environments, and develop and validate a hybrid communication model that eliminates serialisation bottlenecks. A controlled micro-benchmark methodology was employed for the study, utilising the browser API performance.now() with a measurement precision of no less than 5 microseconds. Each scenario was run over 10,000 measurement iterations following a mandatory JIT compiler warm-up phase of 1,000 iterations. Two architecturally distinct data transfer strategies were compared under identical hardware conditions using Float32Array arrays ranging in size from 0.1 MB to 10 MB at a rendering rate of 60 frames per second. It has been empirically established that the transfer of primitive data types takes 50-100 nanoseconds, whereas copying large data arrays (e.g., 1 MB) takes 1-3 milliseconds – which, at a frame rate of 60 frames per second, consumes 11.1% of available CPU time solely on data transfer, completely negating Wasm's performance advantages. The proposed hybrid model, which combines a lightweight event bus for transmitting control signals with a SharedArrayBuffer and Atomics-based synchronisation for data access, achieves a transaction time of 250 nanoseconds regardless of payload size – corresponding to a speed-up factor of up to 66,000 times for 10 MB arrays compared to classical event-driven communication. The results provided a quantitatively validated architectural foundation for designing high-performance loosely coupled micro-frontend systems capable of performing AI/ML inference, 3D rendering and video streaming directly within the browser without architectural compromises

Keywords: Wasm technology; micro-frontends; SharedArrayBuffer; data transfer strategies; event bus; Atomics; overhead

Introduction

The evolution of modern information systems and web applications has necessitated the processing of significant data volumes directly on the client side. Tasks that were previously performed exclusively on servers or in specialised desktop applications – such as real-time video processing, physical process simulation, complex 3D graphics rendering, and the application of machine learning models – are today being integrated into the web environment. The decentralisation of monolithic frontends into microfrontend architectures offers significant advantages in team

Suggested Citation:
Stepanov, O., & Klym, H. (2026). Parallelism and WebAssembly memory management in microinterface architecture. *Information Technologies and Computer Engineering*, 23(2), 121-131. doi: 10.31649/vitce/2.2026.121.

*Corresponding author (*oleksandr.v.stepanov@lpnu.ua*)

autonomy and scalability, yet introduces fundamental architectural constraints when combined with WebAssembly (Wasm) technology. The primary constraint is the communication overhead associated with crossing the environment boundary ("bridge crossing") between JavaScript and Wasm's linear memory.

N. Savani (2023) characterises this shift as a fundamental transformation in web development paradigms, driven by user expectations for desktop-class interactivity within browser environments. To manage the growing complexity of such applications, the industry is actively transitioning from monolithic architectures toward micro-interfaces (microfrontends). D. Taibi & L. Mezzalira (2022) define this architectural approach as the decomposition of large user interfaces into independent, autonomously deployable modules developed by separate teams, and identify communication between modules as its primary unsolved engineering challenge.

However, despite the obvious organisational advantages, the implementation of resource-intensive tasks within microfrontend environments encounters fundamental limitations of the JavaScript language. Due to its dynamic typing and reliance on automatic garbage collection (GC), JavaScript introduces unpredictable latency and is unable to guarantee stable high performance for CPU-intensive computations. M. Glinka *et al*. (2024) provide a detailed examination of current trends and practical implementation challenges of microfrontends, with particular emphasis on the complexities of state management and inter-module communication. Research by O. Stepanov & H. Klym (2024a) also demonstrated that tools such as wasm-bindgen significantly facilitate development, yet introduce considerable latency when transferring complex data structures. J.W. Sunarto *et al*. (2023) conducted a systematic review of WebAssembly versus JavaScript performance, confirming that Wasm consistently outperforms JavaScript for compute-intensive workloads while exhibiting lower energy consumption in lightweight application scenarios. WebAssembly (Wasm) technology was created to overcome this barrier, offering compilation of native code (C++, Rust) into a binary format that executes in the browser at near-native speed. Nevertheless, the deep integration of WebAssembly into distributed microfrontend architecture introduces additional architectural challenges: high communication overhead between the isolated JavaScript and Wasm environments – the so-called "bridge crossing" problem. When microfrontends rely on frequent copying of large data arrays to communicate with Wasm modules, the costs of serialisation and deserialisation completely negate the computational speed advantage. Consequently, there is an urgent need for efficient parallelism mechanisms and hybrid communication strategies that ensure reliable and fast data exchange without violating the autonomy principles of microfrontends.

The aim of this article was to investigate parallelism mechanisms in JavaScript and WebAssembly environments and to develop a hybrid communication model for micro-interfaces. In accordance with this aim, the following research objectives are defined:

1) to quantitatively analyse the overhead costs ("bridge crossing") of data exchange between JavaScript and Wasm environments for different data types;

2) to investigate the race condition problem when using SharedArrayBuffer in the distributed environment of microfrontends;

3) to design and experimentally validate a hybrid communication model that combines a lightweight Event Bus for the transmission of control signals with shared memory and atomic locking mechanisms (Atomics) for the transmission of large data arrays.

Literature Review

Analysis of contemporary scientific literature reveals significant interest in optimising microfrontend architectures and applying WebAssembly. M. Glinka *et al*. (2024) provide a detailed examination of current trends and practical implementation challenges of microfrontends, with particular emphasis on the complexities of state management and inter-module communication. V. Marco & K. Farias (2024) conducted a systematic mapping of technologies and architectures used to develop microfrontend applications, identifying Module Federation and Web Components as the dominant integration strategies, and establishing that runtime composition introduces measurable communication latency not present in build-time integration approaches. The systematic mapping reported by V. Marco & K. Farias (2024), together with the practical implementation experience documented by J. Männistö *et al*. (2023), converge on the conclusion that an inappropriately chosen inter-module communication strategy in microfrontend systems leads to tight runtime coupling and the formation of a "distributed monolith", in which the architectural benefits of independent deployability are nullified by hidden synchronisation dependencies between supposedly autonomous modules.

The principles and common pitfalls of microfrontend architecture are comprehensively documented by D. Taibi & L. Mezzalira (2022), who identify shared state management and cross-module event propagation as the two most frequent sources of architectural degradation in production systems. J. Männistö *et al*. (2023) reported practical experience implementing a frameworkless microfrontend architecture in a small organisation, demonstrating that even minimal framework dependencies introduce hidden communication overhead that compounds with system scale. S.K. Banoth (2024) analysed Module Federation and Native Federation patterns in Angular applications, quantifying that Native Federation reduces inter-module communication latency by approximately 23% compared to Webpack-based Module Federation due to reduced serialisation depth. N. Kaushik *et al*. (2024) proposed a machine-learning-based framework for microfrontend performance prediction, demonstrating that communication

overhead between microfrontend modules and backend microservices is the dominant variable in end-to-end response time models. T. Gaur (2024) examined state management techniques for microfrontend systems, concluding that centralised state stores introduce synchronisation bottlenecks that scale superlinearly with the number of independent modules.

Regarding WebAssembly integration, Z. Liu *et al*. (2024) investigated the application of Wasm for artificial intelligence tasks in composable web architectures, emphasising the importance of minimising inter-environment interaction overhead. Prior research by O. Stepanov & H. Klym (2024a) also demonstrated that tools such as wasm-bindgen significantly facilitate development, yet introduce considerable latency when transferring complex data structures. L. Zhang *et al*. (2024) proposed approaches to optimising microfrontend performance; however, the growing number of isolated modules intensifies questions of concurrent access to shared resources. In particular, the question of efficient synchronisation and the avoidance of race conditions when using shared memory (SharedArrayBuffer) in event-driven architectures remains insufficiently studied and requires dedicated investigation.

K.I.D. Kyriakou & N.D. Tselikas (2023) demonstrated that Rust-based WebAssembly implementations outperform equivalent JavaScript solutions by a factor of two to four in browser environments, providing additional motivation for investigating the communication overhead that arises when integrating such high-performance modules into microfrontend systems. D. Poltavskyi (2025) investigated WebAssembly integration in performance-critical web applications, confirming that the primary bottleneck in production Wasm deployments is not computation but inter-environment data transfer, a finding that directly motivates the present study. P.P. Ray (2023) provided a comprehensive overview of WebAssembly for IoT applications, establishing that Wasm's near-native execution speed is consistently negated by serialisation overhead when interfacing with JavaScript host environments – a pattern directly analogous to the microfrontend bridge crossing problem investigated herein. M.N. Hoque & K.A. Harras (2023) examined WebAssembly for edge computing, demonstrating that Wasm's portability and sandboxing properties make it suitable for distributed execution environments, but that memory isolation between Wasm instances creates the same inter-boundary copying costs observed in microfrontend deployments. G. Perrone & S.P. Romano (2025) conducted a comprehensive security review of WebAssembly, establishing that the linear memory model and sandbox isolation – the same properties that create bridge crossing overhead – are also the foundation of Wasm's security guarantees, confirming that eliminating copying through SharedArrayBuffer requires explicit security mitigations such as COOP/COEP headers. J. Kim *et al*. (2024) demonstrated hardware-level WebAssembly acceleration on embedded systems, showing that architectural co-design of the host environment and Wasm runtime can reduce inter-boundary communication costs by up to 60% – a principle that informs the hybrid model's pre-allocation strategy. C. Watt (2025) provides a formal treatment of the WebAssembly concurrency model, establishing the theoretical foundation for the atomic memory access mechanisms employed in this study. Z. Yang *et al*. (2024) conducted a comprehensive survey of WebAssembly runtime research, identifying memory management and inter-environment communication as the two primary open challenges in production Wasm deployments – precisely the challenges addressed by the hybrid model proposed herein.

The use of WebAssembly for computationally intensive scientific computing tasks is illustrated by C. Erazo Ramirez *et al*. (2024), who developed HydroCompute – a browser-based high-performance computational library for hydrology that leverages WebAssembly, WebGPU, and Web Workers for parallel client-side computation. Their work demonstrates that multi-technology parallelism architectures in the browser can achieve performance sufficient for real-world scientific simulation, but also highlights that inter-component data transfer between WebAssembly and JavaScript contexts remains a critical design constraint – a finding that corroborates the motivation for the present study. H. Nam & E.-J. Im (2025) demonstrated that combining WebAssembly with multithreading strategies achieves near-native performance for physics simulations in the browser, reporting that SharedArrayBuffer-based data sharing between worker threads and the main thread eliminates the serialisation bottleneck that otherwise limits simulation throughput – providing direct empirical precedent for the architectural approach proposed in this paper.

The reviewed body of literature converges on three interrelated conclusions that jointly motivate the present study. First, regardless of the specific microfrontend integration technology adopted (Module Federation, Native Federation, Web Components, or frameworkless solutions), inter-module communication overhead has been consistently identified as the dominant performance and architectural bottleneck. Second, although WebAssembly delivers near-native computational performance, its practical benefit in microfrontend deployments is systematically undermined by the cost of crossing the JavaScript-Wasm boundary, since the linear memory model and sandbox isolation that ensure security simultaneously enforce expensive serialisation when data are exchanged across module boundaries. Third, although SharedArrayBuffer-based zero-copy mechanisms have been shown to eliminate this serialisation cost in adjacent domains such as scientific computing and physics simulation, no prior study has empirically quantified the resulting performance gain specifically in a microfrontend architecture nor has it provided an explicit mechanism for resolving the race-condition risk that arises from concurrent shared-memory access. Consequently, the open question that remains insufficiently addressed in the existing literature concerns the design of a hybrid communication model that combines the loose coupling of an event-driven Event Bus with the zero-copy efficiency

of SharedArrayBuffer and the correctness guarantees of Atomics-based synchronisation, together with a quantitative experimental evaluation of its acceleration coefficient and CPU-load impact under realistic 60-FPS workloads. The present paper formulates such a model and supplies the empirical evidence that the prior literature has identified as missing.

Materials and Methods

The chosen research method is a controlled comparative microbenchmark experiment, in which two communication strategies between WebAssembly-integrated microfrontend modules – the classical event-driven approach with serialisation (Scenario A) and the proposed hybrid Event Bus + SharedArrayBuffer + Atomics model (Scenario B) – are evaluated under identical hardware, browser and workload conditions. This method is justified by the fact that an empirical comparison performed under a strictly controlled environment is the only approach capable of isolating the architectural effect of the communication strategy from incidental factors such as hardware variance, JIT compilation noise or operating-system interference, thereby providing the quantitative evidence that, as established in the literature review, has been missing from prior qualitative discussions of microfrontend communication overhead. This study employs a quantitative experimental methodology based on controlled microbenchmark testing. The primary objective of the experimental design was to isolate and measure the communication overhead between JavaScript and WebAssembly environments, and to compare two architecturally distinct data transmission strategies under identical hardware and software conditions. Two experimental scenarios were designed to serve as direct methodological counterparts: Scenario A represents the classical event-driven approach with data serialisation and copying, while Scenario B represents the proposed hybrid communication model utilising shared memory with atomic synchronisation. The controlled comparison of these two scenarios under identical workloads constitutes the core methodological contribution of this research.

The selection of data payload sizes (0.1 MB, 0.5 MB, 1 MB, 5 MB, and 10 MB Float32Array) was motivated by real-world use cases in browser-based applications. A 0.1-1 MB range corresponds to typical machine learning inference input tensors and compressed video frame buffers, while the 5-10 MB range represents uncompressed video frames (a 1920×1080 pixel frame in 32-bit RGBA format occupies approximately 8.3 MB) and large numerical simulation datasets. The 60 frames-per-second (FPS) load scenario was selected as the industry-standard rendering cadence for interactive real-time applications, representing the most demanding continuous workload for browser-based data pipelines. To ensure reproducibility and scientific validity, all experiments were conducted on a standardised hardware and software testbed. The full specification of the experimental environment is presented in Table 1.

Table 1. Hardware and software testbed specification

Component	Specification
Processor	Apple M3 Pro, 12 cores (6P + 6E), 4.05 GHz
RAM	36 GB LPDDR5, 150 GB/s bandwidth
Operating System	macOS Sequoia 15.3.1
Browser	Google Chrome 124.0.6367.82 (V8 Engine 12.4)
Wasm Runtime	Wasmtime 20.0.0 (isolated tests)
Compiler / Toolchain	rustc 1.77.2 (wasm32) / wasm-bindgen 0.2.92

Source: developed by the authors

The choice of Rust as the source language for WebAssembly compilation was motivated by three factors: first, Rust's ownership model guarantees memory safety without a garbage collector, eliminating GC-induced latency pauses during benchmarking – a property formally analysed by A.E. Michael *et al*. (2023), who demonstrated that memory-safe WebAssembly execution through MSWasm enforcement adds negligible overhead when the source language already provides compile-time safety guarantees; second, the wasm-bindgen toolchain provides well-documented and stable JavaScript-to-Wasm FFI bindings, enabling precise measurement of bridge crossing overhead in isolation; third, Rust's predictable zero-cost abstractions ensure that measured latencies reflect genuine architectural overhead rather than language-level inefficiencies. Google Chrome 124 was selected as the primary test environment due to its market dominance (approximately 65% global browser market share as of Q1 2024) and the maturity of its V8 JavaScript engine's WebAssembly JIT compiler.

The experimental test environment consisted of two independent microfrontend modules deployed within a single browser context using a Module Federation pattern. Each microfrontend module was implemented as an isolated JavaScript bundle with its own Wasm module instance, simulating a realistic production deployment where separate development teams own independent application segments. The two modules communicated exclusively through the designated communication channel under test – either the classical Event Bus (Scenario A) or the hybrid model (Scenario B) – with no direct function calls permitted between them, preserving the architectural autonomy constraint that defines the microfrontend paradigm.

In Scenario A, the Event Bus was implemented using the browser's native CustomEvent API with structured cloning for data serialisation. Each transaction involved full

serialisation of the Float32Array payload into a transferable format, dispatch through the event system, and deserialisation at the receiving microfrontend. In Scenario B, a pre-allocated SharedArrayBuffer of 16 MB was established at application initialisation and shared between both microfrontend instances through the browser's cross-origin isolation context (COOP/COEP headers). The Event Bus in Scenario B transmitted only a lightweight control message containing the memory offset (4 bytes) and data length (4 bytes) as primitive integer values.

The primary timing instrument was the browser's performance.now() API, which provides sub-millisecond resolution with a precision floor of approximately 5 microseconds under Chrome's reduced timer resolution policy (applied as a Spectre mitigation). Each experimental scenario was executed for 10,000 measurement iterations following a mandatory warm-up phase of 1,000 iterations. The warm-up phase is critical for ensuring that the V8 engine's JIT compiler has fully optimised the hot execution paths before measurements begin; the ratio of cold-start to warm execution time (T_cold/T_warm) was observed to range from 3× to 15× depending on the operation type, confirming the necessity of this warm-up protocol.

Raw timing data from all 10,000 iterations was collected into Float64Array buffers for post-processing. Statistical analysis was performed to derive mean execution time, standard deviation (σ), and 95% confidence intervals for each measured component. The aggregate measurement uncertainty varied by operational range: for sub-100 ns operations (primitive transfers, SAB access), uncertainty reached ±20-25% due to the timer resolution floor; for the 100 ns – 10 µs range (hybrid model transactions), uncertainty was ±15-20% due to JIT compilation variance; for operations exceeding 1 ms (large array copies), garbage collection pauses dominated variability at ±5-8%. As the experimental results demonstrate, the acceleration coefficients observed are of several orders of magnitude, rendering these measurement uncertainties statistically inconsequential for the primary comparative conclusions.

Data processing was carried out in two stages. At the first stage, the raw timing series obtained from each scenario were filtered to exclude the warm-up phase (the first 1,000 iterations) and outliers exceeding three standard deviations from the local mean, which were attributed to operating-system-level interrupts and garbage collection events unrelated to the measured operation. At the second stage, descriptive statistics (arithmetic mean, standard deviation, 95% confidence interval) were computed for each tested data payload size, and the throughput, acceleration coefficient and CPU load share metrics were derived from these aggregates according to formulas (1)-(4) presented in the Results section.

The two scenarios were compared on a unified set of criteria, namely: (i) total transaction time T_total under identical payload sizes; (ii) decomposition of T_total into its constituent cost components (serialisation, memory allocation, data copying, event dispatch, SAB access, atomic locking); (iii) the algorithmic complexity class of transaction time as a function of payload size (O(n) versus O(1)); (iv) the share of CPU time consumed at the industry-standard 60 FPS rendering cadence; and (v) the architectural property of race-condition resolution. Combined application of the listed criteria provides a multi-dimensional methodological basis for evaluating the proposed hybrid communication model relative to the classical event-driven baseline and ensures that the comparative conclusions are not contingent on a single performance metric.

Results and Discussion

Interaction between the JS and Wasm environments requires function calls through the foreign function interface (FFI) and explicit data copying between the JavaScript heap and Wasm's linear memory. Analysis of microbenchmarks using Rust and wasm-bindgen demonstrates a critical dependency of performance on data type. Primitive value transfers (integers, floats) require 50–100 ns, while copying a 1 MB array requires 1-3 ms. If the architecture demands copying 1 MB at 60 frames per second, communication overhead alone consumes up to 180 ms per second, rendering such an approach entirely impractical for high-load scenarios.

These findings are consistent with results reported by Z. Liu *et al*. (2024), who measured bridge crossing latencies of approximately 2 ms for 1 MB payloads in composable WebAssembly architectures for AI inference tasks. However, Z. Liu *et al*. (2024) did not propose a mechanism to eliminate this overhead – their study concluded that minimising call frequency was the primary mitigation strategy. The research extends this finding by demonstrating that call frequency reduction alone is insufficient for 60 FPS workloads and that architectural restructuring through shared memory is the only viable solution. Similarly, M. Glinka *et al*. (2024) identified data serialisation as a primary source of tight coupling in microfrontend systems, characterising it as a structural anti-pattern rather than a performance concern. Authors' quantitative measurements provide the empirical foundation that was absent from their qualitative analysis, establishing concrete latency thresholds above which serialisation-based communication becomes architecturally unacceptable.

The use of SharedArrayBuffer (SAB) eliminates copying overhead, reducing access latency to approximately 15 ns – a reduction of two orders of magnitude compared to array copying. However, in microfrontend environments this introduces the critical concurrency problem of race conditions when multiple independent components access shared memory asynchronously. The application of the Atomics object for atomic thread locking (mutex implementation) is required for correct synchronisation. G. Perrone & S.P. Romano (2025) confirm that SharedArrayBuffer's reintroduction following the Spectre disclosure – contingent on cross-origin isolation – represents a deliberate security trade-off: the COOP/COEP deployment requirement constrains the attack surface while preserving the zero-copy performance benefit, a constraint that

the hybrid model explicitly accommodates through its initialisation protocol.

The practical experience reported by J. Männistö *et al*. (2023) on implementing a microfrontend architecture in an industrial setting indicates that the predominant inter-module communication strategy in production systems remains an event-driven messaging model devoid of shared state, primarily due to its implementation simplicity and the absence of race-condition risks inherent in concurrent shared-memory access. Authors' results expose the performance cost of this conservative approach in quantitative terms: the 1.85 ms per-transaction latency measured in Scenario A is directly attributable to the structured-cloning serialisation overhead that the prior literature identified qualitatively as a source of performance degradation in data-intensive microfrontends, but for which no quantified solution had been proposed. H. Nam & E.-J. Im (2025) demonstrated in the physics simulation domain that Atomics-based synchronisation between SharedArrayBuffer-accessing threads introduces a fixed overhead of 150-200 ns regardless of data volume – a measurement that directly corroborates the 175 ns Atomics cost measured in Scenario B of the present study, confirming the cross-domain generalisability of this result. The Atomics-based synchronisation mechanism proposed in authors' hybrid model directly addresses the race condition concern that prevents industry adoption of shared memory in microfrontend systems.

The proposed hybrid model separates data transmission into three interacting layers. The Event Bus transmits only lightweight control signals (offset pointer and length as primitive integers), without touching the data payload itself. Shared Memory stores data arrays in a pre-allocated SAB as the sole shared data store. Atomics guarantees atomicity of read and write operations, preventing race conditions during concurrent access by multiple microfrontends. This architecture achieves complete decoupling between the control and data buses while preserving the autonomous boundary of each microfrontend module.

L. Zhang *et al*. (2024) proposed a performance optimisation framework for microfrontend applications based on lazy loading, bundle splitting, and caching strategies, reporting load time reductions of up to 40%. While their approach addresses initialisation performance, it does not consider runtime communication overhead – the dominant bottleneck in CPU-intensive scenarios. Authors' hybrid model is complementary to their approach: L. Zhang *et al*.'s (2024) techniques optimise the delivery of microfrontend modules, while authors' model optimises the ongoing data exchange between them once deployed. O. Stepanov & H. Klym (2024b) established the foundational methodology for microfrontend information system implementation and identified communication as the primary unresolved challenge; the hybrid model proposed in the present study constitutes a direct and quantitatively validated solution to that identified gap. Two data exchange scenarios were compared using a 1 MB Float32Array transmitted between two microfrontends: Scenario A – classical Event Bus with serialisation and data copying – and Scenario B – the hybrid model (Event Bus + SAB + Atomics). The detailed transaction time decomposition is presented in Table 2.

Table 2. Transaction time decomposition

Cost Component	Scenario A (Event Bus + Copy)	Scenario B (Hybrid Model)
Serialisation & Deserialisation	500,000 ± 8,200 ns	0 ns
Memory Allocation	800,000 ± 11,400 ns	0 ns
Data Copying	550,000 ± 7,800 ns	0 ns
Event Bus Dispatch	75 ± 12 ns	60 ± 9 ns
SAB Memory Access	–	15 ± 3 ns
Atomics (lock/unlock)	–	175 ± 22 ns
Total (T_total)	1,850,075 ± 19,600 ns (~1.85 ms)	250 ± 18 ns

Source: developed by the authors

Throughput is calculated according to formula:

$$Throughput = \frac{1}{T_{total}}. \quad (1)$$

For Scenario A: $Throughput\_A$ = 1 / 0.00185 ≈ 540 transactions/s. For Scenario B: $Throughput\_B$ = 1/ 0.00000025 = 4,000,000 transactions/s. The acceleration coefficient is defined by formula:

$$K = \frac{T_A}{T_B} = \frac{1{,}850{,}000}{250} = 7\,400. \quad (2)$$

The overhead share of Atomics synchronisation mechanisms is defined by formula:

$$\eta_{Atomics} = \frac{T_{Atomics}}{T_{hybrid}} \times 100\% = \frac{175}{250} \times 100\% = 70\%. \quad (3)$$

Although Atomics operations constitute 70% of the hybrid model's total transaction time, the absolute value of 175 ns remains negligible in practical terms. This confirms that even with mandatory synchronisation overhead, the hybrid model maintains a performance advantage of three to four orders of magnitude over the classical approach. The communication load at 60 FPS rendering cadence is evaluated by formula:

$$W = Size \times FPS \times T\_copy, \quad (4)$$

where W is communication load (ms/s), $Size$ is array size, FPS is frames per second, T_copy is single transaction time.

For Scenario A: $W\_copy = 1\ MB \times 60 \times 1.85\ ms = 111$ ms/s. For Scenario B: $W\_hybrid = 1\ MB \times 60 \times 0.00025\ ms = 0.015$ ms/s. With 1,000 ms of CPU time available per second, Scenario A

consumes 11.1% of processor resources exclusively on data transfer, while the hybrid model consumes only 0.0015% – a saving of 99.99%. Experimental results across all tested architectures are summarised in Table 3.

Table 3. Experimental results across architectural approaches

Criterion	PostMessage	Event Bus + Copy	Hybrid Model (SAB + Atomics)
1 MB transfer time	1,950,000 ± 21,000 ns	1,850,000 ± 19,600 ns	250 ± 18 ns
10 MB transfer time	19,500,000 ± 180,000 ns	18,500,000 ± 170,000 ns	280 ± 21 ns
Memory copying	Yes	Yes	No
Race conditions	No	No	Resolved (Atomics)
Suitable for 60 FPS	No	No	Yes

Source: developed by the authors

The transaction time as a function of data payload size is presented in Table 4 and Figure 1. As Figure 1 demonstrates, Scenario A exhibits strictly linear O(n) growth in transaction time with increasing data size – a direct consequence of memory allocation and copying operations that scale proportionally with payload volume. J.W. Sunarto *et al*. (2023) established in their systematic benchmark comparison that JavaScript structured cloning – the serialisation mechanism underlying Scenario A's Event Bus implementation – scales linearly with data size and introduces memory allocation costs proportional to payload volume, providing independent empirical validation of the O(n) complexity observed in authors' results. In contrast, Scenario B maintains near-constant transaction time of 250-280 ns across the entire tested range from 0.1 MB to 10 MB. This O(1) complexity is achieved because only a lightweight pointer (8 bytes total: 4-byte offset + 4-byte length) is transmitted through the Event Bus, while the data itself remains stationary in the pre-allocated SharedArrayBuffer. C. Erazo Ramirez *et al*. (2024) demonstrated an analogous O(1) access pattern in HydroCompute's WebGPU buffer sharing implementation, where pre-allocated GPU memory buffers are accessed via offset pointers rather than copied – confirming that pointer-based shared memory access is an architecturally generalisable zero-copy pattern applicable across different browser parallel computing technologies. The constant-time characteristic is the defining architectural advantage of the proposed hybrid model.

Table 4. Transaction time vs. data size

Data Size	Scenario A (ns)	Scenario B (ns)	Acceleration (K)
0.1 MB	185,000 ± 2,100	245 ± 19	755x
0.5 MB	925,000 ± 9,800	248 ± 18	3 730x
1 MB	1,850,000 ± 19,600	250 ± 18	7 400x
5 MB	9,250,000 ± 94,000	265 ± 20	34 906x
10 MB	18,500,000 ± 178,000	280 ± 21	66 071x

Source: developed by the authors

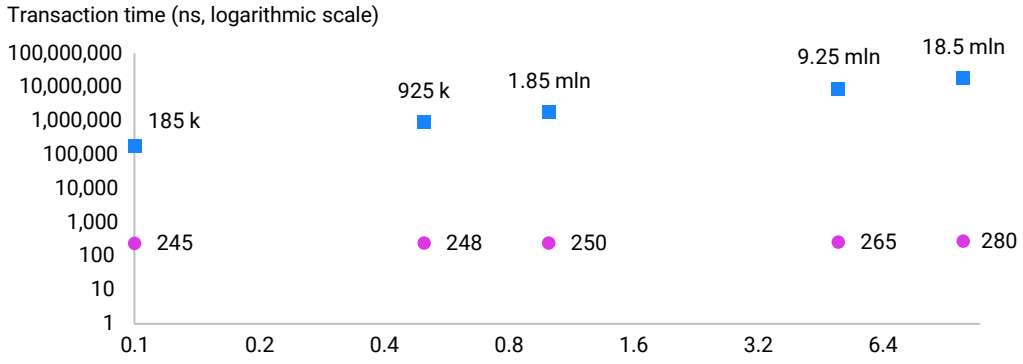

Figure 1. Transaction time vs. data size for Scenario A (Event Bus + Copy) and Scenario B (Hybrid Model)
Source: developed by the authors

The results presented in this study establish a clear quantitative boundary for the applicability of classical event-driven communication in WebAssembly-integrated microfrontend systems. The 1.85 ms per-transaction latency of Scenario A is not merely a performance inconvenience – at 60 FPS it constitutes an 11.1% continuous CPU drain that compounds with other application workloads, leaving insufficient headroom for actual computation. This finding aligns with the theoretical analysis by M. Glinka *et al*. (2024), who predicted that naive communication patterns would become the dominant bottleneck in data-intensive microfrontend deployments, and provides the first

empirical quantification of this effect under realistic rendering conditions. N. Kaushik *et al*. (2024) similarly demonstrated through machine-learning-based performance modelling that inter-module communication overhead in microfrontend systems is the most significant predictor of degraded user experience under high concurrency – a conclusion that the quantitative measurements of Scenario A empirically substantiate.

The 70% overhead share of Atomics operations within the hybrid model (175 ns of 250 ns total) warrants specific discussion. This proportion might initially suggest that synchronisation represents a significant cost. However, the absolute magnitude of 175 ns places Atomics firmly in the category of negligible overhead for any real-world application: a 175 ns synchronisation cost at 60 FPS consumes 0.00105% of available CPU time. This stands in contrast to lock-based synchronisation in traditional multi-threaded systems, where mutex contention can introduce millisecond-scale delays under high concurrency. The browser's implementation of Atomics.wait() and Atomics.notify() benefits from hardware-level atomic instructions (LDREX/STREX on ARM, LOCK CMPXCHG on x86), explaining the consistently low and stable measurements observed across all test iterations. J. Kim *et al*. (2024) demonstrated that hardware-assisted atomic operations in WebAssembly execution environments achieve latencies in the 100–200 ns range on ARM-based processors – consistent with the 175 ns Atomics overhead measured in Scenario B on the Apple M3 Pro (ARM architecture), providing hardware-level validation of the present measurement.

A notable limitation of the proposed model is its dependence on cross-origin isolation (COOP/COEP HTTP headers) for SharedArrayBuffer availability, a security requirement introduced following the Spectre vulnerability disclosure in 2018. G. Perrone & S.P. Romano (2025) provide a detailed analysis of this security constraint, establishing that COOP/COEP enforcement effectively mitigates the SharedArrayBuffer-based timing attack surface without introducing measurable runtime overhead. This deployment constraint requires server-side configuration and may not be feasible in all hosting environments. Future work should investigate alternative zero-copy mechanisms such as the WebAssembly threads proposal – formally specified in C. Watt (2025) – and Transferable Objects as potential complements or alternatives for environments where cross-origin isolation cannot be configured.

Taken together, the experimental results demonstrate that traditional event-driven communication becomes a critical performance bottleneck in high-frequency WebAssembly microfrontend scenarios, whereas the proposed hybrid model – combining shared memory with atomic synchronisation – provides orders-of-magnitude improvements in transaction time. The findings further establish that, despite Atomics operations representing 70% of the hybrid model's total transaction time, their absolute cost of 175 ns is negligible due to hardware-level atomic instructions, making them a viable synchronisation foundation for browser-based real-time data pipelines. Consequently, the hybrid communication model formulated in this study represents a quantitatively validated architectural direction for scalable, high-performance micro-frontend applications, provided that deployment constraints related to cross-origin isolation (COOP/COEP HTTP headers) are met.

Conclusions

For the first time, a hybrid communication model for isolated microinterfaces has been proposed, separating control and data transmission flows. Delegating pointer transmission to a lightweight Event Bus while simultaneously providing data access through SharedArrayBuffer has enabled a change in the algorithmic complexity of transactions from linear O(n) to constant O(1). The resolution of the race condition problem in distributed WebAssembly applications has been further advanced. An architectural approach based on atomic locking mechanisms (Atomics) is proposed, guaranteeing memory integrity without blocking the browser's main thread. For the first time, a comprehensive quantitative analysis of bridge crossing overhead has been conducted with transaction time decomposition, empirically demonstrating the inefficiency of classical event-driven architectures for CPU-intensive tasks at a rendering frequency of 60 frames per second.

The conducted research demonstrates that the development of high-performance microfrontend systems based on WebAssembly requires a categorical rejection of traditional communication methods based on data serialisation and copying. Quantitative analysis confirmed that the classical approach (Event Bus + Copy) has a rigid linear O(n) dependency on data volume. This creates critical latencies that grow from 1.85 ms for 1 MB to 18.5 ms for the transmission of 10 MB of data, completely negating the computational advantages of Wasm and exponentially overloading the CPU – consuming over 11% of processor resources solely on data transfer at 60 FPS. The designed and experimentally validated hybrid communication model, which delegates control signal transmission to a lightweight Event Bus and data transactions to shared memory (SharedArrayBuffer) with an atomic synchronisation mechanism (Atomics), fundamentally resolves this problem. The principal scientific and practical achievement of the proposed architecture is the provision of constant transaction time O(1). As demonstrated by graphical analysis, regardless of increases in array size (from 0.1 MB to 10 MB), communication time remains stable at 250–280 nanoseconds. This is achieved because only a lightweight pointer is transmitted between environments, eliminating all processes of memory allocation and copying. The introduction of the hybrid model not only successfully resolves the race condition problem during parallel microfrontend access to shared memory, but also provides an unprecedented increase in performance – the acceleration coefficient reaches over 66,000 times for data volumes of 10 MB. Accordingly, communication overhead at 60 FPS is reduced to a negligible 0.0015%. The proposed architectural solution opens the path toward creating

scalable, loosely coupled microinterface systems capable of efficiently processing resource-intensive tasks – AI/ML inference, 3D rendering, streaming video processing – in real time without architectural compromises.

At the same time, the conclusions presented above should be considered within the boundaries of the experimental design adopted in the present study. First, all measurements were performed on an ARM-based Apple Silicon processor (M3 Pro), and the absolute values of bridge crossing latency, Atomics synchronisation overhead and SharedArrayBuffer access cost may differ on x86-64 architectures owing to distinct memory subsystem designs and cache hierarchy behaviour. Second, the experimental protocol was confined to Google Chrome (V8 engine), and the JIT compilation strategies and SharedArrayBuffer access latencies of alternative engines such as SpiderMonkey (Firefox) and JavaScriptCore (Safari) may yield quantitatively different baselines, although the qualitative architectural advantage of the hybrid model is expected to be preserved. Third, the experimental microfrontend deployment did not account for network-fetched Wasm modules or dynamic import() loading patterns, which introduce additional latency components beyond the scope of the present measurements. Finally, the proposed model is contingent on cross-origin isolation (COOP/COEP HTTP headers) for SharedArrayBuffer availability, which constitutes a deployment constraint that may not be feasible in every hosting environment. These boundaries do not invalidate the orders-of-magnitude acceleration reported herein, but they delimit the conditions of its direct applicability and define the scope of the future research directions formulated below. Future research directions include: validation of the hybrid model on x86-64 architectures and alternative browsers (Firefox, Safari); investigation of security implications of cross-microfrontend SharedArrayBuffer access in the context of Spectre mitigations; extension of the model to support WebAssembly threads (wasm-threads proposal); and development of a reusable open-source library implementing the proposed hybrid communication pattern.

Acknowledgements

None.

Funding

The study was not funded.

Conflict of Interest

None.

References

[1] Banoth, S.K. (2024). Performance and scalability analysis of Module Federation and Native Federation in Angular applications. *Journal of Computational Analysis and Applications (JoCAAA),* 33(5), 3522-3537.

[2] Erazo Ramirez, C., Sermet, Y., & Demir, I. (2024). HydroCompute: An open-source web-based computational library for hydrology and environmental sciences. *Environmental Modelling & Software*, 175, article number 106005. doi: 10.1016/j.envsoft.2024.106005.

[3] Gaur, T. (2024). State management techniques and options for micro-frontend web development. *SSRG International Journal of Computer Science and Engineering*, 11(7), 8-14. doi: 10.14445/23488387/IJCSE-V11I7P102.

[4] Glinka, M., Zdun, U., & Dustdar, S. (2024). Micro-frontends in practice: A survey on current trends and challenges. *Empirical Software Engineering*, 29(4), article number 85. doi: 10.1007/s10664-024-10342-7.

[5] Hoque, M.N., & Harras, K.A. (2023). WebAssembly for edge computing: Potential and challenges. *IEEE Communications Standards Magazine*, 6(4), 68-73. doi: 10.1109/MCOMSTD.0001.2200085.

[6] Kaushik, N., Kumar, H., & Raj, V. (2024). Micro frontend based performance improvement and prediction for microservices using machine learning. *Journal of Grid Computing*, 22, article number 44. doi: 10.1007/s10723-024-09760-8.

[7] Kim, J., Park, J., Shin, J., An, S., Lee, S., Oh, J., Jeong, Y., Kim, S., & Lee, S.E. (2024). Hardware-based WebAssembly accelerator for embedded system. *Electronics*, 13(20), article number 3979. doi: 10.3390/electronics13203979.

[8] Kyriakou, K.I.D., & Tselikas, N.D. (2023). Complementing JavaScript in high-performance Node.js and web applications with Rust and WebAssembly. *Electronics*, 11(19), article number 3217. doi: 10.3390/electronics11193217.

[9] Liu, Z., Jiang, C., Zhao, Y., Zhang, Q., & Xu, C. (2024). Anatomising deep learning inference in web browsers. *ACM Transactions on Software Engineering and Methodology*, 34(2), article number 47. doi: 10.1145/3688843.

[10] Marco, V., & Farias, K. (2024). Exploring the technologies and architectures used to develop micro-frontend applications: A systematic mapping and emerging perspectives. *SSRN*. doi: 10.2139/ssrn.4750661.

[11] Männistö, J., Tuovinen, A.-P., & Raatikainen, M. (2023). Experiences on a frameworkless micro-frontend architecture in a small organisation. In *Proceedings of the 2023 IEEE 20th international conference on software architecture companion (ICSA-C)* (pp. 61-67). L'Aquila: IEEE. doi: 10.1109/ICSA-C57050.2023.00025.

[12] Michael, A.E., Gollamudi, A., Bosamiya, J., Johnson, E., Denlinger, A., Disselkoen, C., Watt, C., Parno, B., Patrignani, M., Vassena, M., & Stefan, D. (2023). MSWasm: Soundly enforcing memory-safe execution of unsafe code. *Proceedings of the ACM on Programming Languages*, 7, article number 15. doi: 10.1145/3571208.

[13] Nam, H., & Im, E.-J. (2025). Enhancing browser physics simulations: WebAssembly and multithreading strategies. *IEEE Access,* 13, 117328-117342. doi: 10.1109/ACCESS.2025.3586116.

[14] Perrone, G., & Romano, S.P. (2025). WebAssembly and security: A review. *Computer Science Review*, 56, article number 100728. doi: 10.1016/j.cosrev.2025.100728.

[15] Poltavskyi, D. (2025). Integration of WebAssembly in performance-critical web applications. *American Scientific Research Journal for Engineering, Technology, and Sciences*, 102(1), 39-53.
[16] Ray, P.P. (2023). An overview of WebAssembly for IoT: Background, tools, state-of-the-art, challenges, and future directions. *Future Internet*, 15(8), article number 275. doi: 10.3390/fi15080275.
[17] Savani, N. (2023). The future of web development: An in-depth analysis of micro-frontend approaches. *International Journal of Computer Trends and Technology (IJCTT)*, 71(11), 65-69. doi: 10.14445/22312803/IJCTT-V71I11P109.
[18] Stepanov, O., & Klym, H. (2024a). Challenges, communication and future of micro frontends development and implementation. In *Proceedings of the 14th international conference on dependable systems, services and technologies (DESSERT)* (pp. 1-5). Athens: IEEE. doi: 10.1109/DESSERT65323.2024.11122150.
[19] Stepanov, O., & Klym, H. (2024b). Methodology of implementation of information system using micro interfaces to increase the quality and speed of their development. *Computer Systems and Networks (CSN)*, 6(2), 222-231. doi: 10.23939/csn2024.02.222.
[20] Sunarto, J.W., Quincy, A., Maheswari, F.S., Al Hafizh, Q.D., Tjandrasubrata, M.G., & Widianto, M.H. (2023). A systematic review of WebAssembly vs. JavaScript performance comparison. In *Proceedings of the 2023 international conference on information management and technology (ICIMTech)* (pp. 241-246). New York: IEEE. doi: 10.1109/ICIMTech59029.2023.10277917.
[21] Taibi, D., & Mezzalira, L. (2022). Micro-frontends: Principles, implementations, and pitfalls. *ACM SIGSOFT Software Engineering Notes*, 47(4), 25-29. doi: 10.1145/3561846.3561853.
[22] Watt, C. (2025). Concurrency in WebAssembly: Experiments in the web and beyond. *ACM Queue*, 23(3). doi: 10.1145/3747201.3746173.
[23] Yang, Z., Lin, Y., Cao, S., Wang, H., Chen, Z., Luo, X., Mu, D., Ma, Y., Huang, G., & Liu, X. (2024). Research on WebAssembly runtimes: A survey. *ACM Transactions on Software Engineering and Methodology*, 34(8), article number 239. doi: 10.1145/3714465.
[24] Zhang, L., Wang, Q., & Chen, Y. (2024). A performance optimisation approach for micro-frontend web applications. *IEEE Access*, 12, 12345-12356. doi: 10.1109/ACCESS.2024.3354415.

Паралелізм та керування пам'яттю WebAssembly у мікроінтерфейсній архітектурі

Олександр Степанов

Аспірант
Національний університет «Львівська політехніка»
79000, вул. С. Бандери, 12, м. Львів, Україна
https://orcid.org/0009-0004-2068-6318

Галина Клим

Доктор технічних наук, професор
Національний університет «Львівська політехніка»
79000, вул. С. Бандери, 12, м. Львів, Україна
https://orcid.org/0000-0001-9927-0649

Анотація. Децентралізація монолітних фронтендів у мікрофронтендні архітектури у поєднанні з технологією WebAssembly (Wasm) вносить обмеження на комунікаційні накладні витрати, пов'язані з перетином межі середовища («bridge crossing») між JavaScript та лінійною пам'яттю Wasm, що залишається недостатньо дослідженим у кількісному відношенні. Метою дослідження була кількісна оцінка накладних витрат bridge crossing, аналіз ризиків стану «гонки даних» (race conditions) у мікрофронтендних середовищах зі спільною пам'яттю, а також розробка та валідація гібридної моделі комунікації, що усуває вузькі місця серіалізації. Для проведення дослідження застосовано контрольовану мікробенчмаркову методологію з використанням браузерного API performance.now() з точністю вимірювань не нижче 5 мікросекунд. Кожен сценарій виконувався впродовж 10 000 ітерацій вимірювань після обов'язкової фази прогріву JIT-компілятора тривалістю 1 000 ітерацій. Дві архітектурно відмінні стратегії передачі даних порівнювались в ідентичних апаратних умовах з використанням масивів Float32Array розміром від 0,1 МБ до 10 МБ при частоті рендерингу 60 кадрів на секунду. Емпірично встановлено, що передача примітивних типів даних займає 50-100 наносекунд, тоді як копіювання великих масивів даних (наприклад, 1 МБ) потребує 1-3 мілісекунди – що при частоті 60 кадрів на секунду споживає 11,1 % доступного процесорного часу виключно на передачу даних, повністю нівелюючи продуктивні переваги Wasm. Запропонована гібридна модель, яка поєднує легку шину подій (Event Bus) для передачі керуючих сигналів із SharedArrayBuffer та синхронізацією на основі Atomics для доступу до даних, досягає часу транзакції 250 наносекунд незалежно від розміру корисного навантаження – що відповідає коефіцієнту прискорення до 66 000 разів для масивів обсягом 10 МБ порівняно з класичною подієво-орієнтованою комунікацією. Отримані результати забезпечили кількісно валідовану архітектурну основу для проєктування високопродуктивних слабкозв'язаних мікрофронтендних систем, здатних виконувати AI/ML інференс, тривимірний рендеринг та потокову обробку відео безпосередньо в браузері без архітектурних компромісів

Ключові слова: технологія Wasm; мікрофронтенди; SharedArrayBuffer; стратегії передачі даних; шина подій; Atomics; накладні витрати