Bachelor Thesis

IU International University of Applied Sciences
Bachelor of Science Media Informatics

Achieving Low Latency and High Throughput in Real-Time Web Applications Using Web Workers

Author:	Christoph Stecher
Supervisor:	Prof. Dr. Paul Libbrecht
Date of Submission:	01. August 2024

Abstract

This bachelor thesis compares client-sided Web Worker data transmission mechanisms for computationally demanding real-time web applications via a custom-developed microbenchmarking suite. The findings show performance characteristics for different workloads and data types, operating systems, and hardware architectures. Suggestions for optimal data transfer to and from Web Workers in sandboxed browser environments are provided based on benchmark results.

The automated web application benchmark suite and test definitions developed in this thesis provide valuable guidance for web benchmarking and highlight areas for improvement in other Web Worker benchmarks. Data for binary large objects, floating-point values, JavaScript objects, and latency tests, in a variety of test configurations was collected and then statistically analyzed using the programming language R. The results are consistent between browsers and show that conventional event-based messages between threads provide minimal latency compared to other methods. However, when transmitting large amounts of data, pre-allocated shared linear memory regions have the least overhead. The most significant increase in performance can be achieved by applying software design patterns that enable JavaScript engine optimizations.

Keywords: Web Worker, Concurrency, Parallel Computing, Real-Time, Web Applications, High Throughput, Low Latency, Transmission

Diese Bachelorarbeit vergleicht clientseitige Web Worker Datenübertragungsmechanismen für rechenaufwändige Echtzeit-Webanwendungen anhand einer selbstentwickelten Mikrobenchmark-Suite. Die Ergebnisse weisen Leistungsmerkmale für verschiedene Arbeitslasten und Datentypen unter mehreren Betriebssystemen und Hardwarearchitekturen nach. Auf Basis der Benchmark-Ergebnisse werden Vorschläge zur optimalen Gestaltung von Datenübertragungen zu und von Web Workern in isolierten Browserumgebungen präsentiert.

Die entwickelte, automatisierte Benchmark-Webanwendung und die dazugehörigen Testdefinitionen stellen einen wertvollen Leitfaden Web-Benchmarking dar und zeigen Verbesserungsmöglichkeiten in anderen Web Worker Benchmarks auf. Es wurden Daten für binäre Großobjekte, Gleitkommawerte, JavaScript Objekte und Latenztests in einer Vielzahl von Testkonfigurationen gesammelt und anschließend mit der Programmiersprache R statistisch ausgewertet. Die Ergebnisse sind konsistent zwischen Browsern und zeigen, dass konventionelle ereignisbasierte Nachrichten zwischen Threads, im Vergleich zu anderen Methoden, eine minimale Latenz aufweisen. Wenn allerdings große Datenmengen übertragen werden, haben vorab erstellte, gemeinsame lineare Speicherbereiche den geringsten Overhead. Die größte Leistungssteigerung kann durch Anwendung von Software-Entwurfsmustern erreicht werden, die Optimierungen seitens der JavaScript-Engines ermöglichen.

Stichwörter: Web Worker, Nichtsequentialität, Parallele Datenverarbeitung, Echtzeit, Webanwendungen, Hoher Durchsatz, Niedrige Latenz, Übertragung

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
BLOB	Binary Large Object
CPU	Central Processing Unit
CORS	Cross-Origin Resource Sharing
CSV	Comma-separated Values
DOM	Document Object Model
DSR	Design Science Research
FTL	Faster Than Light
GB	Gigabyte
GC	Garbage Collection
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
HTTPS	Hyper Text Transfer Protocol Secure
Hz	Hertz
I/O	Input/Output
JIT	Just-In-Time
JS	JavaScript
JSON	JavaScript Object Notation
KB	Kilobyte
KiB	Kibibyte
LTS	Long-term Support
MB	Megabyte
MiB	Mebibyte
ML	Machine Learning
MMU	Memory Management Unit
ms	Milliseconds
NTP	Network Time Protocol
PDF	Portable Document Format
RAM	Random Access Memory
SIMD	Single Instruction, Multiple Data
SOP	Same-origin Policy
URL	Uniform Resource Locator
POSIX	Portable Operating System Interface
W3C	World Wide Web Consortium
WASM	WebAssembly
WebDAV	Web-based Distributed Authoring and Versioning

1 Introduction

1.1 Problem Statement

Enabled by the computing landscapes significant change over the past decades, the web has slowly and steadily developed into a ubiquitous computing platform: Web standards define Application Programming Interfaces (APIs) for presenting rich media content and application functionality in a platform-independent and secure manner. Workloads that used to require full-featured desktop software installations, specific to one operating system and hardware architecture, can now be conveniently processed on the web by simply accessing a website. Entire applications are being written on top of web APIs, including graphics editors and digital audio workstations (Kutskir, 2017; Soundtrap AB, 2024).

JavaScript code on the web is sandboxed to protect users and to isolate or block potentially malicious code distributed by third-party websites (Google LLC., 2024b). These sandboxing mechanisms deviate from low-level system resource management capabilities (e.g. processes), that native Windows and Portable Operating System Interface (POSIX) APIs provide (Haverbeke, 2018, p. 222). When an application with resource intensive workloads is developed and affected by bottlenecks in JavaScript's single-threaded runtime environment, Web Workers offer a computing model comparable to native threads to offload computations.

While approaches for parallel computation with threads and interprocess communication primitives, provided by operating systems and hardware, are a well-researched topic (Kerrisk, 2010, p. 617-620; Silberschatz et al., 2013, p. 122), research on multi-threading and communication primitives for Web Workers is sparse: Many existing work has since been superseded by new web standards and browser features, does not consider new security mechanisms (such as timer precision), is of insufficient detail, or does not consider latency requirements at all. Some browser implementations are also lacking behind specified standards, e.g. with Mozilla Firefox currently not providing asynchronous operations for certain Web Worker data transmission architectures (Mozilla Corporation, 2018a).

1.2 Objectives and Research Question

With over 63% of developers using JavaScript and web tooling for their applications (Stack Overflow, 2023), and modern workloads increasingly demanding more resources while maintaining responsiveness, understanding the impact of concurrency techniques is crucial for developers seeking to optimize their applications. Especially when considering real-time workloads, every optimization counts to guarantee a satisfactory user experience on a variety of computing devices. To achieve this, the thesis focuses on quantifying the performance gains that specific data transfer methods in concurrent application architectures can bring to the web. With latency and throughput measurements under various workloads, the thesis aims to provide empirical data that highlights the tangible benefits of each data transmission method for real-time constrained use-cases.

By comparing low- (e.g. shared memory and atomics) and high-level (e.g. message passing with implicit serialization) data communication primitives, the thesis seeks to determine which approaches offer the best performance for certain types of workloads and application architectures. This comparison will help developers to identify key aspects that are necessary to develop high throughput and low latency real-time web applications using Web Workers and make informed decisions about which data transmission methods to adopt for concurrent workloads in their projects.

The thesis additionally analyzes how well web applications employing concurrency techniques can scale under increased load. This is tested using a set of benchmarks across several devices, hardware architectures, operating systems, and browsers and it illustrates potential bottlenecks as well as limitations when handling a variety of workloads. The aim is to promote best practices for implementing computationally demanding use-cases in real-time web applications based on these experiments and highlight differences of browser implementations and operating systems, and area vectors where browsers themselves can be improved. The thesis should ideally act as resource to empower the developer community and provide scientifically validated and practical recommendations to effectively utilize concurrency and therefore enhance application performance and scalability, contributing to the advancement of high-performance web development.

1.3 Structure of the Work

The thesis is divided into six chapters. This first introductory chapter is followed by a theoretical foundation, explaining all necessary terms and ensuring a common understanding of concepts covered in later chapters. Besides general computer science knowledge regarding concurrency and associated limitations, as well as real-time constraints, an overview of core web technologies and their performance related characteristics are given. Chapter three reviews previous research and highlights related work from other authors, after which the fourth chapter addresses the methodology and quantitative data analysis used in this thesis. The development approach for the benchmark software used for data acquisition is based on the Design Science Research (DSR) process. A presentation and discussion of the findings to answer the research question follows in the fifth chapter. The sixth and final chapter concludes this work.

2 Theoretical Foundation

2.1 Parallel Computing

The focus of this thesis is to find optimal transmission methods between web contexts for a variety of semi-independent tasks that need to be run in parallel. For example, a digital audio workstation would dynamically load in samples and effects, process audio (i.e. apply filters), generate new tracks, have some amount of crosstalk for e.g. sidechains, and later combine all tracks into one or two output channels, all using the CPU (Kiefer, Molitorisz, Bieler, & Tichy, 2015, p. 405). A 3D graphics application on the other hand would typically run dedicated code on the shader units of a Graphics Processing Unit (GPU), which are simpler and more limited, but very effective at filling out grids of pixels in a highly dynamic graphics scene (Pacheco, 2011, p. 32). It is an application developer's task to pick the correct optimization strategy, though the aim of this thesis is to enable a more informed choice when the application's target platform includes Web Workers. Other non-CPU related acceleration techniques, e.g. WebGPU to access GPUs resources that can, according to the specification, in the future also be shared to worker contexts (World Wide Web Consortium, 2024b), are not further detailed in this thesis. To be able to apply optimizations in the subsequent chapters of this work, fundamental parallel computing concepts are explained first.

Modern computers provide more than a single unit of execution (CPU core). Eventually, a program that performs computationally expensive tasks will exhaust the resources of a single CPU core and can only improve in performance by making use of additional Central Processing Unit (CPU) cores or other acceleration hardware of the machine (Pacheco, 2011, p. 3-4). Not all workloads benefit from the same kind of parallelization approach. Whether a workload scales weakly or strongly also depends on its coupling or rather independence on data (Diehl et al., 2024, p. 69-75; Silberschatz et al., 2013, p. 168; Hunter and English, 2021, p. 176).

2.2 Instruction Pipelining and SIMD

CPU cores consist of complex circuitry with different execution areas, dedicated to specific functions, for example floating-point and integer registers, caches, etc. Executing simply one instruction after another would utilize only part of the die circuitry relevant for said instruction at a time (Pacheco, 2011, p. 25). To optimize data throughput and consequently the average amount of instructions executed per clock cycle, CPUs make use of instruction pipelining and optionally Single Instruction, Multiple Data (SIMD).

Instruction pipelining is an internal and transparent mechanism to detect independent machine instructions and distribute them efficiently to execution units within the CPU core, which then execute them in parallel (ibid., p. 25-28). While the inner workings of automated instruction pipelining are an entire research topic on their own, the relevant bit for this thesis is that instruction pipelining is only free of side effects regarding observable changes of computer memory, but not timing. Speculative execution, which is a specific optimization in instruction pipelining, has security implications that

directly impact the design of the web sandbox, as later described in section 2.10.

CPUs and programming languages provide additional building blocks to process data in parallel on a single CPU core in the form of SIMD extensions (Pacheco, 2011, p. 29-30). Compact machine code can execute complex operations that are coordinated among multiple elements of the CPU core circuitry without involving instruction pipelining or optimization heuristics, which delivers predictably high performance and high core utilization. Compilers will attempt to generate code that optimizes the utilization of CPU circuitry (Ghosh, 2023, p. 78-88). Using additional hints or explicit use of SIMD instruction in low-level code allows applications to optimize per-core throughput before advancing into parallelization.

2.3 Processes and Threads

Once a single core is at its limit, and if the workload allows for it, computations need to be scheduled on additional CPU cores to further scale the workload across available hardware resources (Kerrisk, 2010, p. 38-39; Silberschatz et al., 2013, p. 263). Threads are a basic unit of execution that enable this kind of scaling. In POSIX, as implemented by Linux, macOS, and other UNIX-like systems, a newly launched process will consist of one thread by default. This single thread can then fork into additional processes, or spawn additional threads in the same process. The primary difference between processes and threads is the level of isolation that they provide. While threads and processes are theoretically equally powerful in computational capabilities, all threads within a process share their view on memory and available program code by default, and hence provide a simple parallelization primitive. Complex programs can leverage processes for stability, security, and performance (A. Williams, 2012, p. 6-7).

2.4 Shared Memory

Isolation and security properties are no longer enforced by application logic alone, but also the operating system (e.g. processes, reduced privileges to access APIs) and the hardware/CPU (virtual memory) (Kerrisk, 2010, p. 23-24 and 118-121). Shared memory is a concept where programs can share a view on memory across multiple processes, which yields access times equal to managing that memory directly inside a thread, but still provides most of the isolation properties of processes (ibid., p. 997). All parties that have access to a shared memory region can theoretically read and write to it at any given time, causing problems if no proper synchronization technique is applied.

2.5 Problems

2.5.1 Deadlocks

Circular dependencies or incorrectly programmed synchronization mechanisms may result in a hanging program. When a thread needs to use two or more shared resources and one of the resources is already in use/locked by another thread that also needs access to both resources, neither thread can continue (A. Williams, 2012, p. 47). Trying to acquire locks on shared resources in the same

order each time can prevent deadlocks since an already locked resource prevents other threads from continuing. However, this does not solve problems such as priority inversions and race conditions (Silberschatz et al., 2013, p. 217-218).

2.5.2 Race Conditions

Race conditions can occur when an application depends on the sequence or timing of events in concurrent workloads where multiple threads access shared memory resources simultaneously (Pacheco, 2011, p. 50-51; A. Williams, 2012, p. 35-37). If these accesses are not properly synchronized, the system can produce unpredictable and erroneous outcomes. For example, when two threads update a shared counter variable and both threads read the current value of the counter at the same time, increment it, and then write back the result, the counter will only be incremented once instead of twice. Concurrent access to a shared region of memory, regardless of whether it is shared between threads or processes, requires proper synchronization mechanisms such as locks or atomic operations, to prevent race conditions and ensure that all involved parties have a consistent view on the data (Kerrisk, 2010, p. 631).

2.6 Task Scheduling and Synchronization

2.6.1 Atomic Operations

One synchronization solution are atomic operations, which are operations that happen in a single step, i.e. without observable side effects for other concurrent processes/threads (ibid., p. 90). Logical and mathematical operations, for example the addition of two variables and storing the result into a third, are not atomic by default for several reasons. Firstly, such operations usually consist of multiple internal sub-operations, e.g. reading both input variables, adding them, then storing the result. A concurrent process that runs the same operations would falsify the result, because these operations are not associative as described in section 2.5.2. Secondly, CPUs themselves employ caches that would make the memory write of the result invisible to other cores, that maintain their own cached view on memory. Thirdly, even systems with single-core CPU require proper synchronization if a preemptive scheduler is in use, which could switch to another program or thread at any time (Silberschatz, Galvin, & Gagne, 2013, p. 207 and 263-264). CPUs therefore offer instructions to enable basic atomic operations and synchronize caches. These instructions are offered by programming languages via “intrinsics” and are the building blocks for more high-level synchronization functions.

2.6.2 Mutual Exclusion via Semaphores/Spinlocks/Futexes

Atomics usually operate on a single memory address, and generally cannot manipulate more complex data structures safely. Mutual exclusion (Mutex) is a locking concept that ensures concurrent access to complex data structures in a safe manner (Kerrisk, 2010, p. 639; Hunter and English, 2021, p. 131-132). This synchronization between shared data structures among multiple CPU cores is achieved

by preventing access within one critical section from several threads at the same time.

Mutexes can be implemented either via the operating system to integrate with scheduling (so that a waiting process will yield its execution time to other processes that can post to the semaphore), purely in user space using atomic operations (e.g. via spinlocks, which use up all their CPU time until a concurrent core posts to continue), or via an optimized combination of the two (e.g. futexes, where the kernel assists in process synchronization only if preconditions are met to avoid unnecessary context switches) (Pacheco, 2011, p. 169). Browsers use these primitives to provide core functionality, e.g. Firefox uses futexes for implementing the `Atomic.wait()` operation (Mozilla Corporation, 2024b).

A generic implementation are semaphores, as defined by e.g. the POSIX standard via `sem_open()`, `sem_wait(sem)` and `sem_post(sem)` (Kerrisk, 2010, p. 1089-1090; Pacheco, 2011, p. 174-175 and 199). A semaphore has an initial value set via init, which can be decremented via wait and incremented via post. The wait operation will block if the counter is zero or below, whereas the post operation will increment the counter and might release a waiting thread. An initial value of 1 effectively creates a binary lock.

2.6.3 Task Queues and Job Graphs

A task queue is a more high-level synchronization concept where a message queue is maintained to insert work into and have it processed by threads asynchronously (Pacheco, 2011, p. 241-242; Kerrisk, 2010, p. 1063-1068 and 1087). This significantly simplifies ensuring safe access to state across threads in a performant manner, by isolating state and their side effects to tasks. The actor model would further enable verification of side effects being safe (A. Williams, 2012, p. 100; Hunter and English, 2021, p. 144).

Tasks queues are built on top of the basic primitives described in previous sections. Semaphores can for example be used to both synchronize access to a task queue and ensure that threads are efficiently put to sleep when there is no work to process (Pacheco, 2011, p. 171-175 and 207). Depending on the implementation, a condition variable can enable easier modelling of the behavior by waiting for incoming tasks and automatically putting threads back to sleep when a condition (tasks to process are present) is not met (Pacheco, 2011, p. 179; A. Williams, 2012, p. 69-75).

2.7 Real-Time Computing

2.7.1 Hard, Firm and Soft Deadlines

Computations are real-time when they have a strict deadline until which they must complete, either in terms of wall clock time, or number of CPU cycles. Deadlines are classified by their failure mode (Tanenbaum & Bos, 2015, p. 37-38 and 87 and 164):

- Hard deadlines must be kept under all circumstances, otherwise the entire system will fail.
- Firm deadlines allow for some form of costly recovery when missed.
- Soft deadlines must be upheld to guarantee some amount of quality of service.

A controller for a rocket which does not issue a command in time qualifies as a missed hard deadline and could let the rocket enter an unstable and unrecoverable state, leading to destruction (Tanenbaum & Bos, 2015, p. 38). Robots in an assembly line would reset to a known-good state after missing a firm deadline, which increases production cost. Audio systems that miss a soft deadline for calculation of the next audio buffer cannot pass it on to hardware, which results in audio glitches ("dropouts"). The calculation of future buffers is, however, unaffected and continues normally.

2.7.2 Complexity of Real-Time Computing

The implementation of hard- and firm-deadline multi-tasking systems is challenging without support from the firmware and operating system. Timely completion of hard real-time tasks is only possible if all components that can interrupt tasks, such as firmware, operating system, and application runtimes, are aware of deadlines and can therefore schedule concurrent tasks so that meeting the deadline is guaranteed (ibid., p. 87 and 164-167).

Desktop and server systems almost exclusively allow for designing systems with soft deadlines. The firmware (power management interrupts), operating system (scheduling), application runtime (e.g. JavaScript which must perform Just-In-Time compilation), and memory manager (e.g. a Garbage Collector) are mostly unaware of deadlines of individual tasks (ibid., p. 87 and 164-167). However, they allow for some amount of prioritization to make it more likely that soft deadlines are met regardless of whether the system is overloaded.

Real-time deadlines are usually never isolated, but a series of deadlines that must be upheld for a system to perform its function properly. For example, in audio systems, buffers must not only be computed and submitted in time but must also be delivered to monitors (i.e. a sound system or in-ear monitors) or a remote party (e.g. over a telephone network) with acceptable latency to remain synchronized and therefore useful. Videogames and other multimedia applications consist of several real-time pipelines, such as system resource management, physics calculations, rendering (GUI and in-game camera), audio, event management, and networking (Haigh-Hutchinson, 2009, p. 4).

2.7.3 Real-Time Optimizations

When processing audio at a sample rate of 44100 Hz and a typical buffer size of 256 samples, the window until the next buffer calculation needs to happen is 5.8 ms. Graphical applications that run at a refresh rate of 60 Hz have less than 16.67 ms of processing time available until the next frame is due.

To more likely meet (soft) real-time deadlines, general optimization techniques, such as optimizing memory layout by grouping frequently used data together, can be helpful (A. Williams, 2012, p. 238). In audio applications, data proximity can be kept close by interleaving samples inside a buffer (left and right channel) to keep the memory access contiguous (Pacheco, 2011, p. 31; Tanenbaum and Bos, 2015, p. 87). Videogames might make use of an entity component system for more efficient memory layout.

Other optimization methods include parallel computation techniques such as SIMD, as described in section 2.2. Chapter 3.1 briefly describes some parallelism strategies and how they compare against each other.

2.8 Memory Management

2.8.1 Virtual Memory

CPUs are built with multi-process systems in mind and offer a Memory Management Unit (MMU) to let operating systems efficiently isolate processes (Kerrisk, 2010, p. 120, 514 and 527). The MMU addresses the following problem: If all processes were capable to access the entire Random Access Memory (RAM) through one flat address space, then one process alone could cause data corruption in other areas in case of programming errors. It would not be possible to safely scale processes and manage all their used resources reliably (Silberschatz et al., 2013, p. 14-15).

Virtual memory is the address space presented to a program by the CPU/MMU. It maps virtual memory page ranges to physical memory or devices (Kerrisk, 2010, p. 118-121). A page has a hardware-specific size, e.g. 4 KiB or 16 KiB. The operating system tracks which physical memory ranges belong to which programs and maintains page tables to fulfill memory layout change requests for programs. Multiple programs can share the same physical pages with different virtual addresses. This, combined with signaling primitives described earlier, forms the basis of interprocess communication mechanisms. Co-processors like GPUs can be accessed directly through virtual memory via memory mapped I/O to minimize the overhead and latency of e.g. texture copies and editing of the GPU command queue via the graphics driver without going through the kernel.

Due to virtual memory, it is possible for operating systems to remove parts of running processes from main memory heuristically, e.g. to fulfill requests of another process (Pacheco, 2011, p. 23-25). If the memory is accessed again, the CPU will fail to resolve the virtual address and issue a fault to the operating system. The operating system can at that point swap the removed bit of memory back in; this is generally referred to as page fault (Kerrisk, 2010, p. 119). If the operating system concludes that there is no memory to swap in, it issues a segmentation fault and notifies the process through a signal handler. Applications will usually terminate in response to a segmentation fault.

2.8.2 Heap and Stack Memory

The stack and the heap are the two primary constructs used by programs to maintain their memory (Kerrisk, 2010, p. 115-122; Tanenbaum and Bos, 2015, p. 753-767). In addition to stack and heap, advanced programs use shared/mapped memory, memory mapped I/O, or other custom approaches optimized to the current platform and capabilities of the CPU (e.g. huge pages, sparse use of virtual memory).

The stack is a set of pages, usually kilobytes to a few megabytes in size, that is allocated per thread (Kerrisk, 2010, p. 115-122). Management of the stack is assisted by the CPU. The purpose of the

stack is to provide a temporary storage mechanism for functions when deferring execution to sub-functions of the same program. Local variables are then stored onto the stack, information about where to return to is added, and the sub-function is called. Upon completion, the sub-function will invoke a return command of the CPU, which looks up the return address on the stack and restores the previous function's context.

The heap on the other hand is a data structure shared by multiple threads of one process (Kerrisk, 2010, p. 115-121). Programs use it either for storing long-living data that escapes the scope of the stack, or for storing short-lived data of unknown or unbounded size. A program's runtime offers functionality for heap memory management, e.g. libc's malloc and free functions. Their implementation will use primitives previously described in section 2.6 to ensure that concurrent editing of heap management data structures is done safely. Compared to stack memory, the available space on the heap is plenty but care must be taken to minimize the amount (but not size) of allocation requests for optimal performance, so that the number of edits done to heap structures is small.

2.8.3 Manual Memory Management

The simplest yet error-prone method of memory management is to simply not have an automated memory management mechanism at all (Ghosh, 2023, p. 103-106). It is the programmer's task to release memory at the appropriate point when it is no longer required. The complexity of this task is proportional to that of the program and how far memory/objects propagate.

2.8.4 Reference Counting

To assist in allocating and releasing objects, programming languages (e.g. PHP, Objective C) have automatic reference counting (A. Williams, 2012, p. 44 and 200-204; Silberschatz et al., 2013, p. 795). C++ offers a construct for enabling reference counting on objects via `shared_ptr`. Objects are released when they are no longer referenced. However, it possible for an object to reference itself or reference objects that, in a cycle, reference the original object. These objects would be leaked. Reference counting therefore either requires the addition of a Garbage Collector for finding and eliminating cycles, or the use of "weak" references, which do not increment an object's reference count. The latter must be used explicitly by the programmer.

2.8.5 Garbage Collection

The least error-prone and most easy to use method of memory management is Garbage Collection (GC). It provides simplicity and reduced chance of incorrect memory management but increases CPU time and runtime complexity by running the Garbage Collector (A. Williams, 2012, p. 188; Ghosh, 2023, p. 11). The most common approach for Garbage Collection is a mark-and-sweep algorithm, where a separate thread occasionally pauses the entire program, marks reachable memory from a consistent program snapshot, and finally releases unmarked (unreferenced) memory (Lavieri et al., 2018, p. 157; Silberschatz et al., 2013, p. 526 and 737). Disadvantages of Garbage Collection, besides the increased CPU usage to run the mark-and-sweep algorithm, are higher jitter due to pauses

of program threads, and increased memory usage since memory is retained until at least the next run of the Garbage Collector, possibly longer when optimizations for incremental Garbage Collection are applied. Garbage Collection is the most common memory management technique for high-level programming languages, including JavaScript, Java, Go, and .NET.

Garbage Collection is often not recommended in use with real-time applications as it introduces non-deterministic pauses during program execution, which can violate strict timing constraints (Tanenbaum and Bos, 2015, p. 74 and 123-124; Ghosh, 2023, p. 11; Silberschatz et al., 2013, p. 526). Latency spikes may be the result when Garbage Collection operations halt program execution to reclaim memory.

2.9 Web Runtime Environments

2.9.1 Sandbox

Web browsers will begin to parse and execute code from a website as soon as it loads, if not specified otherwise (deferred) (Zakas, 2010, p. 1). Users are accustomed to visiting any kind of website on their device, which in return might load additional untrusted code via e.g. advertisement services (Van Acker & Sabelfeld, 2016, p. 32). Code must not only execute quickly in a web browser for good user experience, but also in a secure sandbox, so that one site cannot compromise the user's data on another or on the local device (Tanenbaum & Bos, 2015, p. 358 and 471-472). JavaScript engines implement secure code execution and offer an API surface that does not allow access to OS features, in addition to layers of platform-specific sandboxing mechanisms in case the innermost JavaScript engine is compromised.

Isolation of site data is done by the Same-origin Policy (SOP), which defines that sites with different origins cannot interact with each other's resources. Cross-Origin Resource Sharing (CORS) features allow relaxing the SOP (Van Acker & Sabelfeld, 2016, p. 39-40). Content security policies further restrict which remote resources can be loaded into the current site's context and act as an exploit mitigation technique. To defend against Man in the Middle attacks, which could inject code into insecure connections and thus access resources on that insecure origin, browsers additionally limit access to sensitive APIs to sites that run within a secure context (Haverbeke, 2018, p. 313-315). A context is secure when the browser can be reasonably certain that the remote server that serves the requests is the actual origin (as per SOP). Specifically, it is required to connect via HTTPS, which utilizes Transport Layer Security (TLS), or to a non-network resource like the local machine (localhost), where it is impossible to intercept network traffic.

2.9.2 JavaScript Engines

The task of a JavaScript engine is the interpretation and optimal execution of JavaScript code (Van Acker & Sabelfeld, 2016, p. 34-35). Each major web browser provides its own JavaScript engine, as shown in Table 1.

Table 1: Browser Engines

Browser	Browser Engine	JavaScript Engine
Google Chrome	Chromium	V8
Mozilla Firefox	Gecko	SpiderMonkey
Apple Safari	WebKit	JavaScriptCore

Source: Own representation

These implementations are built around the ECMA-262 specification, which defines the language syntax, semantics, and some core built-in functionality to work with objects, arrays, and related core language features for JavaScript (Ecma International, 2024). Web browsers additionally provide Hyper Text Markup Language (HTML) features, including an event loop integrated with site rendering, a Document Object Model (DOM), and other browser-related APIs such as Web Workers for multitasking. A JavaScript program on a website can either just augment the site layout or implement the entire application logic client-sided within the web browser.

Later in this thesis, all three major JavaScript engines are tested with an extensive set of data transmission workloads to gather performance measurements representative of those in parallel computation tasks. While all engines are built according to the same JavaScript and HTML specifications, some differ in their implementation. The specifications present behavior in an abstracted way, leaving out implementation details. This can affect runtime behavior, such as execution timings and behavior in relation to the Garbage Collector, leading to different user experiences on different browsers. Additionally, JavaScript and HTML support an extreme variety of features, with new ones being proposed on monthly basis. Browsers tend to not have feature parity, which impacts implementation design and performance even further.

2.9.3 JavaScript Optimizations

Modern JavaScript engines use Just-In-Time compilation to dynamically translate and incrementally optimize native machine code from the textual JavaScript representation (Google LLC., 2024b). Optimization of dynamically executed JavaScript is an active area of research, with speed improvements being released to web browsers continuously. Several design decisions of JavaScript make optimization a challenge (Zakas, 2010, p. xii and 27-31 and 53):

- It is a weakly and dynamically typed language, so that code generally cannot be generated for specific data types.
- Object prototypes are mutable, so every function call is indirect.
- The order of execution and variables accessed is mostly unpredictable when asynchronous function calls are used heavily.
- Garbage Collection for memory management adds additional complexity.

When executing JavaScript code, it is first parsed and turned into an abstract syntax tree, which is then converted into machine code by a baseline compiler, such as V8 Ignition, the SpiderMonkey Interpreter or the WebKit Faster Than Light JIT Compiler (FTL) (Wagner, 2014; Google LLC., 2024b; Pizlo, 2016). While running the machine code, profiling data is collected so that the optimizing compiler (V8 TurboFan, SpiderMonkey BaselineJIT) can make assumptions about the machine code and generate faster machine code. De-optimizations and "bailouts" to slower machine code are necessary when those assumptions are not met. This is explained more in depth in the following paragraph. SpiderMonkey can further optimize frequently called functions using a secondary bytecode optimizer called IonMonkey. JavaScriptCore also has additional compilers, that baseline optimize the generated machine/byte code and then further optimize using their Data Flow Graph (DFG) or FTL compiler (Pizlo, 2016).

Optimizations of JavaScript are heuristic and happen at runtime. Profiling data of "hot" functions, which are called frequently, are analyzed for common types (Google LLC., 2024b). When possible, faster machine code is created to handle those types specifically, so that heavily used paths of a JavaScript program become faster over time. This is based on JavaScript objects being essentially dictionaries with key/value pairs (Ecma International, 2024, chap. 6.1.7). Already seen object shapes (i.e. key names, order of keys and value data types) can be memorized and mutated, e.g. when a new parameter is added. Minor deviations, such as changing field orders in objects, causes the JavaScript engine to fall back to a slow path and resume interpretation with baseline bytecode.

JavaScripts primitive data types only provide number values, instead of dedicated floating-point and integer values. Specific views on typed arrays exist, but when using bitwise operations, JavaScript engines can also make use of integer types (Zakas, 2010, p. 156-158). This enables optimization techniques and made the standardization and implementation of WebAssembly possible (Wagner, 2014). Using the pipe operator in JavaScript, numbers can be coerced into integers, resulting in performance improvements in certain scenarios.

2.9.4 WebAssembly

WebAssembly (WASM) is a binary instruction format designed for safe and efficient execution of code in the same runtime environment as JavaScript, which enables near-native performance for web applications (Hunter and English, 2021, p. 155; Jangda et al., 2019, p. 107-118). Code from languages such as C, C++, and Rust can be compiled into WASM (using emscripten or wasm-bindgen). The main performance benefit compared to JavaScript is that WASM is already available in binary format and the engine can therefore skip parts of the in section 2.9.3 mentioned optimization steps (Wagner, 2014). In the context of web parallelism optimizations, WASM can improve the execution times of computationally intensive tasks by leveraging integrations with Web Workers and SIMD. WASM generally increases speed of computations, which is not directly part of this thesis, and it will therefore not be discussed in more detail.

2.9.5 Browser Event Loop

By default, JavaScript provides a single-threaded, asynchronous, event-based environment in browsers (World Wide Web Consortium, 2024a, chap. 8.1.7; Hunter and English, 2021, p. 6). Compared to classic programming languages and environments, JavaScript execution is not subject to a preemptive scheduler (Silberschatz, Galvin, & Gagne, 2013, p. 263-264). It must return to the browser event loop in reasonable time, otherwise page rendering will stall (Zakas, 2010, p. 107-110). This is commonly referred to as "blocking the main thread" of JavaScript execution in literature (Hunter & English, 2021, p. ix). Browsers will notify the user about misbehaving pages and terminate them to restore responsiveness. Therefore, the proper way to architect a JavaScript application in most cases is to register event listeners and let the browser manage function calls, instead of creating an infinite loop on the main thread. High-frequency callbacks can be scheduled with e.g. the `requestAnimationFrame` API to implement fluent animations or other rendering tasks on the main browser event loop in pure JavaScript (ibid., p. 111).

The browser event loop utilizes a task queue for calling JavaScript function code sequentially (World Wide Web Consortium, 2024a, chap. 8.1; Haverbeke, 2018, p. 197-198; Archibald, Jake, 2015; Mozilla Corporation, 2024c). When events fire and registered listeners are called, or other JavaScript code from a `<script>` tag in the global context needs to be executed, the code is appended to the task queue. The task queue is processed until empty. When tasks are enqueued into an empty queue, they are processed immediately, which in terms of the event loop processing model can take up to 50 ms (World Wide Web Consortium, 2024a, chap. 8.1.7.3). Regular tasks are often created outside of the JavaScript application, e.g. through user input, I/O events such as a network resource fetch completing, timers firing, or new scripts being loaded.

Certain functions resolve in microtasks, that have their own microtask queue to improve responsiveness (World Wide Web Consortium, 2024a, chap. 8.1; Haverbeke, 2018, p. 197-200; Archibald, Jake, 2015). Microtasks are run between each task and processed until the microtask queue is empty. This allows for immediate execution of high-priority tasks, which keeps relevant data in CPU caches and reduces latency. Microtasks effectively capture and apply side effects of the previously run macro task, such as running code to follow up on a promise that has been resolved, responding to a MutationObserver callback, or handling work explicitly deferred via `queueMicrotask()`.

The event loop also behaves differently when it is associated with a window context, in contrast to that of a Web Worker (World Wide Web Consortium, 2024a, chap. 8.1; Mozilla Corporation, 2024c). Specifically, the event loop in a window context will defer to update and prune the DOM between execution of tasks, so that a consistent view on the state of the DOM is presented to JavaScript applications. Layout calculations are part of the rendering step. Effects of this strict ordering of events in browsers become visible in later parts of this thesis, where promises from multiple Web Workers under test fire with extremely high frequency and need to be processed on the main (window) event loop.

2.10 Web Workers

The limitation of potentially blocking the event loop when performing computationally expensive work on the main thread is a major disadvantage compared to desktop environments. Web applications can use techniques to limit the time that the event loop is blocked by saving their state, and scheduling their next work block using `setTimeout()` for example (Hunter & English, 2021, p. 6). This enables execution of tasks that require long processing times while maintaining reasonable responsiveness of the site but does not solve throughput and latency limitations in the main event loop (Haverbeke, 2018, p. 197-200).

Web Workers were added to the web standard in 2009 and offer a way to create separate, isolated threads that can be communicated with through message passing (Zakas, 2010, p. 120-124; World Wide Web Consortium, 2024a, chap. 10.1). Workers enable execution of computationally expensive and time-consuming JavaScript code in the background and independently of the main event loop. Functionality to share memory between the main thread and Web Workers only arrived in 2018 and was almost immediately disabled again due to the discovery of Meltdown and Spectre CPU vulnerabilities, for which shared memory and parallel code execution provided a highly accurate timing source (Wagner, 2018). SharedArrayBuffers were added back to web browsers in 2020, after mitigations for Spectre and Meltdown had reached operating systems and CPU microcode, and additional process-level isolation mechanisms were developed for web browsers.

Cross-Origin-Opener-Policy and Cross-Origin-Embedder-Policy headers set to “same-origin” and “require-corp” respectively ensure that only trusted JavaScript code sources can interact with each other’s processes, and thus, through side-channels, extract data via CPU vulnerabilities (Hunter & English, 2021, p. 75). Many parts of this thesis measure the performance of Web Workers in combination with SharedArrayBuffers, and hence will not work on browsers from the period of 2018-2020 without workarounds.

2.11 JavaScript Performance Measurements

The global `performance` object offers access to high-resolution timers, with slightly different resolutions depending on which browser is used, and whether a secure context is available (Mozilla Corporation, 2024a). Browsers limit timer resolution to 100 µs without a secure context to reduce the impact of attacks that exploit CPU side channels (World Wide Web Consortium, 2019, chap. 7.1), like described in the previous sections 2.2 and 2.10. Within a secure context the resolution is increased to 5 µs in Chrome and Safari clamps it to 20 µs (Mozilla Corporation, 2024a; Apple Inc., 2024). Firefox provides the settings `privacy.reduceTimerPrecision` and `privacy.resistFingerprinting` under `about:config`, which disable timestamp precision reduction.

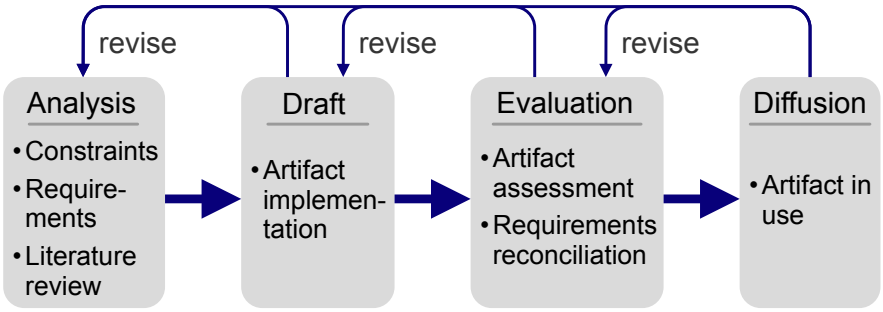
Timestamps captured via `performance.now()` are relative to the time at which the current site or worker was initialized (Mozilla Corporation, 2024a). `performance.timeOrigin` is a monotonic clock that represents the point in time at which the current context was initialized (Kerrisk, 2010, p. 492).

Using both timestamps allows for synchronization of workers and the main thread (World Wide Web Consortium, 2019, chap. 7.2). However, as later discussed in chapter 5.3.2, Chrome deviates from the specification resulting in clock skew due to wall clock time adjustments.

2.12 Design Science Research

To conduct performance benchmarks, an application that gathers the required data is needed. Design Science Research is a research methodology that focuses on developing a so-called artifact, i.e. the benchmark suite in this thesis (Benner-Wickner et al., 2020, p. 6-8; Österle et al., 2010, p. 664-669; Hevner and Chatterjee, 2010, p. 5). First, objectives are defined to address a problem with practical relevance. Next, an artifact is created, which is then evaluated for meeting previously specified requirements. The artifact is iteratively improved until it satisfies the practical criteria, so that it is fit for a practical purpose. Finally, when the artifact solves the specified problem, it should be made available for use (e.g. by publishing source code). These steps are visualized in Figure 1.

Figure 1: Design Science Research Steps



Source: Adapted (with changes) from: Benner-Wickner et al., 2020.

3 Related Work

3.1 Concurrency & Parallelism Strategies

Kiefer et al. (2015, p. 405-414) presented a case study on parallelizing a commercial real-time audio application, referred to as DJ Star. Typical workflow features include mixing and manipulating audio streams (e.g. applying audio effects). The paper analyzes the application's software architecture and runtime performance to identify the best parallelization techniques for data stream-intensive workloads.

Three parallelization strategies were proposed and evaluated: Busy-waiting, thread-sleeping, and work-stealing (Kiefer et al., 2015, p. 409-412). The busy-waiting strategy, though generally seen as inefficient, yielded the best performance with the lowest average execution times (327µs) and a schedule efficiency close to 99% on an eight-core machine (Kiefer et al., 2015, p. 413-414; Tanenbaum and Bos, 2015, p. 124). This strategy proved most effective due to the short, high-frequency processing cycles required by DJ Star, where the overhead of pausing and waking threads with the thread-sleeping strategy was too costly.

Key findings indicate that in the context of real-time and data stream-intensive applications such as DJ Star, busy-waiting can outperform traditional best practice strategies for parallelization (Kiefer et al., 2015, p. 413-414). The study provides valuable insights and guidelines for parallelizing similar performance-critical software.

3.2 Adoption of Web Workers

Many software products on the web use concurrent architecture patterns nowadays. Products such as Google Docs, Slack, Figma and Microsoft Office 365 all use Web Worker instances when inspected with browser developer tools.

In terms of an open-source solution that uses concurrency on the web, Partytown by Builder.io (2024) is an in-beta JavaScript library that addresses the problem of main thread slowdowns through third-party scripts, by offloading them to a Web Worker context. According to Builder.io, their implementation using Atomics has "10x faster communication between threads compared to [the] service-worker requests".

JetStream 2 is a benchmark suite containing a variety of synthetic JavaScript and WASM tests (Apple Inc., 2019). The tests target workload performance directly in contrast to the transmission focused benchmarks addressed in this thesis.

3.3 Web Worker Performance

Previous research regarding Web Workers explored performance gains and scalability in various scenarios. A study by Okamoto and Kohana (2011, p. 592-597) demonstrated that Web Workers can significantly improve the execution speed of computational tasks by offloading them from the main thread. Comparative analyses with single-threaded JavaScript execution reveal that Web Workers

provide a notable performance boost, particularly in applications requiring heavy data processing, as shown by the findings of Karltorp and Skoglund (2020, p. 16-26) and Wihuri (2023, p. 29-51). These studies underline the potential of Web Workers to enhance user experience by reducing latency and improving responsiveness.

Concurrency models play a crucial role in web development, enabling applications to handle multiple tasks simultaneously. Web Workers represent one such high-level model, but others, such as WebAssembly and Service Workers, also contribute to the variety of optimizations that can be implemented as part of concurrency. A comparative study conducted by Svartveit (2021, p. 60-87) shows that, while WebAssembly offers a superior performance for compute-heavy tasks, Web Workers provide a more straightforward approach to parallelism in JavaScript.

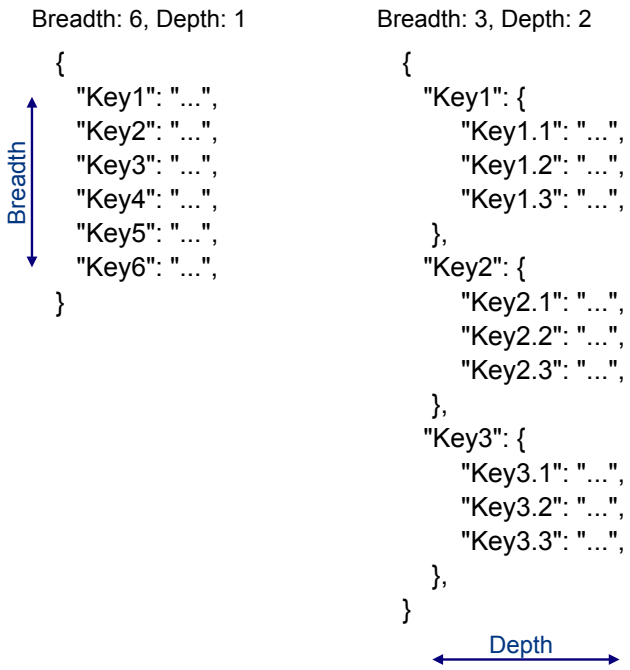
Effective task parallelism and work distribution are critical for maximizing the benefits of Web Workers. Research by Verdú and Pajuelo (2016, p. 105-108) explores how different workloads scale with worker amount. Additionally, worker execution models are classified into single worker, multiple synchronous, and multiple asynchronous workers. Their findings indicate that spawning more Web Worker instances than available cores leads to performance penalties. Whether or not an application window is focused, and other co-running applications exist, also plays a role in the performance observed (foreground process), given that a system manages only a limited amount of computing resources. They propose dynamic runtime analyses to find an optimal Web Worker amount for scenarios where the overhead impact on application performance matters (Verdú, Costa, & Pajuelo, 2016, p. 3525-3539).

Surma (2019b) investigated how the size of data sent to Web Workers impacts `postMessage()` durations. Data transfer benchmarks with different payload sizes were conducted using various browsers, including Chrome, Safari, and Firefox, across a set of devices. However, the tests only consist of workloads with one single Web Worker. The results revealed that object breadth & depth, as shown in Figure 2, and message size significantly impact transmission times. Specifically, messages up to 100 KiB typically fit within the 100 ms response budget associated with immediate user interaction tasks (Google LLC., 2020). However, for JavaScript-driven animations, which should occur every 16 ms, only messages up to 10 KiB are considered risk-free.

To address performance issues, the article suggests several optimization techniques (Surma, 2019b). One method is patching, which involves sending only changed data instead of entire state objects. Another technique is chunking, where large patch sets are broken into smaller chunks and sent sequentially. In addition, the use of shareable datatypes, such as ArrayBuffers, could enhance performance since they do not require copying and could be transferred more quickly. The article also mentions the potential of WebAssembly for efficient serialization and deserialization.

Surma provides the relevant source code for data and plot generation on GitHub (Surma, 2019a). All benchmark iterations are executed 100 times, with 5 additional warmup executions that are discarded. The data that is part of the payload objects and can be of type boolean (30%), a numeric type (30%)

Figure 2: Object Breadth & Depth



Source: Adapted (with changes) from: Surma, 2019b.

or a string (40%). When trying to reproduce the benchmark results, some discrepancies became apparent. The plots show payload sizes as KiB and MiB but in the source code corresponding byte value calculations are divided by 1000, which converts it to KB and MB respectively. Furthermore, the data quantities shown in the plot area are difficult to interpret and seemingly do not exactly relate to the data effectively generated. For example, at an object breadth of five and depth of six, the generated data should be 1 MB according to the plot, however, 1.7 MB are generated using the benchmark.

Lawson (2016) created a test suite comparing `postMessage()` performance with strings and objects. The results show that it is faster to send messages with stringified objects rather than actual objects. However, these tests predate Surma's experiments by over 3 years and browser implementations seemingly have changed in that time, given that Surma came to a different conclusion (Surma, 2019b).

In the context of this thesis, the previously mentioned solutions do not yield the necessary test results to answer the research question. Okamoto and Kohana (2011), Karltorp and Skoglund (2020), Wihuri (2023), and Svartveit (2021) improve performance by relocating costly computations off the main thread onto separate threads. These performance improvements focus on gains for specific workload scenarios by applying parallel computing as a paradigm and are due to the lack of granularity not necessarily generalizable for other workloads, as those can have different throughput and latency requirements. Surma's (2019b) approach was considered for throughput benchmarks but the deficiencies with unit conversions, mixed data types in objects, discarded warmup runs, fixed run iterations, and limitations regarding worker amounts, made it not a suitable solution. The tests that Lawson (2016) conducted stringify and parse the same payload data for each iteration, which makes it possible that the measurements are biased due to cached values (Zakas, 2010, p. 77-79).

4 Methodology

4.1 Research Design

To answer the research question of how to achieve low latency and high throughput in the context of real-time web applications using Web Workers, experiments in form of (micro)benchmarks are developed. The benchmark application is developed according to the Design Science Research methodology and represents an artifact with practical relevance (Benner-Wickner et al., 2020, p. 6-8; Hevner and Chatterjee, 2010, p. 5; Lindner, 2020, p. 38-39). Comparing data transfer techniques for different types of workloads is the primary goal, without benchmarking specific workloads themselves, as those can vastly differ in terms of their computational requirements and are susceptible to minor implementation details. Based on the previous chapters, the benchmark web application needs to be able to test several worker amounts, and categories with a variety of parameters to find potential bottlenecks and explore scaling characteristics.

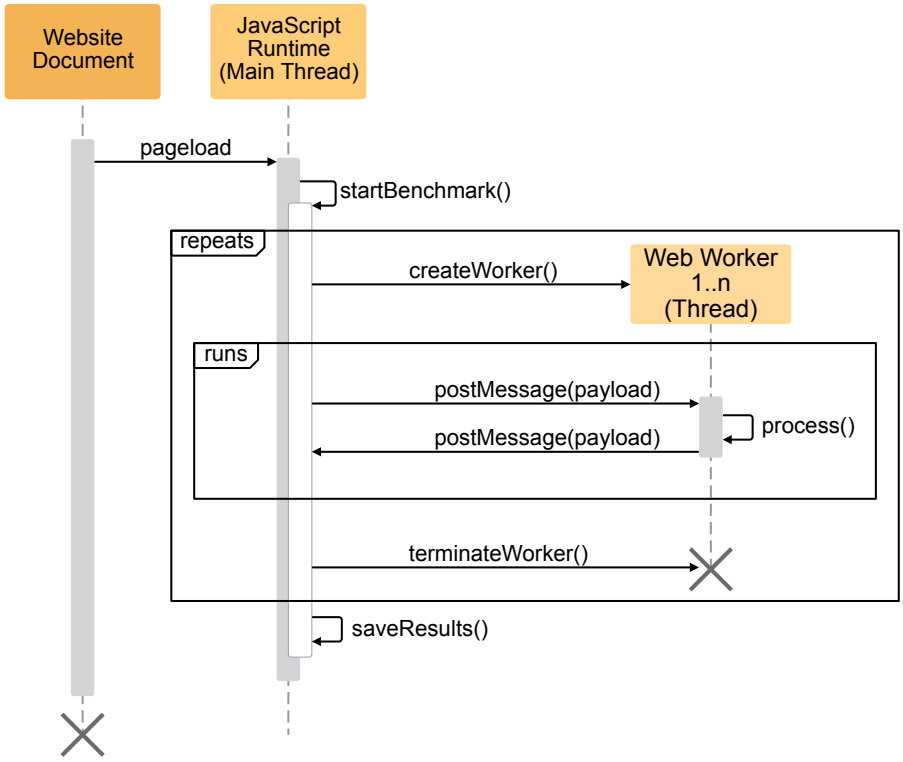
The main metrics considered for testing in the context of the research question are throughput and latency. Five benchmark categories are of interest based on potential insights regarding data structures and control flows in JavaScript runtime-environment implementations, and usefulness for general application architecture design:

- Binary Large Object (BLOB): Generic payload for evaluating throughput in binary formats
- Floating-Point: Number based payload for computations
- Object: Structured data that can vary in complexity
- Latency with Barrier: Response timings for synchronized workloads
- Latency without Barrier: Response timings for independent workloads

Payload refers to the data that is transmitted for each transmission iteration. Strings with random characters of specified size are generated for BLOB payloads and transmitted using the chosen transmission method. On the worker side a new string is generated before a message is sent back. Transmission methods that can be compared are postMessage copy, postMessage transfer, and binary encoded data that is stored in SharedArrayBuffers with an atomic notification mechanism. These operations happen concurrently with the number of workers that are requested in a test. Float payloads generate either an array, or object with the specified number of floats. These values are also randomly generated and again copied via postMessage or written into the shared memory region used with atomics. An option for randomizing object keys exists, which impacts JavaScript engine optimizations, as explained in chapter 2.9.3. Object tests generate data with the requested breadth & depth and copies it via postMessage invoking `structuredClone()`, or as a binary encoded string to workers. The binary encoded string representation can also be tested with SharedArrayBuffers and atomics. Again, an option for randomizing object keys exists. Latency tests do not generate payload data but use different synchronization techniques. This is described in more detail in section 4.2.

A run, also called roundtrip, consists of one transmission to a worker and one transmission back from the worker to the main thread. The specified number of runs for a worker happen consecutively. Repeats loop over the number of runs. Figure 3 shows a sequence diagram that visualizes the order of operations. While defining the first test iterations, 1000 runs and 10 repeats were initially planned for each test. The test repeats should help to evenly distribute outlier values, and inaccuracies caused by browser fingerprinting protections, as mentioned in chapter 2.11. After conducting early development tests, the numbers of test cycles were adapted so that the benchmarks can be reasonably carried out within the two-month time frame available for this thesis. Repeats were reduced to 5 and runs down to 500 due to the tests with the largest payload parameter sizes taking almost 12 hours to finish for one browser on a Mac Mini M1. Because of longer transmission times caused by increased data complexity, object tests were reduced to 300 runs. Latency tests were able remain at 10000 runs as there are no calculation-wise costly payload data sent.

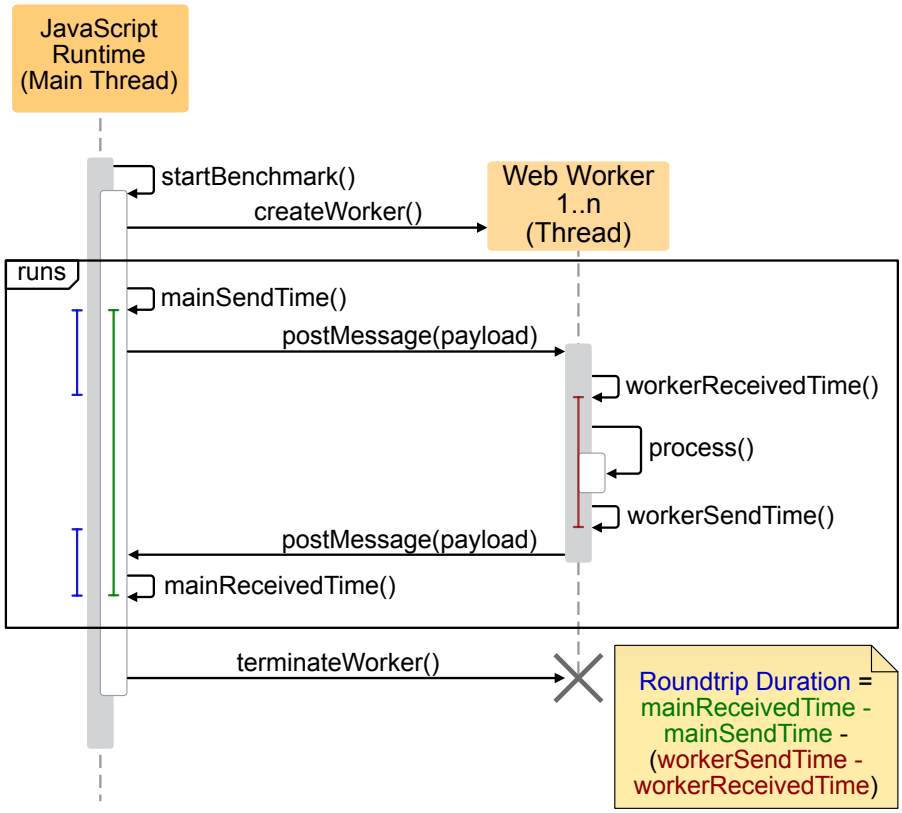
Figure 3: Worker Repeats and Run Sequence



Source: Own representation

All performance measurements are taken between sending and receiving payload data, with data generation times subtracted from the total roundtrip duration as show in Figure 4. Likewise, the cost of serialization is excluded from measurements (where possible), as data serialization/de- & encoding is dependent on a variety of factors, including data types, algorithms, memory layout, and architectural application choices. JavaScript runtime environments alone offer several options (e.g. `JSON.stringify()`, `JSON.parse()`, `btoa()`, `atob()`, `charAt()`, `codePointAt()`, etc.).

Figure 4: Roundtrip Duration Measurement



Source: Own representation

The data needs to be statistically evaluated after collection. This is done via an automated script using R (The R Foundation, n.d.). Automating the evaluation improves reliability, objectivity, and validity, and therefore adheres to quantitative quality criteria (Lindner, 2020, p. 6; Creswell, 2009, chap. 8). Findings and inductive hypotheses are discussed in chapter 5.

4.2 Performance Metrics and Test Categories

For a detailed evaluation, throughput and latency were chosen as the most meaningful metrics to be compared. These metrics are the fundamental performance objectives for which the benchmark workloads must be optimized (Smith, 1993, p. 523; L. G. Williams & Smith, 1995, p. 88-89). Workload definitions are categorized by data types and workload specific parameters that can be scaled for each scenario, as well as other general benchmark parameters such as worker amount, shuffled or serial dispatch order of workers, and randomized or deterministic object keys where applicable.

The methods by which data can be sent are either event-based messages (postMessage), or atomics-based notifications on shared memory. These data communication primitives require different worker setups described more in depth in chapter 4.4.3. While `postMessage()` represents a high-level interaction interface between threads, providing implicit data serialization out of the box, and the option to transmit data by reference or by value using the `structuredClone()` algorithm, atomics offer a low-level synchronization primitive that allows for a software architecture designs akin to native platforms.

The latency tests were split into synchronous (i.e. with barrier) and asynchronous (i.e. without barrier) benchmarks (Pacheco, 2011, p. 176). In the synchronous scenario all workers wait for the current run to finish before moving on to the next run, whereas asynchronous workers finish all runs as fast possible, independently of other workers. All workloads were tested with 1, 2, 4, and 8 workers. The combination of methods and categories results in the following workload tests:

1. BLOB (Variables: worker amount; payload size)
    - Atomics Binary Encoded SharedArrayBuffer Blob [variations: 28]
    - postMessage Blob Copy [variations: 28]
    - postMessage Blob Array Transfer [variations: 28]
2. Object (Variables: worker amount; payload breadth and depth)
    - Atomics Binary Encoded SharedArrayBuffer Object [variations: 144]
    - PostMessage Binary Encoded Object Copy [variations: 144]
    - PostMessage Object Copy (structuredClone; randomized and deterministic keys) [variations: 288]
3. Float (Variables: worker amount; float amount)
    - Atomics Float SharedArrayBuffer [variations: 20]
    - PostMessage Float Array Copy [variations: 20]
    - PostMessage Float Object Copy (structuredClone; randomized and deterministic keys) [variations: 40]
4. Latency with Barrier / synchronous (Variables: worker amount; serial and shuffled worker dispatch)
    - Atomics [variations: 16]
    - PostMessage [variations: 16]
5. Latency without Barrier / asynchronous / “Speed” (Variables: worker amount; serial and shuffled worker dispatch)
    - Atomics [variations: 16]
    - PostMessage [variations: 16]

The input parameters to all test variations act as independent variables and roundtrip durations are the dependent variables. There are 804 workload test variations for all categories, methods, and parameters combined.

4.3 Hardware and Software

Three devices and four operating systems were selected as execution environments. Not all browsers are available for all platforms, which is shown in Table 2. The chosen devices are between four and five years old and represent baseline setups rather than cutting-edge machines. This should help developers to optimize for a broader userbase, ensuring that applications run smoothly on older systems that many users may still operate. Two of the devices, the Mac Mini M1 and iPad 9th Gen

use ARM-based CPUs, whereas the Intel i7-8565U is x86-based, providing a mixture of architectures. All devices were connected to a power outlet and otherwise idle during measurements.

Table 2: Devices, Operating Systems and Browsers Used in Benchmark Experiments

	Chrome 126	Firefox 127.0.1 (no Atomics)	Safari 17.5
Mac Mini M1, 16GB LPDDR4X-4266MT/s SDRAM, macOS Sonoma 14.5	✔	✔	✔
Lenovo L390 Yoga i7-8565U, 16GB DDR4-2400MHz SDRAM, Windows 11 Pro 23H2	✔	✔	✘
Lenovo L390 Yoga i7-8565U, 16GB DDR4-2400MHz SDRAM, Ubuntu 24.04 LTS	✔	✔	✘
iPad 9th Gen A13-Bionic, 3GB LPDDR4X SDRAM, iPadOS 17.5.1	✘	✘	✔

Source: Own representation

Across all devices and browsers a total of 5760 benchmark tests were planned.

4.4 Benchmark Suite

4.4.1 Technological Choices

First, a simple synthetic benchmark was written as proof of concept based on the findings of the previous chapters. TypeScript is used as programming language, which is superset of JavaScript and gets transpiled using Vite. The source code is available at https://github.com/ch2i5/worker-bench. The build step changes source code to JavaScript by removing type annotations and changing some language features into ECMAScript compatible instructions. This approach should be representative of real-world applications, given that many large web application products nowadays are built using TypeScript (Stack Overflow, 2023).

The benchmark application is served on a virtual private server (VPS) and can be accessed at https://workerbench.cs3.dev/. The server utilizes the required CORS headers (Cross-Origin-Opener-Policy and Cross-Origin-Embedder-Policy) to provide a cross-origin isolated, secure context. This enables high resolution timers for all performance measurements and allows for the use of SharedArrayBuffers (Hunter and English, 2021, p. 75). Benchmark results can be downloaded as CSV-files.

4.4.2 Requirements and Features

The benchmark suite was iteratively improved based on new findings and requirements. After the first development tests it became apparent that a repeatable way of executing a broad variety of test definitions is needed, instead of entering the test parameters into a Graphical User Interface (GUI). Two options for automatic test execution via URL parameters were added: `testconfig` takes URL-encoded JavaScript Object Notation (JSON) objects or an array of objects and runs these instructions, whereas `testplan` takes a link to a JSON file containing an array of test definition objects, that is then executed as a test plan. After finishing a test and tearing down all involved workers, execution of the next test is postponed for one second so that the Garbage Collector has time to ideally clean up now unused, previously allocated resources, to reduce the impact on the next test. Examples for these URL parameters can be seen in Table 3. Valid config fields are documented in appendix A. The testplan definitions used in this thesis are available at the URLs listed in Table 4.

Table 3: Benchmark Automation URL Examples

Testconfig	https://workerbench.cs3.dev/?testconfig=[{ %22test%22:%22postMessageBlobCopy%22, %22repeats%22:0,%22workerAmount%22:2,%22runs%22:100, %22randomDispatchOrder%22:false,%22payloadSize%22:2048, %22download%22:true}]
Testplan	https://workerbench.cs3.dev/?testplan=https%3A%2F%2F workerbench.cs3.dev%2FfloatTestplan.json

Source: Own representation

Table 4: Benchmark Testplan URLs

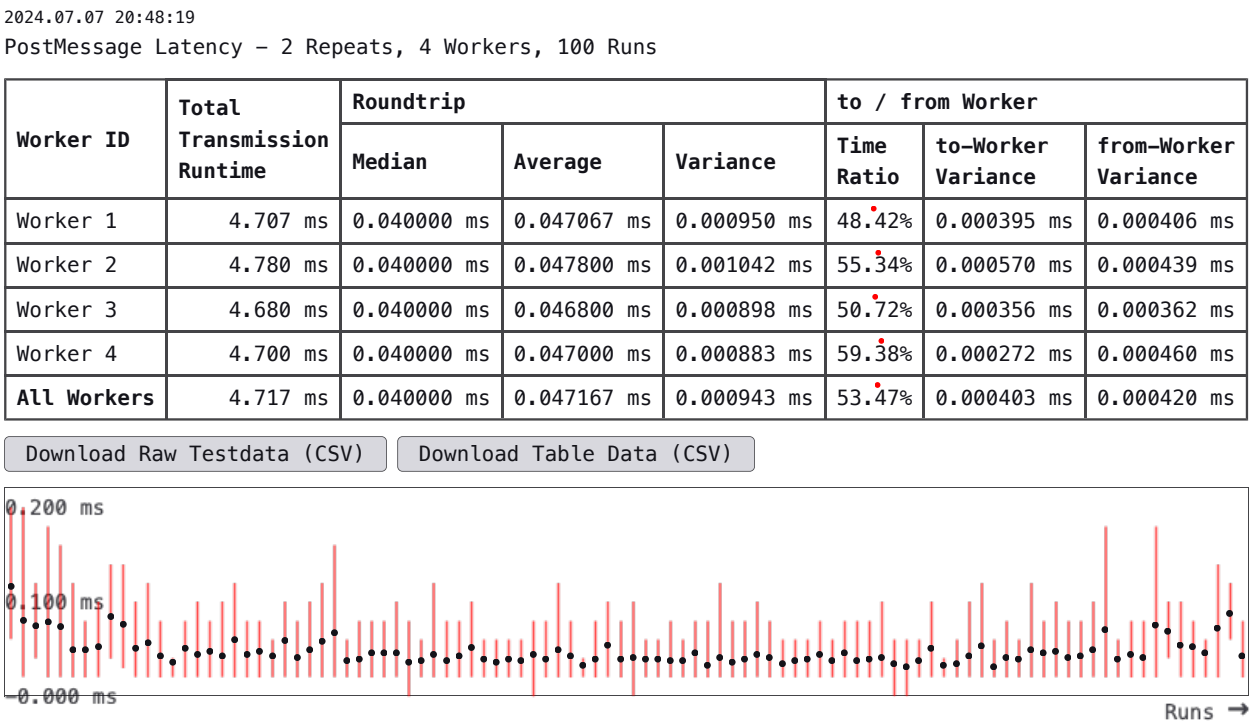
https://workerbench.cs3.dev/blobTestplan.json
https://workerbench.cs3.dev/objectTestplan.json
https://workerbench.cs3.dev/floatTestplan.json
https://workerbench.cs3.dev/latencyTestplan.json
https://workerbench.cs3.dev/speedTestplan.json

Source: Own representation

Due to the large number of long-running benchmarks, the test suite needs to be able to persist all data automatically and unsupervised. This functionality can be enabled using the `download` flag. The iPad's isolated filesystem proved to be a challenge for downloading test results on-device. To solve this problem, a CSV-endpoint option was added to the test parameters, where the test results are sent to via HTTP PUT requests. The endpoint needs to be able to accept these requests and create files on disk, for example by configuring it as Web-based Distributed Authoring and Versioning (WebDAV) folder.

Furthermore, an option for visualizing runs after test execution was added to the GUI, to verify data immediately. A table with aggregated data categorized by workers is also shown (see Figure 5). Location parameters from descriptive statistics are used as methods for interpreting roundtrip times. The median and average make it possible to get a quick overview of the measurements. The variance is calculated as it shows the statistical dispersion, and therefore how diverse the worker measurements were. In addition, the time ratio between sending and receiving is presented.

Figure 5: Benchmark Suite User Interface Excerpt



2024.07.07 20:48:19

PostMessage Latency – 2 Repeats, 4 Workers, 100 Runs

Worker ID	Total Transmission Runtime	Roundtrip			to / from Worker		
		Median	Average	Variance	Time Ratio	to-Worker Variance	from-Worker Variance
Worker 1	4.707 ms	0.040000 ms	0.047067 ms	0.000950 ms	48.42%	0.000395 ms	0.000406 ms
Worker 2	4.780 ms	0.040000 ms	0.047800 ms	0.001042 ms	55.34%	0.000570 ms	0.000439 ms
Worker 3	4.680 ms	0.040000 ms	0.046800 ms	0.000898 ms	50.72%	0.000356 ms	0.000362 ms
Worker 4	4.700 ms	0.040000 ms	0.047000 ms	0.000883 ms	59.38%	0.000272 ms	0.000460 ms
All Workers	4.717 ms	0.040000 ms	0.047167 ms	0.000943 ms	53.47%	0.000403 ms	0.000420 ms

Download Raw Testdata (CSV) Download Table Data (CSV)

Source: Own representation

4.4.3 Implementation Overview

In case of microbenchmarks, external factors that could influence test results should be minimized. System resources must be used accordingly as outlined in chapter 3.1, and code needs to be structured in a way that no unnecessary allocations and operations are utilized. To get reliable and consistent results, JavaScript optimization techniques discussed in chapter 2.9.3 are applied.

Before starting the benchmarks, objects for storing test results are created. Each workload test is designed in a way that no benchmark resource allocations occur during measurements, as depicted in Figure 6. However, synthetic workload allocations are made during benchmark execution, to mimic real world data usage. All workloads that transmit payload data (i.e. not latency tests) generate and allocate payload data twice per run. First before sending a message to the worker and the second time on the worker side before sending a message back. This data is newly calculated to prevent payloads from being cached. As mentioned in section 4.1 and shown in Figure 4, time spent processing is subtracted when measuring roundtrip durations.

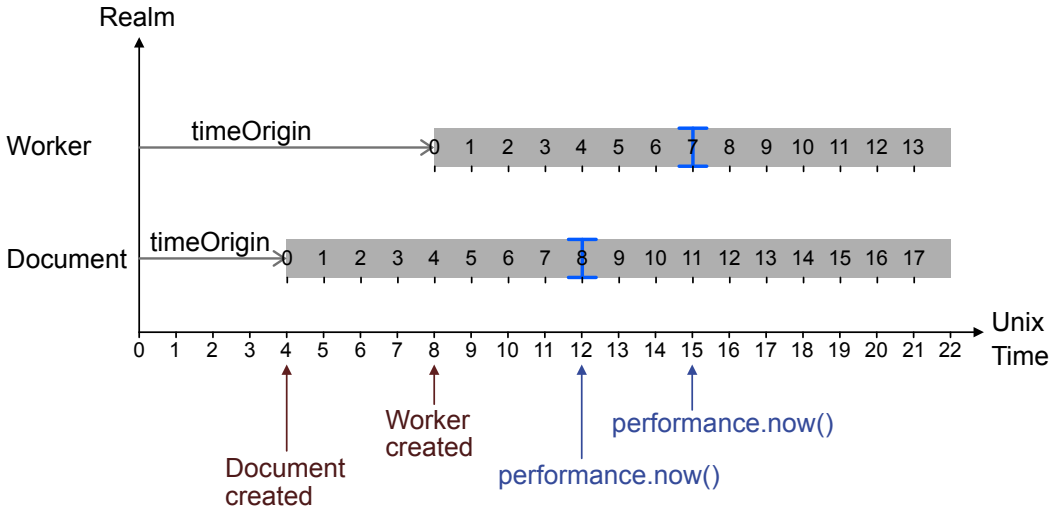
Figure 6: Start Benchmark - Pseudocode

```
function startBenchmark(config) {
    // allocate space for run results
    const resultSlots = new Array[config.repeats][config.runs][config.workerAmount];

    for (let currentRepeat = 0; currentRepeat <= config.repeats; currentRepeat++) {
        const runResultSlots = resultSlots[currentRepeat];
        await initializeBenchmark(config); // create and setup workers

        for (let currentRun = 0; currentRun < config.runs; currentRun++) {
            const workerResultSlots = runResultSlots[currentRun];
            await doBenchmark(config, workerResultSlots);
        }
        await teardownBenchmark();
    }
    saveResults(resultSlots)
}
```

Source: Own representation

As stated in chapter 2, when using `performance.now()` to get the current time, it is relevant to consider the realms/contexts `timeOrigin`, which in theory represents a monotonic clock that cannot be changed, so that all timings are correctly synchronized to one timeline (Kerrisk, 2010, p. 492). Figure 7 visualizes in a simplified way, how these time offsets relate to each other. Google, in practice, does not adhere to the World Wide Web Consortium (W3C) specification and instead of providing a monotonic clock, Chrome's `timeOrigin` is susceptible to wall clock adjustments (Google LLC., 2021b). The longer a web context exists, the greater the drift becomes. This can result in large negative measurement values. Due to the precision clamping in all browsers, small negative values can also occur when merging the timestamps into one timeline, if the differences in `performance.now()` measurements between execution contexts are small.

Figure 7: Document and Web Worker timeOrigin



Source: Own representation

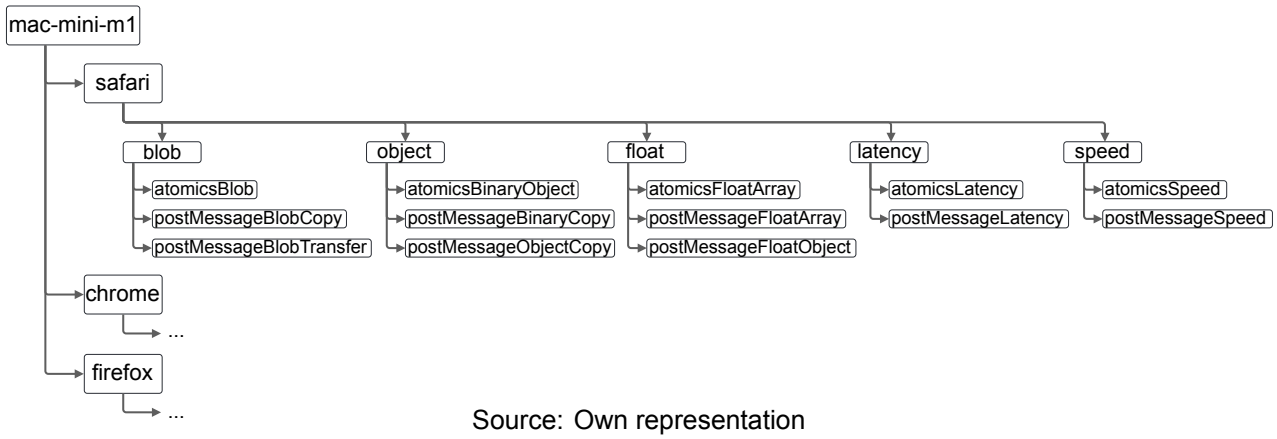
Each workload test implementation can be seen as a test harness responsible for managing worker resources, i.e. payloads, and communication between the workers and main thread. Dispatching workloads to the workers in the same order during development tests has shown that the last workers in the list have roundtrip durations more than twice as long compared to the first ones. This is likely due to the main thread task queue being bottlenecked from coordinating all worker communication. The `workerRandomDispatch` option was added to the benchmark parameters to control whether dispatch to the workers is randomized at every run, which results in evenly distributed runtime measurements (as later shown in Figure 18).

Figure 21 in appendix B shows a workload implementation example. Additionally, for atomics-based tests a SharedArrayBuffer is created once per worker on startup, which is reused for subsequent runs.

4.5 Analysis in R

An automatic evaluation pipeline was developed due to the amount of raw test data. Using a rudimentary implemented R script, which is available at https://github.com/ch2i5/worker-bench-results, the CSV output from the benchmark suite is aggregated and turned into plots. A bash script is provided for direct execution from the terminal via `makePlots.sh <devicename-folder>`. The script takes a parameter, which needs to be an existing folder name relative to the R script, ideally named after the device on which the tests were carried out. A new `graphs` folder will be created in the folder of the R script. The device folder needs to contain directories of the browsers that were used to execute the benchmarks. Inside the browser folders, test-type directories containing the benchmark CSV-files must be provided. Figure 8 shows the folder structure, adhering to the `test` parameter names listed in appendix A. The R script recursively scans for CSV-files inside the device folder and uses the path to the CSV, e.g. “mac-mini-1/safari/blob/atomicsBlob/filename.csv”, for test aggregation and interpretation of metadata.

Figure 8: Device Benchmark Folder Structure



Source: Own representation

The benchmark suite test files are named automatically but need to be placed accordingly into the corresponding directory. Depending on the type of test, the file names include different configuration

parameters. Timestamps at the beginning of the files are solely for archival purposes. The rest of the file name is relevant, as the R script parses it via Regular Expressions to determine how each file needs to be evaluated and aggregated. Table 5 shows some examples of generated file names. Plots are then created as Portable Document Format (PDF) files, so that they can be used without quality loss inside scientific papers, such as this one.

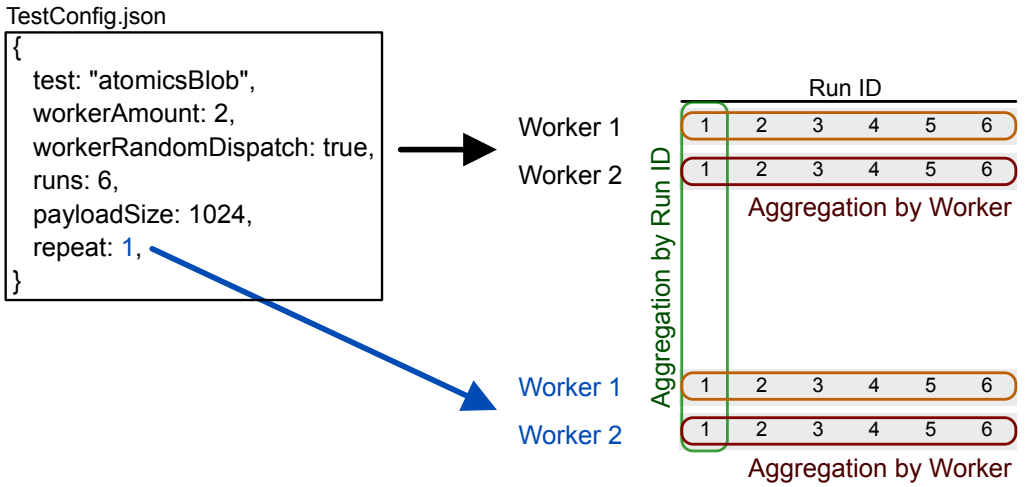
Table 5: Benchmark Test Filename Examples

Exemplary Filenames
`2024-06-19_21-28-45_Atomics_Blob_1MiB_5Repeats_4Workers_500Runs_ShuffledDispatch_Raw.csv`
`2024-06-19_16-52-46_PostMessage_Latency_5Repeats_8Workers_10000Runs_SerialDispatch_Raw.csv`
`2024-06-20_21-53-55_PostMessage_Object_Copy_B2D2L16_5Repeats_1Workers_300Runs_ShuffledDispatch_DeterministicKeys_Raw.csv`

Source: Own representation

Interpretations of the collected data in at least two dimensions can be of interest during the analysis process, as shown in Figure 9. Data can be evaluated per run or per worker basis. Aggregating by run ID allows for a regression analysis and gives insights into how test performance changes over time, e.g. in the beginning, middle, and end. Aggregating by worker shows how the different test executions and workers relate to each other. Additionally, the data aggregated on runs is filtered between the 5th and 95th percentile to remove outliers from the dataset. The abundance of data caused by five repeats (six total iterations) of each test should not lead to a significant loss of data despite the filtering. Therefore, outlier values, such as unexpectedly high values induced by system slowdowns and background tasks, and small negative values due to clamping, are assumed to have little impact on the analysis results.

Figure 9: Worker Data Analysis Dimensions



Source: Own representation

Since the data aggregated on workers is plotted on a logarithmic scale, the occasionally occurring negative values cannot remain as part of the dataset. One option to remove them, is to add the absolute value of the most negative datapoint to all points, shifting measurements into a positive range, while retaining distances between the points. However, this shift could potentially bias already positively biased measurements further. Interpreting those values as zero also biases the plots and can in test scenarios with fast runtimes and just one worker cause loss of detail (i.e. a horizontal line in the plot). Instead, for visualization purposes, values below zero are not plotted, which is also the default R behavior for logarithmic plots. The statistical methods for creating the plots are the same as in section 4.4.2 but applied through R base functionality.

5 Findings

5.1 Failed Tests

After running the testplans which took a time span of around two weeks for all devices and browsers, roughly 6 GB of raw data were generated. The first notable insight is that out of the overall 5760 planned benchmark tests, 5738 (99.62%) ran until completion. BLOB tests with big payloads failed on some devices/browsers, as documented in Table 6.

Table 6: Interrupted Benchmark Tests

Device	Browser	Test	Reason for Failure
iPad 9th Gen A13-Bionic, 3GB, iPadOS 17.5.1	Safari 17.5	postMessageBlobCopy 1 MiB, 2+ Worker	Range Error: Out of memory
iPad 9th Gen A13-Bionic, 3GB, iPadOS 17.5.1	Safari 17.5	All 10 MiB Tests	“This webpage was reloaded because a problem occurred.”
Mac Mini M1, 16GB, macOS Sonoma 14.5	Safari 17.5	postMessageBlobCopy 10 MiB, 2+ Worker	Range Error: Out of memory
Mac Mini M1, 16GB, macOS Sonoma 14.5	Chrome 126	postMessageBlobCopy 10 MiB, 8 Worker	Error Code: SIGILL
Lenovo L390 Yoga i7-8565U, 16GB, Windows 11 Pro 23H2	Chrome 126	postMessageBlobCopy 10 MiB, 8 Worker	Error Code: SIGILL
Lenovo L390 Yoga i7-8565U, 16GB, Ubuntu 24.04 LTS	Chrome 126	postMessageBlobCopy and postMessageBlobTransfer 10 MiB, 8 Worker	Error Code: SIGILL

Source: Own representation

The 10 MiB tests on iPad crashed the benchmark tab, briefly showed an error message, and refreshed the page. Safari on the Mac Mini stopped the benchmark execution as it ran out of memory and threw an “Out of memory” error, however, the page stayed responsive. Without begin able to validate the crash reason, developer comments in the WebKit bug tracker (Jackson, 2022, comment 14) indicate that there is a memory limit per Safari tab and the operating system does not swap memory, as described in chapter 2.8.1. In Google Chrome the benchmark tab crashed and showed a SIGILL error, i.e. the process executed an illegal instruction (Kerrisk, 2010, p. 391). Using `dmesg` on Ubuntu 24.04 LTS to look at the kernel log printed: `traps: DedicatedWorker [30455] trap invalid opcode ip:62fb70ad44c8 sp:74239fbf3b90 error:0 in chrome[62fb6a3be000+b2e4000]`. Firefox finished all planned tests, apart from not implementing `Atomics.waitAsync()` and therefore not being able to run the Atomics-based testplans, which were not the ones that crashed the other browsers.

5.2 Benchmark Results

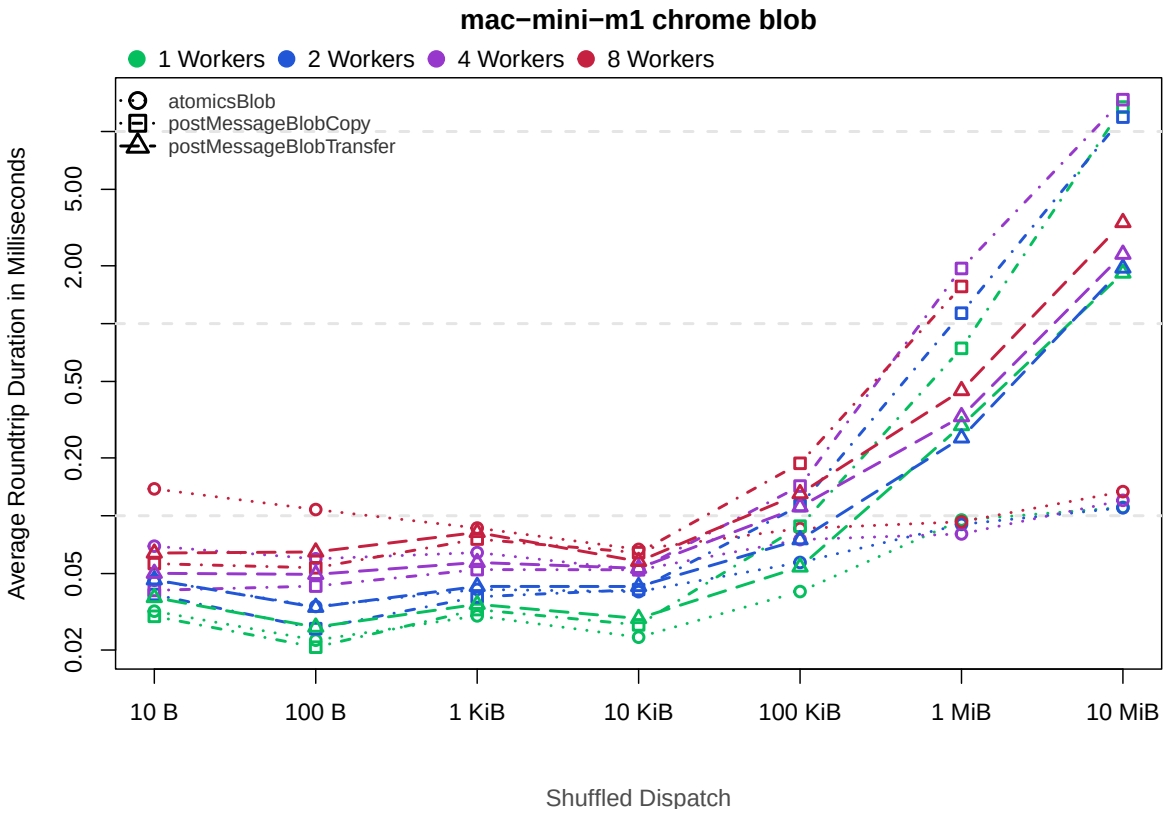
5.2.1 Presentation of Results

As the web platform is a highly abstracted application environment with many parameters that influence timings, such as Garbage Collection, general trends are presented in the following sections instead of specific values, as these are of greater importance for architectural web application design. The trends discussed in the following sections are representative across device and browser data sets. Displaying all evaluated data requires too much visual space and aggregating the data any further would lead to a loss of accuracy in the representation, therefore individual devices are used as examples below. All plots and raw data are provided in the GitHub repository located at https://github.com/ch2i5/worker-bench-results.

5.2.2 BLOB Performance

The postMessage copy results with BLOB data are consistent to the findings that Surma (2019b, chap. “Benchmark 2”) published. Up to a payload size of 100 KiB, postMessage durations are generally safe to be used in a user interaction context (Google LLC., 2024a). When comparing postMessage BLOB transfer and copy, the transfer operations up to 10 KiB were similar but became faster when transmitting more data across all devices and browsers, illustrated by the Mac Mini M1 examples in Figure 10. This behavior is to be expected, due to the faster nature of transferring ownership instead of copying memory.

Figure 10: Mac Mini M1 Chrome BLOB Measurements

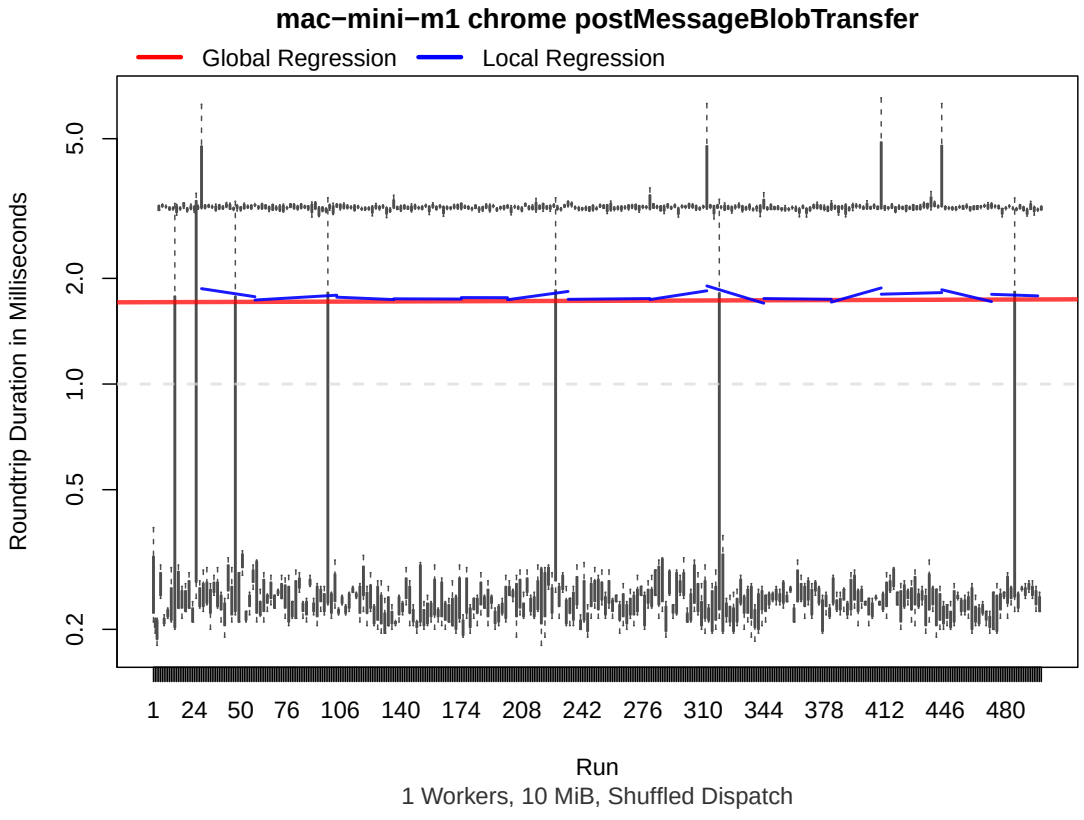


Source: Own representation

Transmissions using shared memory and atomics with up to 100 KiB payloads end up being minimally slower than postMessage copy and transfer, which is in line with the findings by Kiefer et al. (2015), mentioned in chapter 3.1, where thread-sleeping is not the most efficient method when working with high frequency, concurrent workloads. However, especially for transmitting larger payloads above 100 KiB, atomic based roundtrip durations only grow slightly with the amount that the execution environment is taxed, whereas postMessage copy timings grow linearly to the amount of data transferred, i.e. in $O(n)$, and end up being slower than the atomic tests.

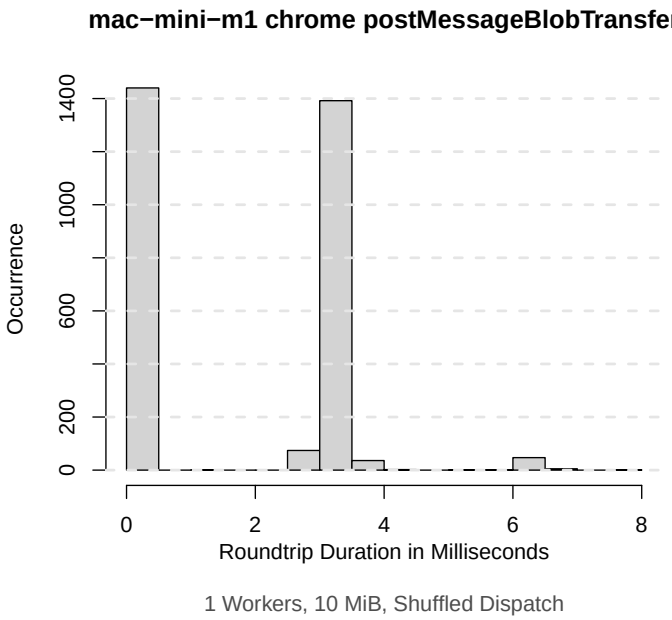
When transferring 10 MiB payloads via postMessage to one or more workers, the durations show an alternating delay pattern between runs, which is most pronounced in Chrome. Figure 11 shows the pattern during runs and the distribution is shown in more detail in Figure 12. The Chrome profiler shows major Garbage Collection events at the time of transition from the main thread to the worker contexts and the other way around that correlate with the delay timings.

Figure 11: Mac Mini M1 Chrome BLOB Transfer Alternating Delay



Source: Own representation

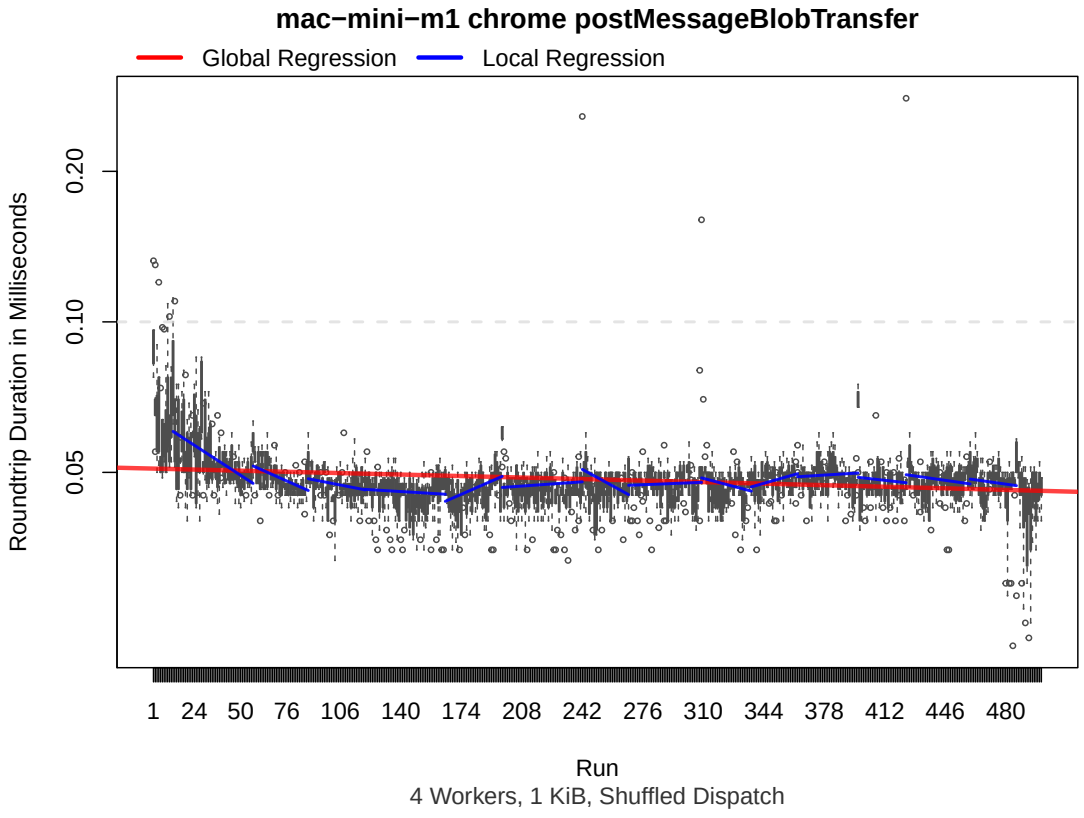
Figure 12: Mac Mini M1 Chrome BLOB Transfer Alternating Distribution



Source: Own representation

When looking at smaller run-based results, as exemplified by the 1 KiB payloads in Figure 13, transmissions get marginally faster throughout the runtime, after an initial increase in speed, enabled by JavaScript optimizations (see chapter 2.9.3).

Figure 13: Mac Mini M1 Chrome BLOB Bytecode Optimization



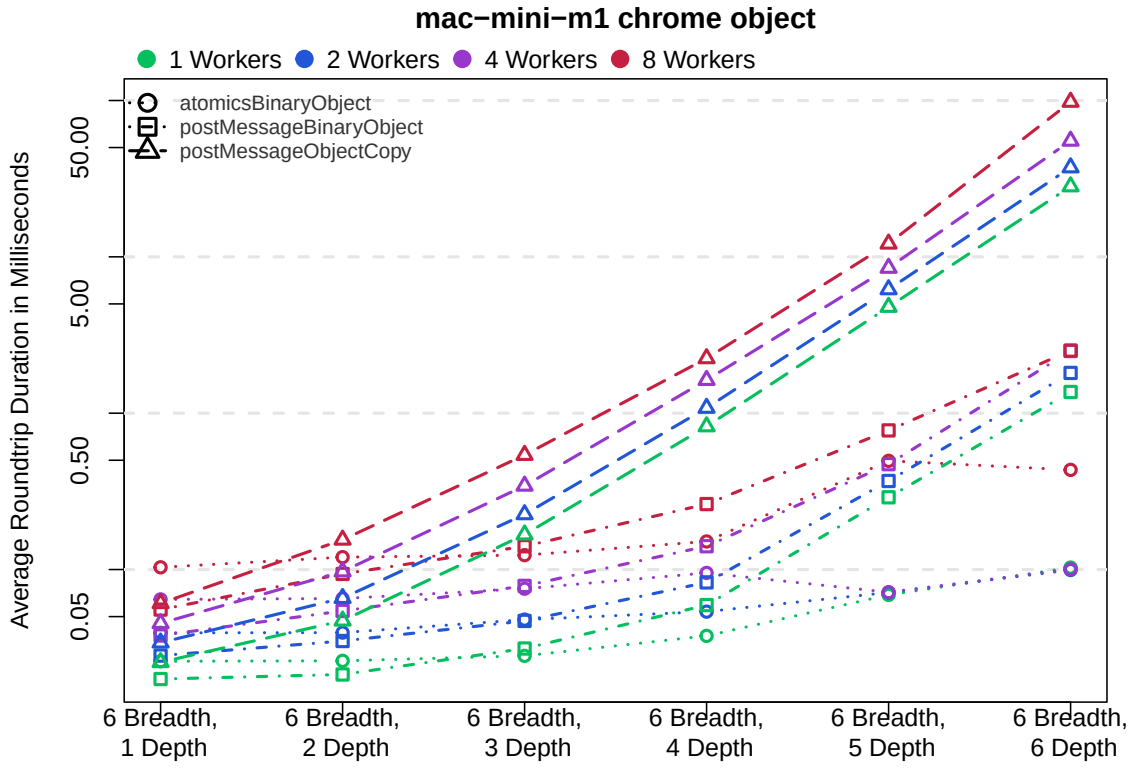
Source: Own representation

5.2.3 Object Complexity

The object workload tests handle more complex data structures compared to just binary information, though the individual data values contained inside all objects are of type string. Compared to the BLOB workloads, this gives insights into how transfer times scale with object complexity when sent between isolated instances and thereby highlights how `structuredClone()` serialization inside nested objects impacts the performance in contrast to only on the most outer level serialized, binary encoded object transmissions (chapter 4.2; World Wide Web Consortium, 2024a, chap. 2.7.10 and 2.7.7). Serialization costs for binary encoding the objects was excluded from the tests, as stated in chapter 4.1, due to different serialization techniques being their own research field and otherwise adding too many variables to this research topic.

Figure 14 shows that copying the objects via postMessage takes the longest again when working with large and complex data, confirming the BLOB results. The performance difference between sending a binary encoded object and a regular object is minimal, when working with shallow objects. By referencing the other breadth plots in appendix C or in the GitHub repository, it can be inferred that object breadth does not influence transmission performance in a noticeable way, at least on a depth level of one and up to a breadth of six. When increasing the depth, object copy transmission durations grow linearly with the amount of data that is transferred. Binary encoded object transmission timings also grow linearly, but as mentioned in the previous paragraph, serialization costs are not included.

Figure 14: Mac Mini M1 Chrome Object Complexity - Breadth Number 6

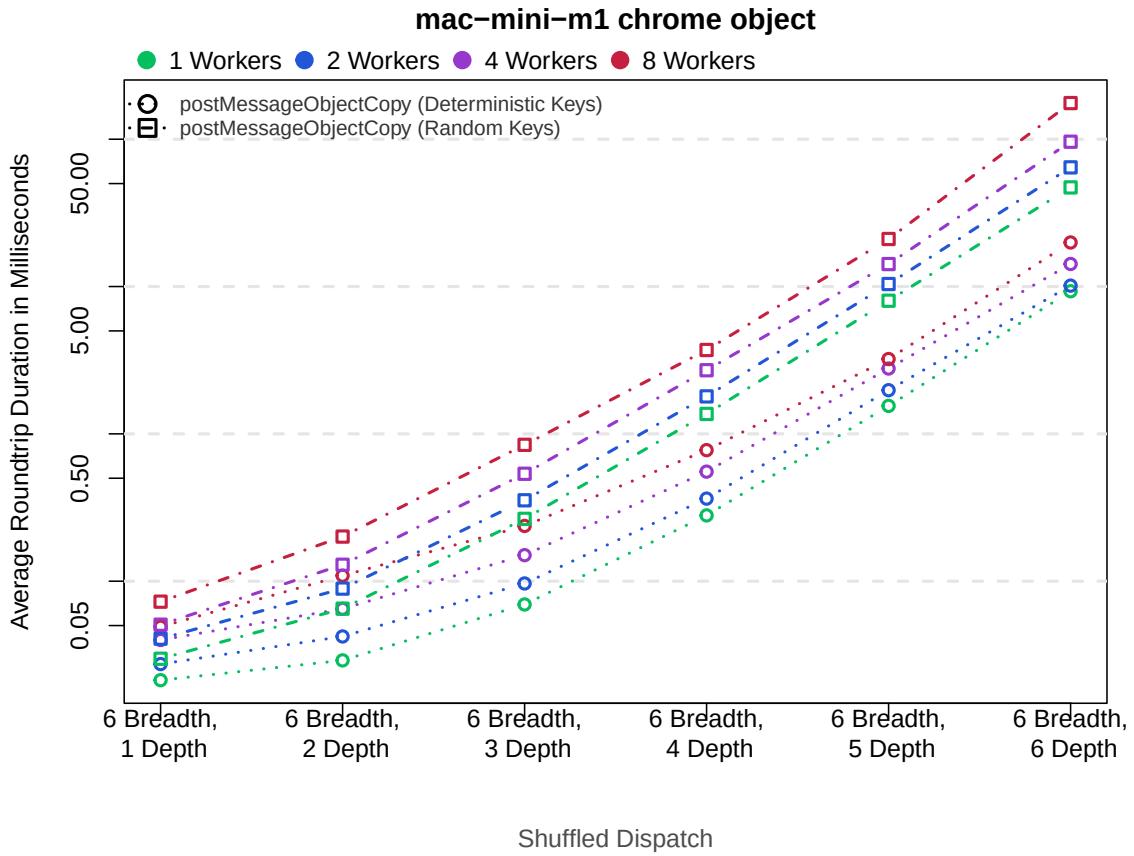


Source: Own representation

According to Surma's comparison (2019b, chap. "Benchmark 2"), the difference in transmission durations between copying regular and binary encoded objects and in Figure 14 should be the `structuredClone()` serialization times. Atomics with shared memory offer the most throughput again at the most complex test parameter settings. At a breadth of six and beyond a depth level of three, which corresponds to a data size of around 9 KiB, the cost of postMessage becomes greater than the overhead of atomics. Direct values of the object complexity measurements taken here are not comparable to the results that Surma (2019b) published, because those objects contain mixed data types (see chapter 3.3).

Furthermore, application architecture design that utilizes JavaScript engine optimizations benefits significantly from better performance, which is demonstrated in Figure 15. As mentioned in chapter 2.9.5, JavaScript engines optimize bytecode based on object shapes, which leads to a higher throughput when data is structured in the same schema. Appendix D shows that with the same object complexities and test parameters, a transmission with deterministic keys takes on average around 15 ms, whereas the random key variant needs roughly 95 ms per roundtrip.

Figure 15: Mac Mini M1 Chrome Object Keys

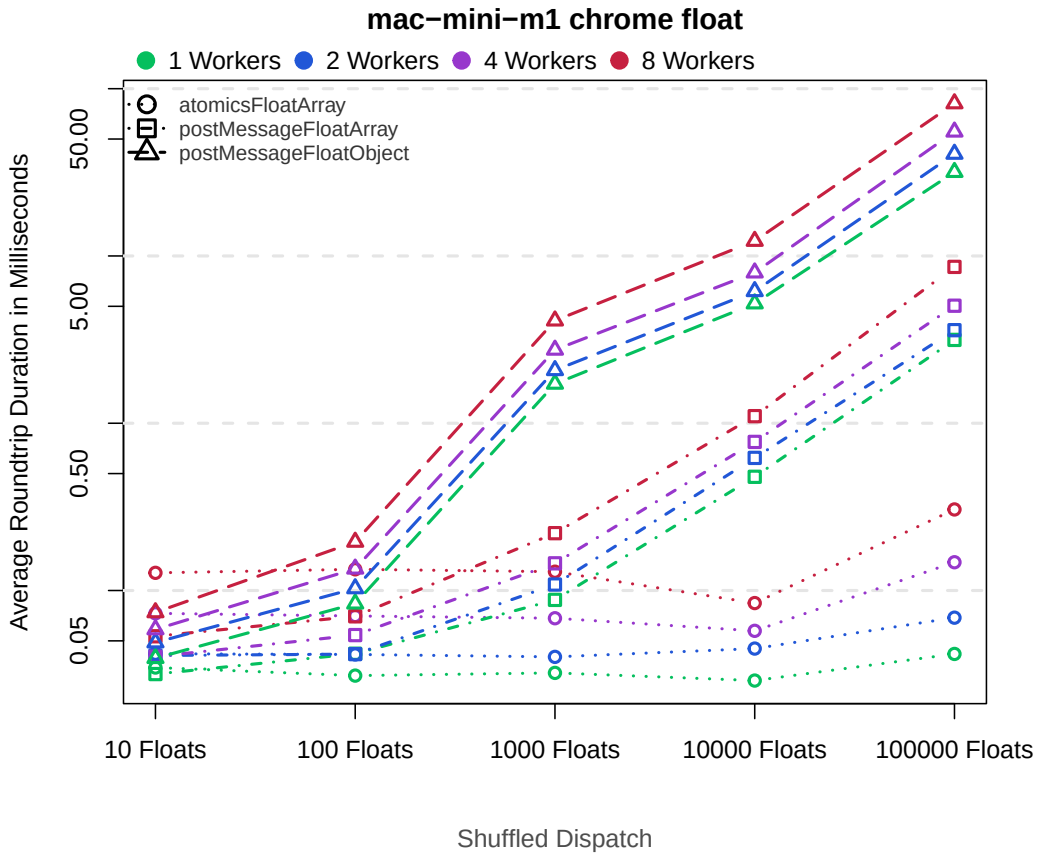


Source: Own representation

5.2.4 Floating-Point Throughput

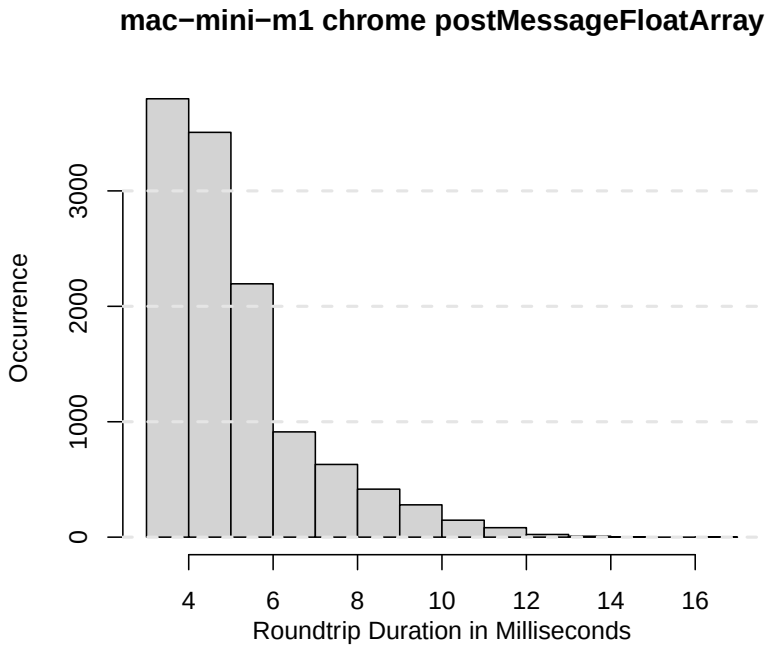
Floating-point values are commonly used in a variety of workloads, and optimizing their throughput between Web Workers and the main thread can increase overall application performance by leaving more time for calculations. Figure 16 compares the benchmark test results and confirms the trends from previous workload tests, as they also apply to float transmissions. At a small number of payload values the differences between methods are again minimal, with atomic notifications having slightly more overhead, but between 100 and 1000 floats, the object copy becomes significantly slower. Copying an array instead of an object that contains more than 1000 float values is faster by a factor of roughly 10. Atomics and SharedArrayBuffers again have the best performance characteristics at a high value count and do not rely on any form of serialization during runtime as floats are a primitive data type. The tests with larger amounts of floating-point values are at a scale, where it safe to assume that they exhaust CPU caches. This is likely shown in the increased variance of measurement durations, visualized in Figure 17.

Figure 16: Mac Mini M1 Chrome Floating-points



Source: Own representation

Figure 17: Mac Mini M1 Chrome Floating-Point Variance



Source: Own representation

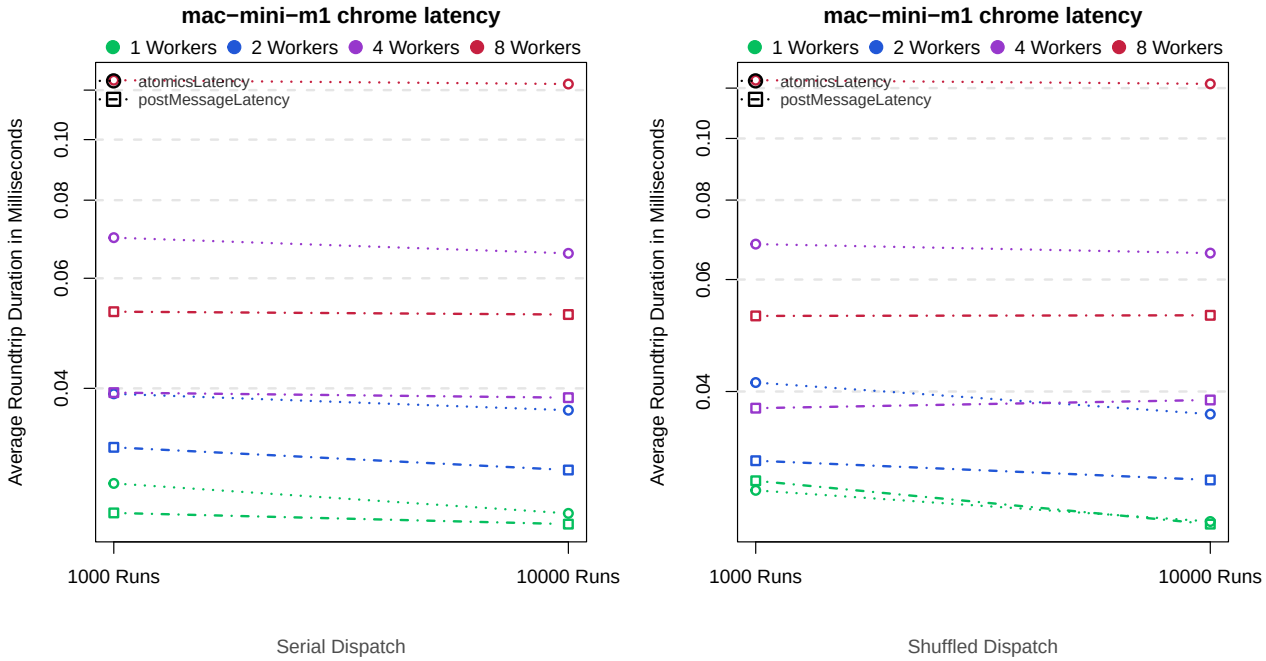
5.2.5 Latency Implications

The latency test design with and without barrier (synchronous and asynchronous) shows how fast a roundtrip or rather ping between both execution contexts can happen, without including a considerable payload. The synchronous tests adhere to all workers run boundaries, whereas the asynchronous tests execute all runs individually for each worker in succession. Tests with 10000 runs were included, so the effect of longer runtimes can be seen. The aggregated worker data in Figure 18 shows that average roundtrip durations tend to get faster the more runs there are, as long as the thread amount does not exceed the capacity of the available hardware resources. However, Figure 19 indicates that after the initial JavaScript engine optimizations the performance stays constant. Therefore, the fewer runs a test has, the higher the impact of non-optimized runs.

Figure 18 also shows that a serial task dispatch to workers causes a slightly steeper curve on the graph, which suggests a larger change in measurements throughout the runtime. By using a shuffled task dispatch strategy, the execution times become more evenly distributed (chap. 4.4.3).
`Atomics.notify()` takes overall longer than postMessage, confirming the other workload trends with minimal data transmission.

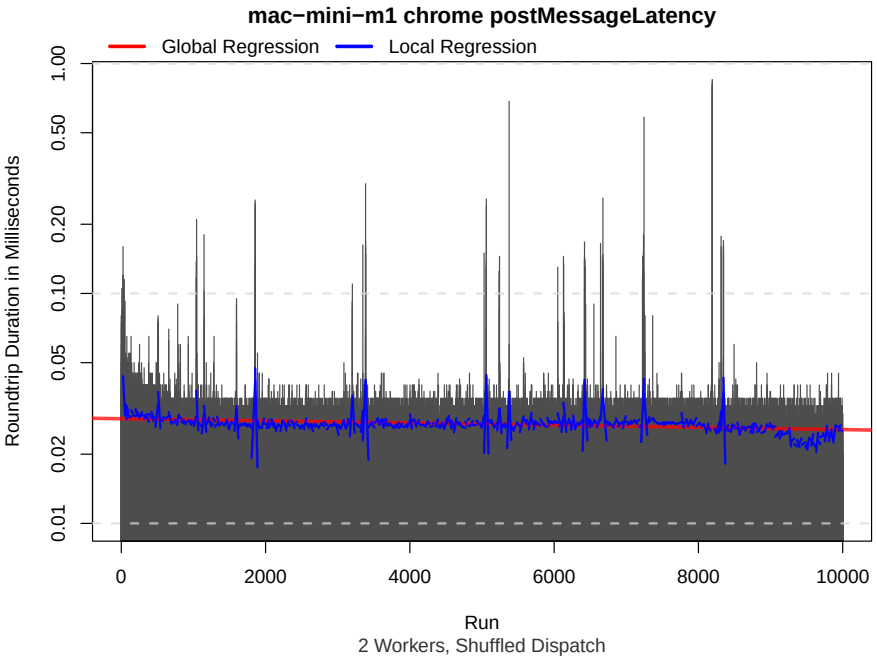
The local trend changes/spikes in Figure 19 could be caused by Garbage Collection events, or the main thread being otherwise busy.

Figure 18: Mac Mini M1 Chrome Synchronous Latency



Source: Own representation

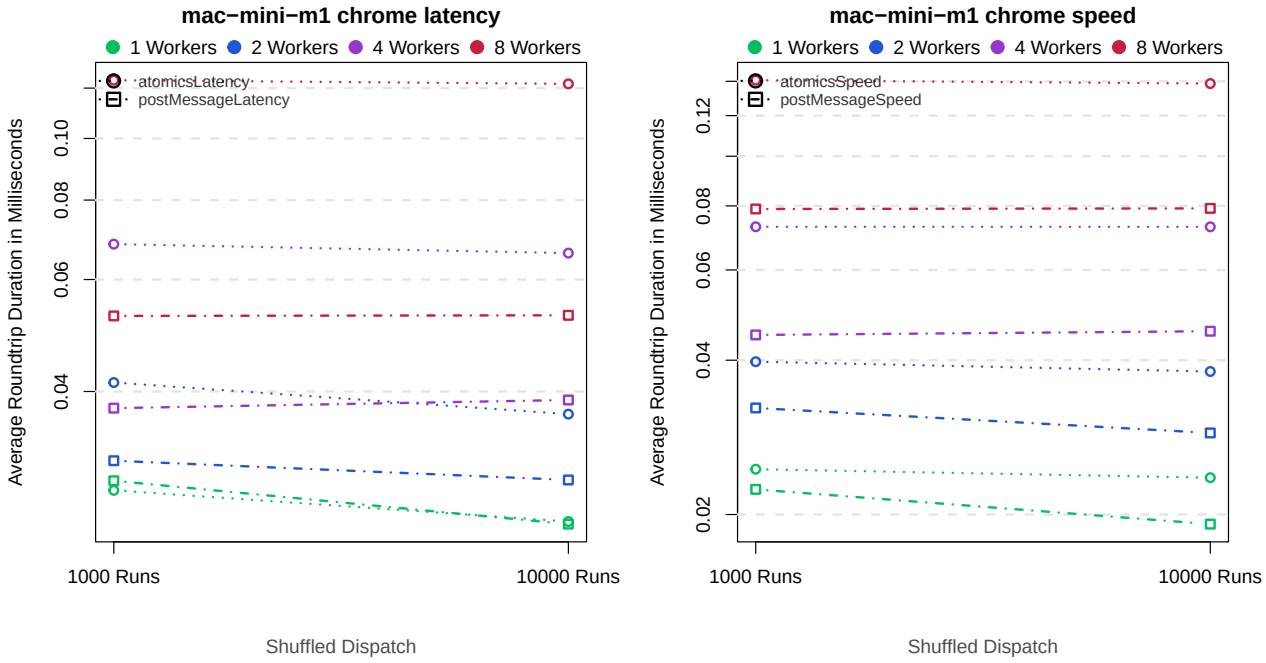
Figure 19: Mac Mini M1 Chrome Synchronous Latency Runs



Source: Own representation

The asynchronous tests, shown in Figure 20 on the right side, end up being slower in all cases, despite not having a barrier. This lack of a barrier is presumably the reason for the slowdown, as the main thread is preoccupied managing worker responses in short succession. The (micro)task queue likely accumulates several task entries in batches and therefore represents a bottleneck.

Figure 20: Mac Mini M1 Chrome Synchronous / Asynchronous Comparison



Source: Own representation

5.3 Discussion

5.3.1 Worker Scaling

A key observation is, that the more workers that are involved, the greater the average roundtrip duration, as shown in Figure 10 or in Figure 18. One factor to consider are the available system resources (i.e. logical threads) but also how the main thread can manage communication between contexts. As pointed out in section 5.2, atomic based tests are slower when no or small payloads are involved, which could be attributed to the implementation relying on asynchronous promise wrappers, to make the functionality more easily usable for development (as demonstrated in appendix B). Resolving the asynchronous atomic wait instructions in several steps via promises through the microtask queue likely adds some overhead, depending on what else is enqueued. Another factor could be that the underlying JavaScript engine implementation of atomics is slightly slower compared to the postMessage implementation (Mozilla Corporation, 2024b).

By having more workers that send messages concurrently, main thread bottlenecks can occur (Verdú and Pajuelo, 2016, p. 106-107). Staying within the limitations of the available hardware resources is important, as having more software threads than logical ones introduces additional context switching penalties and cache misses that translate to lost performance (chap. 2.6.2 and 2.8.1). Depending on the application this might not be a problem in practice, for the reasons discussed in section 5.3.3.

5.3.2 Browser Bugs

Apart from the tests that failed and Firefox not supporting asynchronous atomic waiting, another browser bug was found during data collection. The benchmark suite measures the total roundtrip time but also determines how much of the roundtrip time was spent sending and receiving ("to" and "from" Worker), which is stored in separate columns in the CSV-files. This granularity is made possible by the test design and use of `timeOrigin` to synchronize all performance measurements into the same timeline. However, Google Chrome does not follow the W3C specification and sets the `timeOrigin` based on the systems wall clock, instead of providing a monotonic clock value (Google LLC., 2021b). For short workload runtimes this does not necessarily have an effect, but long persisting website sessions, such as the workload tests that run for several hours at a time, are prone to time drifts between origins due to clock adjustments, e.g. via Network Time Protocol (NTP). In practice, this bug is reflected in "to" worker measurements becoming excessively large and "from" worker measurements becoming negative. Adding these values together results again in the roundtrip time, though the time ratio between sending and receiving is incorrect and unusable. The Chrome development team fixed this `timeOrigin` drift issue in 2022 but reverted the change to the previous behavior a month later, after the Microsoft Office Online team complained about discrepancies in their applications, as they use the wall clock synchronized `timeOrigin` as timestamp to compare externally changed documents (Google LLC., 2021a).

An open bug report regarding `timeOrigin` also exists for Safari/WebKit (Apple Inc., 2021). This bug does not influence benchmark measurements as it only surfaces after system sleep. The benchmark suite tries to acquire a screen wake lock when a testplan is started, which can fail on Safari specifically due to security concerns but macOS was prevented from sleeping during the benchmarks using the `caffeinate` command.

Firefox likewise has a `timeOrigin` bug that only drifts between different processes (Mozilla Corporation, 2018b). However, worker threads spawned in the same process as the main thread are not affected by the bug. The measurements taken for the thesis do not show this issue.

As for the failing tests with large payloads (i.e. 10 MiB), Chrome and Safari seem to differ in their implementations from Firefox, as Firefox finishes the tests correctly and without running out of memory. The benchmark suite is not optimized in the sense, that generating and sending a payload needs more memory than just the payload size. Possibilities for running out of memory could be that Chrome and Safari keep references on mutated objects more eagerly, or that the Garbage Collectors run at different intervals. Documentation on Apples supposed memory limit for Safari tabs would be useful for developers, as not being able to count on memory swap poses a severe limitation when working on complex web applications (Jackson, 2022, comment 14).

5.3.3 Recommendations

Since multimedia applications fundamentally differ in their requirements and implementations, and software architecture, different optimization approaches may be used. Even within one application, several transmission methods can be useful for certain tasks. The data from section 5.2 demonstrates, that low latency and high throughput in real-time applications with Web Workers can be achieved by building an architecture around predefined, shared memory regions, especially the larger payload data sizes become. Access to these regions from the worker and main thread is relatively performant because no new memory needs to be allocated after initialization and no duplication of data needs to happen, effectively eliminating high-cost operations and leaving more time for workload processing instructions. Depending on the use-case it might make sense to overprovision the shared memory size slightly to prevent memory allocations shortly thereafter. Workloads that require no large data payloads but fast transmission of signals between contexts should make use of postMessage event-based concurrency implementations. A mixed approach of SharedArrayBuffers for data storage and postMessage as notification mechanism can combine the advantages of both techniques.

Regardless of workload type, applications benefit the most from using code patterns that utilize JavaScript engine optimizations. Adhering to the same object structures (i.e. object key names and order) yields a performance improvement of a factor between six and eight, as exemplified in Figure 15.

The worker shuffling mentioned in section 5.2.5 should likely not be implemented in real applications, as the reason behind it was to statistically distribute measurements more evenly. Work should be distributed according to a strategy that is suitable for a particular application (i.e. round robin, least busy, work-stealing, etc.) (Hunter and English, 2021, p. 129; Kiefer et al., 2015, p. 408-409). Another possibility is to use the `Atomics.notify()` count option to wake up workers on a first in, first out (FIFO) approach (Hunter & English, 2021, p. 100).

Multimedia applications (e.g. audio, video, 3D rendering), software that uses Machine Learning (ML), or other workloads with large amounts of data can also profit from predefined shared memory, if there is a need to access data from different contexts within the application, or if there is a benefit to move processing into a separate thread. However, limited system resources must be considered and compromises made where necessary, i.e. keeping all data in memory with large and many files is often not possible (chap. 5.1).

6 Conclusion

6.1 Summary

Optimizing in abstracted environments such as browsers, that make use of high-level language features (e.g. Garbage Collection), is not an easy task. An automated benchmark application was created using the Design Science Research approach, that can run repeatable test definitions on various devices to find data transmission patterns for low latency and high throughput in web applications using web workers (chap. 4.1). Six gigabyte of test data were generated on three devices, four operating systems, and three web browsers. With all test parameter variations, 5760 tests were planned, of which 99.62 % finished. The failed tests included large payloads when testing Chrome and Safari. Firefox was unaffected from test failures but does not support asynchronous waiting for atomics-based notification mechanisms, which were not the tests that failed in other browsers. The trends in measurement results are otherwise consistent between browsers.

More Web Workers generally cause more communication overhead for the main thread, when the main thread is responsible for worker coordination (chap. 5). The most throughput with large data can be achieved using shared memory regions and atomics, and the best latency with little data using postMessage. Depending on the workload a mixed approach can make sense.

Due to JavaScript engine byte code optimizations, code execution of frequently used functions becomes faster, which is most noticeable up to the first 50 calls (chap. 2.9.3; chap. 5.2). When transferring objects, the use of same object shapes significantly improves transmission timings. In a test with otherwise same object parameters, an extremely large object with random keys took 95 ms to transmit compared to 15 ms with deterministic keys.

Binary Large Object payloads up to 100 KiB are safe to be used in a user interaction context, where response latency matters (chap. 5.2.2). Larger postMessage payload transmission durations grow linearly with data size, whereas atomic shared buffer transmission durations grow slightly with the amount of overhead the system encounters, making shared memory buffers more performant at those sizes of data. Furthermore, at large postMessage payload transmission sizes, e.g. 10 MiB, alternating delay patterns of roughly 3 ms emerge, which are caused by Garbage Collection events.

Payloads consisting of objects with a breadth of six and depth of three, which corresponds to a size of about 9 KiB, become more costly to transmit via postMessage than the overhead that atomics have (chap. 5.2.3). Although serialization costs are not included in these measurements due to the variety of serialization options. For very large data the most throughput can be achieved via shared memory regardless, and binary encoded object data is less costly to transfer than complex objects that implicitly invoke the structured clone algorithm with postMessage.

When transmitting little floating-point data, it is again faster to use postMessage (chap. 5.2.4). With more than 10 floating-point values it becomes more efficient to use arrays for data storage, rather than objects, as object transmission costs grow by a factor of 10. Between 100 and 1,000 floating-point

values postMessage takes increasingly more time, whereas atomics and shared buffers experience no performance overhead. Floating-point values do not require serialization when used in conjunction with arrays, as they are a primitive data type.

Dispatching data to workers in a serial order causes the workers that are at the end of the list to take almost twice as long (chap. 5.2.5). This happens due to main thread coordination bottlenecks and can be prevented through randomizing the worker order for each run. Having no synchronization strategy (i.e. no barrier for parallel workloads) emphasizes the main thread bottleneck. Therefore, a suitable strategy should be chosen depending on the type of application.

When conducting the benchmarks, time origin bugs in browsers became apparent (chap. 5.3.2). Chrome's implementation shows the biggest impact, as it deviates from the specification and applies wall clock adjustments instead of providing a monotonic clock value (Google LLC., 2021b).

6.2 Further Research

Due to the limited two-month time frame given by the International University of Applied Sciences to work on the thesis, not all aspects of the gathered data could be examined in full detail. The variety of test results contain likely more performance patterns, and detailed browser comparisons could not be made as it would have exceeded the scope of this thesis. The CSV-files that came from tests conducted with Safari and Firefox contain usable sending and receiving time ratios which can be used to gain more insights into the behaviors of the workloads examined in the previous chapters. Sending and receiving time ratios taken with Chrome are unusable due time origin wall clock adjustments (chap. 5.3.2). The benchmark web application visualizes this ratio information, but the graphs shown in this thesis do not.

The plots of data aggregated from runs were filtered between the 5th and 95th percentile for cleaner representation in graphs, but it might be relevant in the context of real-time applications to look closer at the outlier values and try to figure out what causes them to occur, as the 99th percentile can represent lag spikes and dropouts (chap. 4.5 and 2.7.1).

Furthermore, research into communication across workers could be of interest for more efficient work division, such as divide and conquer approaches, map & reduce, etc. without having to rely on the main thread for coordination. Looking into factors such as greater worker amount diversification, monitoring memory usage, or including serialization times could show promising results for further optimizations regarding web applications.

In conclusion, there are further possibilities for optimizations in real-time web applications beyond the topics that are covered in the previous chapters. The results discussed in this thesis may provide a foundation for developers aiming to build fast and efficient multi-threaded web applications.

Bibliography

Apple Inc. (2019). Introducing the JetStream 2 Benchmark Suite. Retrieved July 12, 2024, from https://webkit.org/blog/8685/introducing-the-jetstream-2-benchmark-suite/

Apple Inc. (2021). Bug 225610 - performance.now() does not tick during system sleep. Retrieved July 18, 2024, from https://bugs.webkit.org/show_bug.cgi?id=225610

Apple Inc. (2024). WebKit/Source/WebCore/page/Performance.cpp. Retrieved July 11, 2024, from https://github.com/WebKit/WebKit/blob/9a4557b/Source/WebCore/page/Performance.cpp#L60

Archibald, Jake. (2015). Tasks, microtasks, queues and schedules. Retrieved July 11, 2024, from https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/

Benner-Wickner, M., Kneuper, R., & Schlömer, I. (2020). Leitfaden für die Nutzung von Design Science Research in Abschlussarbeiten. http://hdl.handle.net/10419/229136

Builder.io. (2024). Partytown. Retrieved July 8, 2024, from https://partytown.builder.io/

Creswell, J. W. (2009). *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches* (3rd Ed.). Sage Publishing.

Diehl, P., Brandt, S. R., & Hartmut, K. (2024). *Parallel C++: Efficient and Scalable High-Performance Parallel Programming Using HPX*. https://doi.org/10.1007/978-3-031-54369-2

Ecma International. (2024). ECMAScript 2025 Language Specification. Retrieved July 5, 2024, from https://tc39.es/ecma262/

Ghosh, S. (2023). *Building Low Latency Applications with C++*. Packt Publishing Ltd.

Google LLC. (2020). Measure performance with the RAIL model. Retrieved July 10, 2024, from https://web.dev/articles/rail

Google LLC. (2021a). Performance timeOrigin has significant drift from system clock. Retrieved July 12, 2024, from https://issues.chromium.org/issues/40866530

Google LLC. (2021b). performance.timeOrigin is set from the system clock instead of the monotonic clock, which is inconsistent with the W3C High Resolution Time spec. Retrieved July 12, 2024, from https://issues.chromium.org/issues/40181277

Google LLC. (2024a). Interaction to Next Paint (INP). Retrieved July 10, 2024, from https://web.dev/articles/inp

Google LLC. (2024b). The V8 Sandbox. Retrieved July 25, 2024, from https://v8.dev/blog/sandbox

Haigh-Hutchinson, M. (2009). *Real-Time Cameras: A Guide for Game Designers and Developers*. CRC Press.

Haverbeke, M. (2018). *Eloquent JavaScript: A Modern Introduction to Programming* (3rd). No Starch Press.

Hevner, A., & Chatterjee, S. (2010). *Design Research in Information Systems: Theory and practice* (R. Sharda & S. Voß, Eds.; Vol. 22). Springer New York. https://doi.org/10.1007/978-1-4419-5653-8

Hunter, T., & English, B. (2021). *Multithreaded Javascript: Concurrency Beyond the Event Loop*. O'Reilly Media, Inc.

Jackson, D. (2022). Bug 195325 - Canvas context allocation fails because "Total canvas memory use exceeds the maximum limit". Retrieved July 28, 2024, from https://bugs.webkit.org/show_bug.cgi?id=195325#c14

Jangda, A., Powers, B., Berger, E. D., & Guha, A. (2019). Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, *abs/1901.09056*, 107–120. http://arxiv.org/abs/1901.09056

Karltorp, J. D., & Skoglund, E. (2020). *Performance of Multi-threaded Web Applications using Web Workers in Client-side JavaScript*. https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1434463&dswid=-589

Kerrisk, M. (2010). *The Linux Programming Interface: A Linux and UNIX System Programming Handbook* (1st). No Starch Press.

Kiefer, M. A., Molitorisz, K., Bieler, J., & Tichy, W. F. (2015). Parallelizing a Real-Time Audio Application - A Case Study in Multithreaded Software Engineering. *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 405–414. https://doi.org/10.1109/IPDPSW.2015.32

Kutskir, I. (2017). Creating photopea. Retrieved July 17, 2024, from https://blog.photopea.com/creating-photopea.html

Lavieri, E., Verhas, P., & Lee, J. (2018). *Java 9: Building Robust Modular Applications*. Packt Publishing Ltd.

Lawson, N. (2016). High-performance Web Worker messages. Retrieved July 5, 2024, from https://nolanlawson.com/2016/02/29/high-performance-web-worker-messages/

Lindner, D. (2020). *Forschungsdesigns der Wirtschaftsinformatik: Empfehlungen für die Bachelor- und Masterarbeit*. Springer Gabler Wiesbaden. https://doi.org/10.1007/978-3-658-31140-7

Mozilla Corporation. (2018a). Implement the Atomics.waitAsync proposal. Retrieved July 2, 2024, from https://bugzilla.mozilla.org/show_bug.cgi?id=1467846

Mozilla Corporation. (2018b). performance.timeOrigin + performance.now() is waaay off in one content process, not another. Retrieved July 12, 2024, from https://bugzilla.mozilla.org/show_bug.cgi?id=1429926

Mozilla Corporation. (2024a). DOMHighResTimeStamp. Retrieved July 10, 2024, from https://developer.mozilla.org/en-US/docs/Web/API/DOMHighResTimeStamp

Mozilla Corporation. (2024b). mozilla-central/js/src/builtin/AtomicsObject.cpp. Retrieved July 10, 2024, from https://phabricator.services.mozilla.com/source/mozilla-central/browse/default/js/src/builtin/AtomicsObject.cpp

Mozilla Corporation. (2024c). The Event-Loop. Retrieved July 3, 2024, from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Event_loop

Okamoto, S., & Kohana, M. (2011). Load distribution by using Web Workers for a real-time web application. *International Journal of Web Information Systems*, *7*(4), 381–395. https://doi.org/10.1108/17440081111187565

Österle, H., Becker, J., Frank, U., Hess, T., Karagiannis, D., Krcmar, H., Loos, P., Mertens, P., Oberweis, A., & Sinz, E. J. (2010). Memorandum zur gestaltungsorientierten Wirtschaftsinformatik. *Schmalenbachs Zeitschrift für betriebswirtschaftliche Forschung*, *62*(6), 664–672. https://doi.org/10.1007/BF03372838

Pacheco, P. (2011). *An Introduction to Parallel Programming*. Elsevier Inc.

Pizlo, F. (2016). Introducing the B3 JIT Compiler. Retrieved July 12, 2024, from https://www.webkit.org/blog/5852/introducing-the-b3-jit-compiler/

Silberschatz, A., Galvin, P. B., & Gagne, G. (2013). *Operating system concepts* (9th). John Wiley & Sons, Inc.

Smith, C. U. (1993). Software performance engineering. In L. Donatiello & R. Nelson (Eds.), *Performance Evaluation of Computer and Communication Systems* (pp. 509–536). Springer Berlin Heidelberg.

Soundtrap AB. (2024). About soundtrap. Retrieved July 17, 2024, from https://www.soundtrap.com/about

Stack Overflow. (2023). Most Used Programming Languages. Retrieved June 27, 2024, from https://survey.stackoverflow.co/2023/#section-most-popular-technologies-programming-scripting-and-markup-languages

Surma. (2019a). postMessage scatter plot. Retrieved June 15, 2024, from https://gist.github.com/surma/08923b78c42fab88065461f9f507ee96

Surma. (2019b). Is postMessage slow? Retrieved June 15, 2024, from https://surma.dev/things/is-postmessage-slow/

Svartveit, R. B. (2021). *Multithreaded Multiway Constraint Systems with Rust and WebAssembly*. https://hdl.handle.net/11250/2770614

Tanenbaum, A. S., & Bos, H. (2015). *Modern Operating Systems* (4th Ed.). Pearson Education, Inc.

The R Foundation. (n.d.). The R Project for Statistical Computing. Retrieved June 26, 2024, from https://www.r-project.org/

Van Acker, S., & Sabelfeld, A. (2016). JavaScript Sandboxing: Isolating and Restricting Client-Side JavaScript. In A. Aldini, J. Lopez, & F. Martinelli (Eds.), *Foundations of Security Analysis and Design VIII: FOSAD 2014/2015/2016 Tutorial Lectures* (pp. 32–86). Springer International Publishing. https://doi.org/10.1007/978-3-319-43005-8_2

Verdú, J., Costa, J. J., & Pajuelo, A. (2016). Dynamic web worker pool management for highly parallel javascript web applications. *Concurrency and Computation: Practice and Experience*, *28*(13), 3525–3539. https://doi.org/10.1002/cpe.3739

Verdú, J., & Pajuelo, A. (2016). Performance Scalability Analysis of JavaScript Applications with Web Workers. *IEEE Computer Architecture Letters*, *15*(2), 105–108. https://doi.org/10.1109/LCA.2015.2494585

Wagner, L. (2014). asm.js AOT compilation and startup performance. Retrieved July 9, 2024, from https://blog.mozilla.org/luke/2014/01/14/asm-js-aot-compilation-and-startup-performance/

Wagner, L. (2018). Mitigations landing for new class of timing attack. Retrieved July 9, 2024, from https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/

Wihuri, A. (2023). *A case study on parallelism and multithreading in a meteorological JavaScript web application*. https://aaltodoc.aalto.fi/handle/123456789/119411

Williams, A. (2012). *C++ Concurrency in Action*. Manning Publications Co.

Williams, L. G., & Smith, C. U. (1995). Information Requirements for Software Performance Engineering. In H. Beilner & F. Bause (Eds.), *Quantitative Evaluation of Computing and Communication Systems* (pp. 86–101). Springer Berlin Heidelberg.

World Wide Web Consortium. (2019). High Resolution Time Level 2. Retrieved July 7, 2024, from https://www.w3.org/TR/hr-time-2/

World Wide Web Consortium. (2024a). HTML Standard. Retrieved July 1, 2024, from https://html.spec.whatwg.org/

World Wide Web Consortium. (2024b). WebGPU Explainer. Retrieved July 7, 2024, from https://gpuweb.github.io/gpuweb/explainer/#multithreading

Zakas, N. C. (2010). *High Performance JavaScript*. O'Reilly Media, Inc.

Appendices

Appendix A Benchmark Config Parameters

Table 7: Benchmark Config Parameters

Parameter Name	Data Type	Description
test	string	[1]atomicsBlob, postMessageBlobCopy, postMessageBlobTransfer, [2]atomicsBinaryObject, postMessageBinaryObject, postMessageObjectCopy, [3]atomicsFloatArray, postMessageFloatArray, postMessageFloatObject, atomicsLatency, postMessageLatency, atomicsSpeed, postMessageSpeed
repeats	number	0..n (max. 5 recommended)
workerAmount	number	1..n (e.g. 1, 2, 3, etc.)
workerRandomDispatch	boolean	Shuffle worker dispatch order before each run?
runs	number	1..n (e.g. 500)
payloadSize[1]	number	Number of Bytes: 1..n (e.g. 1024)
breadth[2]	number	1..6
depth[2]	number	1..6
digits[2]	number	1..n (Default: 16)
randomKeys[2, 3]	boolean	Randomize keys of tests with Objects?
floatAmount[3]	number	1..n (e.g. 100)
download	boolean	Auto-download CSV-File after test?
visualizeResults	boolean	Show plot of all runs?
csvEndpoint	string	If set, CSV-File will be PUT at URL

Source: Own representation

Appendix B Benchmark Test Harness - Pseudocode

Figure 21: Benchmark Test Harness - Pseudocode

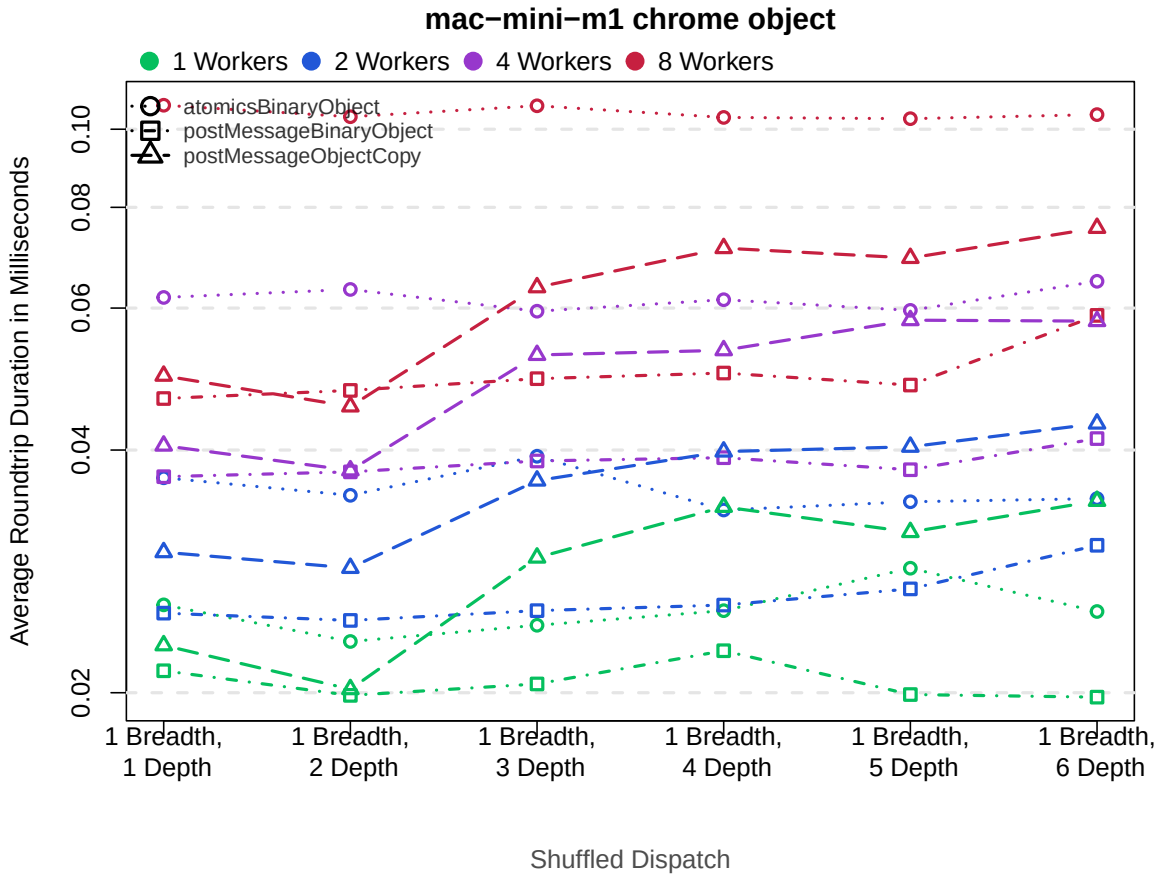
```
// testMain.js
async function doBenchmark(config, workerResultSlots) {
    if (config.randomDispatch) this.shuffleWorkers();
    const barrierPromises = [];

    // generate payloads
    const payloads = [];
    for (let w = 0; w < this.workers.length; w++) {
        payloads.push(generatePayload(config.payloadSize));
    }

    // setup and dispatch runs
    for (let wIdx = 0; wIdx < this.workers.length; wIdx++) {
        const currentWorker = this.workers[wIdx];

        let barrierResolver;
        const barrier = new Promise((res) => {
            barrierResolver = res;
        });
        barrierPromises.push(p);

        currentWorker.addEventListener("message",
            (m) => {
                // access message event data and force implicit deserialization
                const data = m.data;
                const stopTime = performance.now();

                const workerCalculationTime = data.messageSentTime - data.messageReceivedTime;
                const timeOriginDifference = data.workerTimeOrigin - performance.timeOrigin;
                // worker time normalization to main thread
                const workerReceivedNormalized = data.messageReceivedTime +
                    timeOriginDifference;
                const workerSentNormalized = data.messageSentTime + timeOriginDifference;

                const resultSlot = workerResultSlots[wIdx];
                resultSlot.workerIdx = this.workers.indexOf(currentWorker);
                resultSlot.roundtrip = stopTime - startTime - workerCalculationTime;
                resultSlot.toWorker = workerReceivedNormalized - startTime;
                resultSlot.fromWorker = stopTime - workerSentNormalized;
                barrierResolver();
            }, { once: true }
        );
        const payload = payloads[wIdx];
        const startTime = performance.now();
        currentWorker.postMessage(payload);
    }
    return new Promise((resolve) => {
        Promise.all(barrierPromises).then(() => {
            resolve;
        })
    });
}
```

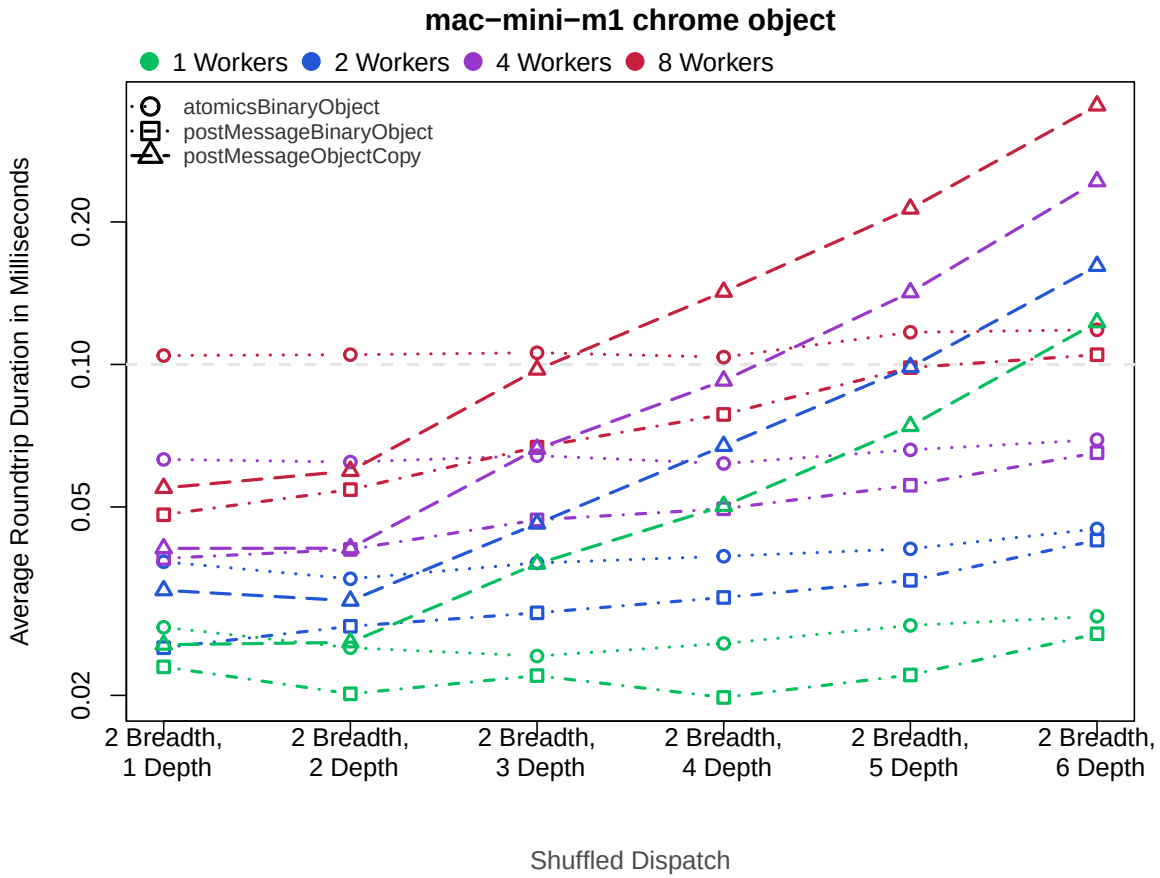
```
// testWorker.js
self.addEventListener("message", (m) => {
    // access message event data and force implicit deserialization
    const data = m.data;
    const messageReceived = performance.now();
    // process
    const payloadSize = data.length;
    const newData = generatePayload(payloadSize);
    //
    newData.messageReceivedTime = messageReceived;
    newData.workerTimeOrigin = performance.timeOrigin;
    newData.messageSentTime = performance.now();
    self.postMessage(newData);
});
```

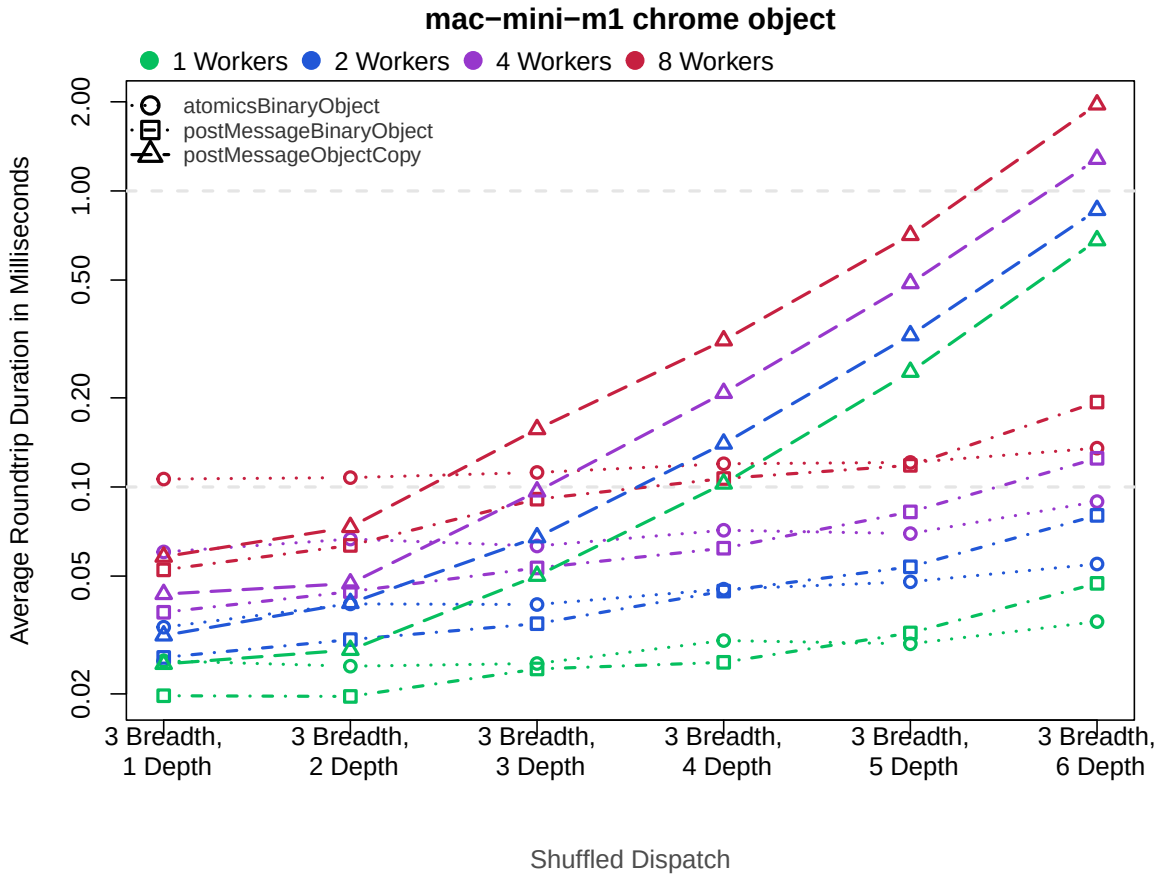
Source: Own representation

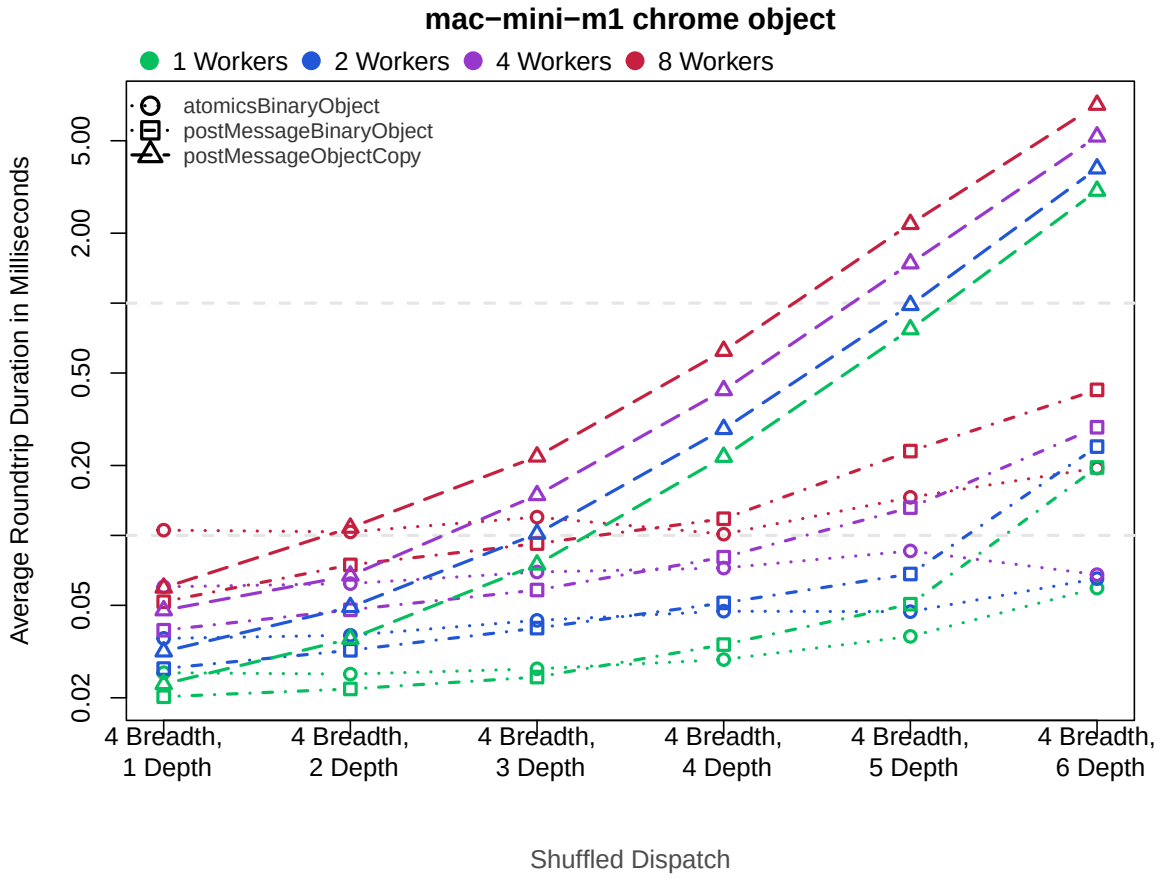
Appendix C Mac Mini M1 Chrome Object Complexity

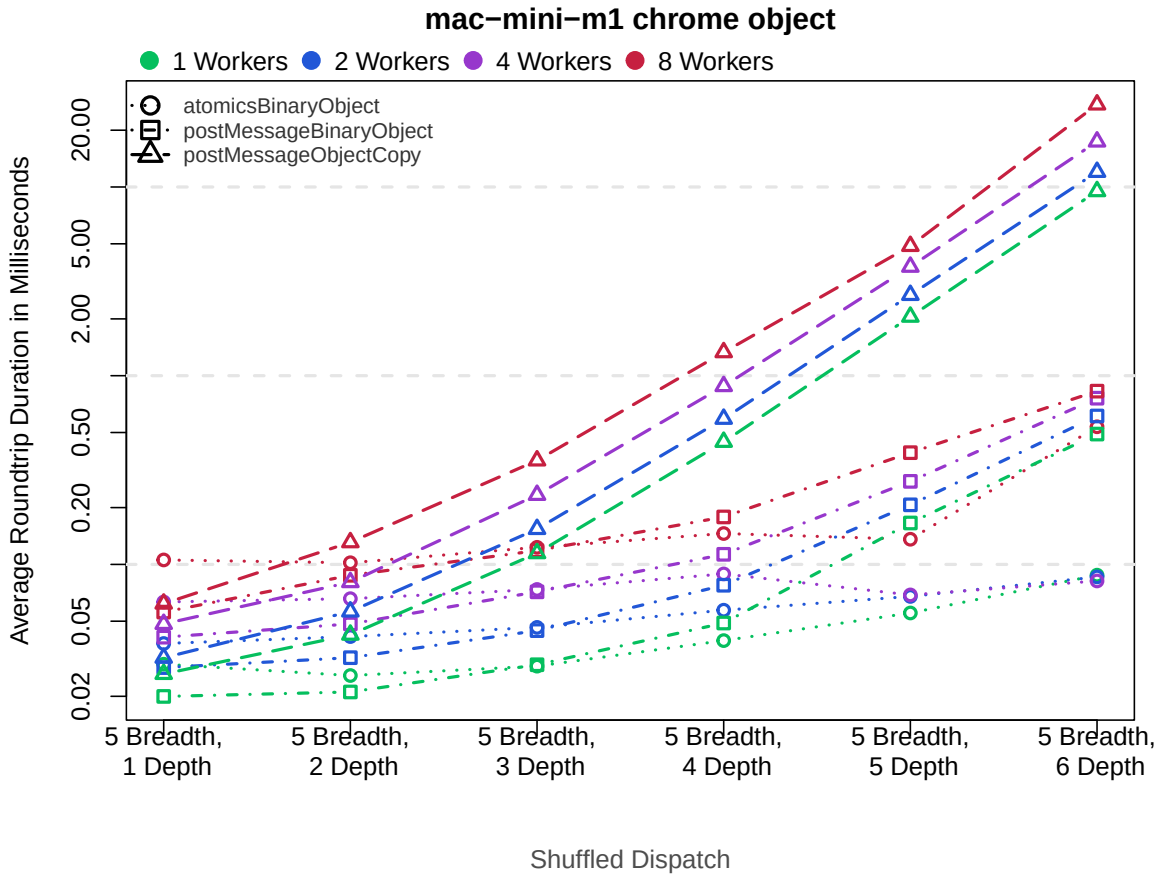
Figure 22: Mac Mini M1 Chrome Object Complexity - 6 Breadth Plots

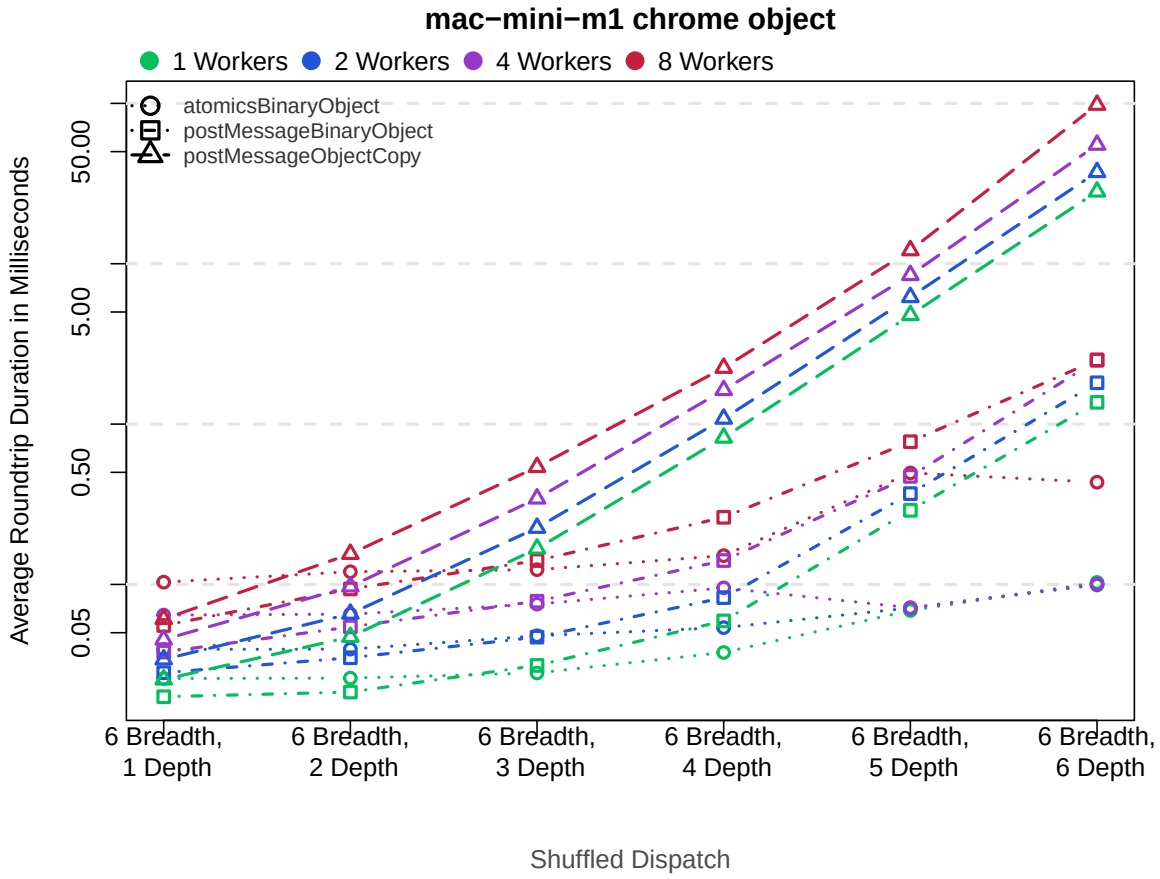





mac−mini−m1 chrome object
1 Workers
2 Workers
4 Workers
8 Workers
atomicsBinaryObject
postMessageBinaryObject
postMessageObjectCopy
Average Roundtrip Duration in Milliseconds
0.02
0.05
0.10
0.20
0.50
1.00
2.00
3 Breadth, 1 Depth
3 Breadth, 2 Depth
3 Breadth, 3 Depth
3 Breadth, 4 Depth
3 Breadth, 5 Depth
3 Breadth, 6 Depth
Shuffled Dispatch


mac−mini−m1 chrome object
1 Workers
2 Workers
4 Workers
8 Workers
atomicsBinaryObject
postMessageBinaryObject
postMessageObjectCopy
Average Roundtrip Duration in Milliseconds
0.02
0.05
0.20
0.50
2.00
5.00
4 Breadth, 1 Depth
4 Breadth, 2 Depth
4 Breadth, 3 Depth
4 Breadth, 4 Depth
4 Breadth, 5 Depth
4 Breadth, 6 Depth
Shuffled Dispatch

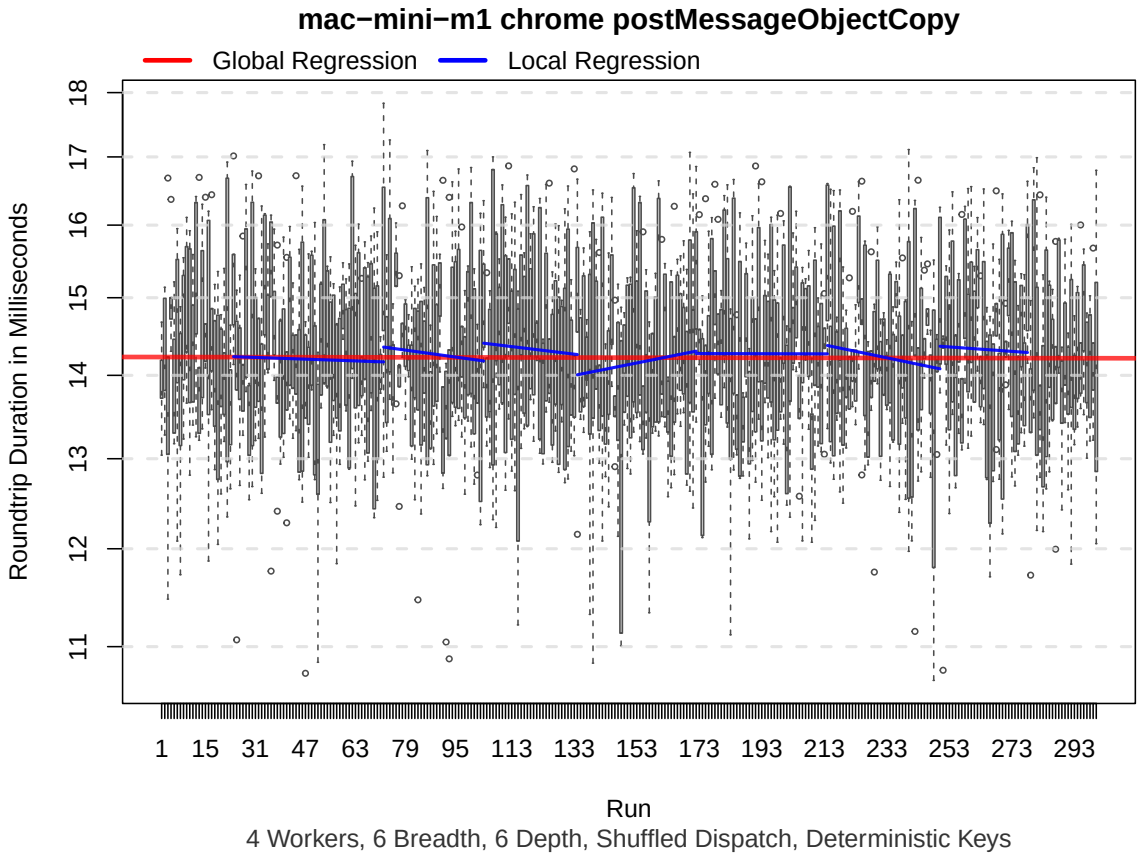


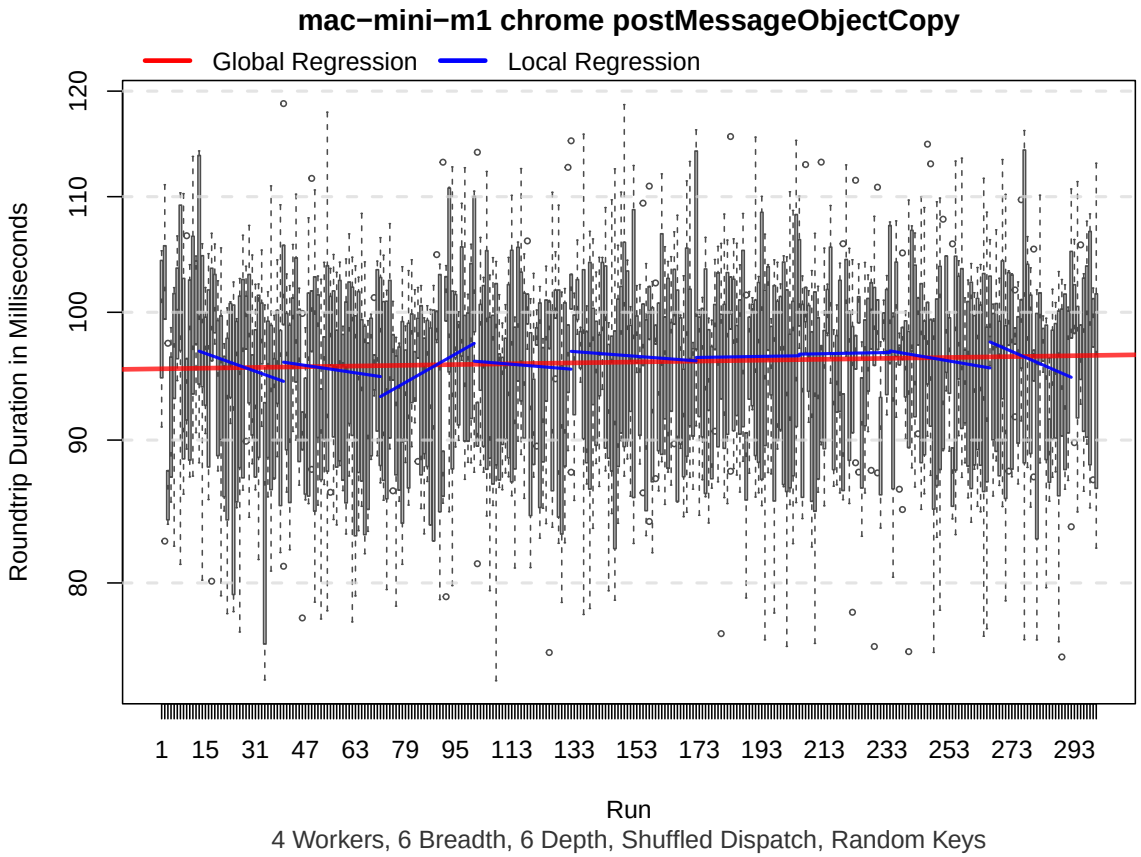


Source: Own representation

Appendix D Mac Mini M1 Chrome Object Keys - Deterministic and Random Runs

Figure 23: Mac Mini M1 Chrome Object Keys - Deterministic and Random





Source: Own representation