Is Mojo Ready for Real-Time? A Language Evaluation for Time-Sensitive System Software

Aaron Bogaert and Antonio Paolillo*

Vrije Universiteit Brussel, Brussels, Belgium
aaron.bogaert@vub.be, antonio.paolillo@vub.be

Abstract. Mojo is a new programming language designed to bridge the gap between Python's usability and the performance of low-level languages such as C and Rust. While Mojo has shown promise in AI and high-performance computing domains, its suitability for time-sensitive system software is unexplored. Real-time systems demand not only raw performance but also strict control over latency, predictability, and hardware resource usage—requirements that few languages meet reliably.
This paper presents the first systematic evaluation of Mojo for real-time and embedded system applications. We introduce a general methodology for language-level real-time evaluation, encompassing metrics such as latency, variability, binary size, GPU control, and predictability under scheduler and interference conditions. We then apply this methodology through a suite of benchmark workloads implemented in Mojo, C, C++, and Rust, ranging from small programs like Mandelbrot and string search to GPU-accelerated matrix multiplication. Our case studies include a reimplementation of cyclictest in Mojo, an assessment of Mojo's GPU execution model, and an evaluation of its applicability to robotic and embedded system design. Results show that while Mojo offers competitive performance and promising capabilities, several limitations—especially in system-level control—must be addressed for adoption in real-time domains.

Keywords: Real-Time Systems · Mojo Programming Language · Deterministic Execution · GPU Acceleration · Benchmarking · Cyclictest.

1 Introduction

Programming languages for real-time and embedded systems must offer more than raw performance. They must enable developers to write software that behaves predictably under stringent timing constraints, with precise control over latency, resource usage, and execution variance. Traditionally, system languages like C and, more recently, Rust, have been the tools of choice in such domains, offering deterministic behavior, low-level control, and minimal runtime overhead.

Mojo is a new programming language introduced by Modular Inc. in 2023, aiming to combine Python's accessibility with the low-level control and performance characteristics of systems languages [24]. It leverages static typing,

* Corresponding author.

MLIR[1], and LLVM[2] to support both ahead-of-time (AOT) and just-in-time (JIT) compilation, offers explicit memory management (without garbage collection), SIMD/vector support, and constructs for parallelism and GPU acceleration. By combining high-level syntax and low-level capabilities, Mojo positions itself as a general-purpose language that is both expressive and performant, with early reports showing performance speedups of 10× or more over Python for data-intensive tasks [34].

These features have already attracted interest in domains like AI and high-performance computing. However, Mojo's suitability for time-sensitive or safety-critical systems is, so far, unexplored. Unlike C or Rust, Mojo is designed from the ground up to act as a bridge between high-level expressiveness and low-level determinism—potentially lowering the barrier for developers coming from Python-heavy domains to access real-time capabilities without sacrificing predictability. This bridging role is particularly relevant in safety-critical and certifiable systems. By encapsulating low-level complexity into reusable, possibly pre-certified MLIR modules, Mojo could reduce the burden on both developers and certification bodies. Instead of baking all target-specific optimizations into the compiler, Mojo's progressive lowering model allows importing architecture-adapted modules directly, focusing verification efforts on high-level logic while trusting well-tested, low-level components. Such a model could make real-time development both more accessible and more auditable—a combination that existing languages do not explicitly optimize for.

However, Mojo is still young, with a relatively small ecosystem and tooling maturity that varies across domains. Its LLVM backend may not yet be optimized for all real-time analyses, and its integration with embedded or RTOS platforms is nascent [12,38,35]. These open questions motivate the present work: before considering Mojo for time-sensitive domains, we must assess not only its peak performance but also its predictability, tooling readiness, and integration potential.

In this paper, we present the first systematic evaluation of Mojo for real-time applications. We introduce a reproducible methodology for language-level real-time evaluation, designed to avoid common pitfalls in cross-language benchmarking: implementations are kept equivalent across languages (Mojo, C, C++, Rust), benchmarks are drawn from standardized workloads, and code is reviewed to minimize developer-skill bias. We evaluate metrics such as execution time, variability, binary size, and GPU execution control under real-time Linux scheduling policies. Our contributions are as follows:

- We perform a qualitative feasibility study of Mojo's integration into robotics pipelines and embedded bare-metal environments.
- We conduct a cross-language evaluation of three representative program types—compute-intensive, compute/memory-mixed, and branch-heavy. We implemented each equivalently in Mojo, C, C++, and Rust, providing indicators of the languages' performance characteristics in real-time contexts.

[1] Multi-Level Intermediate Representation [25].
[2] Low Level Virtual Machine [26].

- We reimplement `cyclictest` in Mojo and compare its latency and timer fidelity against the canonical C version, providing a direct measure of runtime overhead and predictability.
- We present the first assessment of Mojo's readiness for both CPU-bound and GPU-accelerated programming in real-time environments, including an analysis of execution-time variability on GPU workloads.

Through this work, we aim to (1) provide insight into the readiness of Mojo for real-time systems and (2) contribute a reusable, transparent methodology for language-level real-time evaluation. We aim to inform system designers, language developers, and the broader real-time community about the promises and pitfalls of adopting Mojo in timing-critical domains.

Our evaluation shows that while Mojo can match the performance of C and Rust in certain compute-bound workloads and offers promising high-level abstractions for heterogeneous computing, these strengths are offset by notable shortcomings. In particular, our measurements reveal runtime overheads in latency-sensitive tasks, and our qualitative analysis highlights gaps in resource control, thread scheduling, embedded integration, and tooling maturity. These limitations currently prevent Mojo from being a practical choice for hard real-time deployments, but also point to concrete avenues for improvement.

The remainder of this paper is organized as follows. Section 2 reviews the design of Mojo and its features relevant to real-time and embedded contexts. Section 3 details our evaluation methodology, combining qualitative feasibility studies and quantitative performance measurements. Section 4 presents our findings across CPU, GPU, and latency benchmarks, as well as integration case studies. Section 5 discusses related work in real-time languages, benchmarking methodologies, and GPU predictability. Section 6 concludes with a summary of our contributions, identified limitations, and directions for future research.

2 Background: The Mojo Programming Language

Mojo is a systems-oriented programming language that aims to combine the accessible syntax of Python with the low-level performance and control of compiled languages like C++ and Rust. Built on top of MLIR and LLVM, Mojo supports both ahead-of-time (AOT) and just-in-time (JIT) compilation, explicit memory management without garbage collection, deterministic static typing, and high-level abstractions for parallelism and GPU programming.

This section provides a brief overview of the language and toolchain features most relevant to time-sensitive and real-time workloads.

Python-Like Syntax with Systems-Level Semantics. Mojo adopts a Python-inspired syntax for ease of use, while enforcing static types, explicit variable declarations, and deterministic control flow. It supports variable declaration through `var` bindings and performs all type checking at compile time. Unlike Python, Mojo does not rely on dynamic dispatch or garbage collection, avoiding unpredictability introduced by runtime memory management. This design enables the compiler to eliminate runtime type checks and permits aggres-

sive optimization. A side-by-side comparison of Python and Mojo syntax for the same function is shown in Figure 1.

```python
# Python
MAX_ITERS = 1000

def mandelbrot_split(
    real: float,
    imag: float,
) -> int:
    nv = 0
    z_real = real
    z_imag = imag
    for _ in range(MAX_ITERS):
        r2 = z_real * z_real
        i2 = z_imag * z_imag
        if r2 + i2 > 4.0:
            break
        z_imag = 2.0 * z_real * z_imag + imag
        z_real = r2 - i2 + real
    return nv

if __name__ == "__main__":
    print(mandelbrot_split(-0.5, 0.0))
```

```python
# Mojo
alias MAX_ITERS: Int = 1000

fn mandelbrot_split(
    real: Float64,
    imag: Float64,
) -> Int:
    var nv: Int = 0
    var zReal: Float64 = real
    var zImag: Float64 = imag
    for _ in range(0, MAX_ITERS):
        var r2: Float64 = zReal * zReal
        var i2: Float64 = zImag * zImag
        if r2 + i2 > 4.0:
            break
        zImag = 2.0 * zReal * zImag + imag
        zReal = r2 - i2 + real
    return nv

fn main():
    print(mandelbrot_split(-0.5, 0.0))
```

Fig. 1. Comparison of Python and Mojo syntax for a Mandelbrot iteration function. Python executes the function via interpretation, while Mojo compiles it ahead of time.

MLIR and Compilation Model. Mojo code is compiled through a multi-stage lowering pipeline based on MLIR [25], eventually generating optimized native code via LLVM. This design allows for fine-grained optimization and hardware-specific code generation, including SIMD vectorization and GPU offloading. Mojo can be executed in JIT mode for interactive workflows, or built into static AOT binaries using the `mojo build` command—a feature critical for eliminating startup latency in real-time applications.

Memory and Resource Management. Mojo avoids garbage collection in favor of ownership and value semantics, similar to Rust and C++. This enables predictable memory management and deterministic deallocation patterns, which are desirable in systems with tight timing constraints. However, Mojo does not currently expose APIs for fine-grained control over threads, memory sharing across threads, CPU affinities, or system scheduling parameters, which may limit its applicability in low-level, latency-sensitive environments.

Concurrency and Parallelism. The language provides high-level constructs for parallel computation through the `parallelize` keyword and SIMD types. These constructs enable developers to write data-parallel code without explicit thread management. However, current implementations offer limited transparency over the underlying thread spawning behavior, making it difficult to control the degree of parallelism or integrate with OS-level schedulers—a key limitation for real-time systems.

Statically Typed Structs and Traits. In contrast to Python's dynamic classes, Mojo provides `struct`s as its primary user-defined data abstraction. Structs support method definitions, trait-based polymorphism, and field-level

initialization via decorators such as `@fieldwise_init`. Structs are statically laid out in memory, and dispatch is entirely static, eliminating the unpredictability of dynamic method lookup. Traits in Mojo resemble interfaces or concepts in other languages. They define a set of methods or requirements a struct must implement to conform. This enables both abstraction and generic programming without incurring runtime dispatch costs. Importantly, traits also support compile-time evaluation through parameterization.

GPU Acceleration. Mojo includes first-class support for GPU execution. Mojo's GPU programming model is tightly integrated with the language, and includes constructs for managing memory buffers, kernel launches, and device-specific execution—positioning it as a potential candidate for GPU-heavy real-time workloads. However, its runtime-level control and predictability on GPUs are unexplored.

Compile-Time Metaprogramming. Mojo introduces a parameter system for compile-time constants and evaluation. Functions and structs can be parameterized using square brackets (e.g., `repeat[3]("hello")`), allowing the compiler to unroll loops or generate specialized code variants at compile-time. This mechanism supports high-performance patterns while maintaining type and memory safety.

Interoperability and Embedded Suitability. Mojo supports tight integration with Python libraries via interoperability bindings. While useful for prototyping and scientific computing, this feature introduces dynamic runtime dependencies and garbage-collected memory models, and should generally be avoided in real-time or safety-critical contexts. At the time of writing, Mojo targets x86_64 Linux platforms; official support for Arm, embedded boards, or real-time operating systems is not yet available. Its reliance on the Modular SDK also limits its deployability in constrained or sandboxed environments.

Execution Environment. Mojo does not provide official support for embedded deployment or RTOS integration. However, Mojo's use of LLVM and MLIR makes it feasible to explore custom backends integration with real-time kernels, such as PREEMPT_RT Linux or Zephyr, as a future direction.

3 Evaluation Methodology

The objective of our study is to assess the readiness of the Mojo programming language for real-time and embedded system applications. We structure the evaluation along two complementary axes: (i) a *qualitative feasibility analysis*, aimed at determining whether Mojo can integrate into representative real-time software contexts, and (ii) a *quantitative performance assessment*, measuring its execution behaviour under timing constraints and comparing it to established systems programming languages such as C, C++ and Rust. All experiments were conducted on a Linux platform, chosen both for its widespread adoption in real-time and embedded research and because Mojo currently targets Linux as one of its primary runtime environment [30]. All experimental artifacts used in this evaluation are publicly available on GitHub [3].

3.1 Qualitative Feasibility Analysis

The qualitative part of the evaluation investigates the extent to which Mojo can be deployed in software stacks that are common in timing-sensitive domains. Rather than focusing on raw performance, this stage emphasizes functional integration, tooling maturity, and exposure of low-level control mechanisms that are typically required in such systems. We consider three representative scenarios.

First, we examine the ability to integrate Mojo into an existing *robotics control pipeline*. In particular, we use Mojo's `Python.import_module` facility to invoke control routines from the `ur_rtde` library, which provides real-time interfaces for Universal Robots (UR) industrial robotic arms [28]. Although Python is not inherently suited for real-time control, it is nonetheless pervasive in robotics ecosystems—most notably through ROS and its associated tooling—where many time-sensitive components are implemented in Python or rely on Python bindings. Evaluating Mojo in such a context is therefore meaningful: its interoperability with existing Python modules enables developers to incrementally migrate performance-critical sections to a compiled environment without rewriting entire control or perception pipelines. This facilitates piecewise modernization of established robotics software stacks while maintaining compatibility with mature Python libraries. We evaluate the functional viability of such hybrid Mojo–Python workflows and identify potential sources of timing variability, including the Python Global Interpreter Lock (GIL), dynamic dispatch, and garbage collection.

Second, we explore Mojo's suitability for *embedded and bare-metal deployment*. Here, the goal is to determine whether Mojo code can be compiled to object files targeting non-host architectures and linked into an existing real-time or embedded operating system kernel. We target an Armv8 bare-metal environment, emitting objects via `--target-triple` and `--mcpu` flags, and linking them alongside C and assembly components of a toy operating system kernel. We assess the stability of the cross-compilation process, the compatibility of the generated code with the C Application Binary Interface (ABI), and the availability of language features needed for low-level operations such as volatile memory-mapped I/O.

Finally, we evaluate Mojo's ability to *control system-level scheduling parameters*, including thread priorities, CPU affinities, and device latency settings such as `/dev/cpu_dma_latency`. This is critical for applications that require precise control over execution placement and latency isolation. We attempt to configure these parameters from within Mojo and document any limitations or reliance on external tools.

3.2 Quantitative Performance Assessment

The quantitative evaluation characterises Mojo's timing behaviour and compares it to C, C++, and Rust using a common benchmark suite. The suite comprises semantically equivalent implementations across all languages, covering representative computational patterns found in real-time and systems work-

loads: compute-dominated loops, mixed compute/memory kernels without explicit SIMD, and branch-heavy string processing. All implementations are self-contained, avoid dynamic allocation in the hot path to reduce variability, and omit, when possible, external library calls. Benchmarks were executed in controlled, interference-free conditions with CPU isolation. Specifically, the system was booted with the `isolcpus` kernel parameter, and single-thread benchmark processes were pinned to one of the isolated cores using `taskset`. This configuration ensured that each single-threaded benchmark executed on a dedicated core, free from interference by other user or kernel threads. For each workload, we measured per-run execution time over multiple repetitions and derived standard descriptive statistics. We also collected the binary sizes of the object files for these benchmarks.

In addition to the microbenchmarks, we reimplemented `cyclictest` in Mojo to measure wake-up latency under high-priority real-time scheduling, enabling a direct comparison to the canonical C implementation. The Mojo version preserves `cyclictest`'s use of absolute sleeps and includes page locking to avoid major page faults. Results are reported for both generic and `PREEMPT_RT`-patched kernels.

This experimental design allows us to evaluate Mojo's current performance envelope across a range of workloads, identify systematic differences from established systems languages, and highlight the cases where runtime or compiler improvements could have the greatest impact.

4 Results

In this section, we present our findings according to the two evaluation axes outlined in Section 3. Section 4.1 reports the *qualitative feasibility* outcomes, while Section 4.2 summarises the *quantitative performance* results. Where appropriate, we highlight limitations and implementation gaps observed during the experiments.

4.1 Qualitative Findings

To assess Mojo's potential in time-sensitive and system-level contexts, we conducted a series of exploratory case studies. Rather than focusing on performance metrics, these case studies aim to evaluate the language's expressiveness, interoperability, and suitability for deployment in realistic scenarios drawn from robotics and embedded systems. Each case probes whether Mojo's abstractions—and its surrounding toolchain—are capable of supporting software components that typically require fine-grained control, deterministic behavior, and close integration with system software. Our qualitative evaluation aimed to determine whether Mojo can be integrated into these representative real-time software contexts.

Robotics and Python Interoperability. Mojo offers full interoperability with Python, allowing developers to import Python modules and invoke their

functions directly from Mojo code. We validated this using a realistic robotic control task based on the `ur_rtde` library [28] (for controlling a UR-series robotic arm). We wrote a simple Python module `robotmod.py` that establishes a connection with the robot, sends a Cartesian move command, and prints the measured joint positions:

```
# robotmod.py
import rtde_control, rtde_receive

def robot_move():
    rtde_c = rtde_control.RTDEControlInterface("172.17.0.2")
    rtde_c.moveL([-0.143, -0.435, 0.20, -0.001, 3.12, 0.04], 0.5, 0.3)
    rtde_r = rtde_receive.RTDEReceiveInterface("172.17.0.2")
    print("Actual joint positions:", rtde_r.getActualQ())
```

We then invoked this module from Mojo using the Python interoperability interface:

```
# robot.mojo
from python import Python

fn main():
    try:
        robotmod = Python.import_module("robotmod")
        robotmod.robot_move()
    except:
        print("Exception occurred!")
```

Running `robot.mojo` correctly triggered the Python-based motion routine, confirming that Mojo can serve as a lightweight controller or coordinator in Python-centric robotics environments.

However, because the call stack eventually enters the Python runtime, the process inherits the Global Interpreter Lock (GIL), garbage collection, and dynamic dispatch, which reintroduce timing variability. Therefore, while this workflow is highly valuable for rapid prototyping and for integrating with existing robotics pipelines and libraries, it is inherently unsuitable for hard real-time systems. The reliance on the Python runtime introduces sources of variability that cannot be eliminated without fundamentally altering Python itself.

Embedded Systems and Bare-Metal Deployment. We investigated the feasibility of integrating Mojo into an embedded, bare-metal environment by replacing selected components of a toy operating system kernel—originally implemented in C and assembly—with equivalent Mojo code. Our target platform was a bare-metal `AArch64` system (i.e., `Armv8`). The experiment was designed to assess Mojo's cross-compilation capabilities, its compatibility with the C Application Binary Interface (ABI), and its suitability for low-level hardware interaction.

Cross-compilation and linking. Mojo supports ahead-of-time (AOT) compilation to object files, and can emit code for non-host targets by specifying appropriate compilation flags. In our setup, we successfully generated `AArch64` object files and linked them into the kernel image alongside C and assembly sources using a custom linker script and `-nostdlib` to avoid dependencies on the host C library:

```
# Compile Mojo to an AArch64 object file
mojo build sum.mojo --emit object \
  --target-triple aarch64-none-elf --mcpu cortex-a72 -o sum.o

# Link with kernel C/ASM components
aarch64-linux-gnu-gcc -nostdlib -static \
  -T linker.ld ... sum.o -o toyos.elf
```

Exporting symbols for use by C code was functional via the `@export("name", ABI="C")` annotation:

```
# sum.mojo
@export("sum", ABI="C")
fn sum(a: Int32, b: Int32) -> Int32:
    return a + b
```

Foreign Function Interface (FFI) and Memory-Mapped I/O (MMIO). Bare-metal software often interacts with peripherals trough memory-mapped I/O (MMIO), in which device registers are exposed at fixed memory addresses and accessed using volatile loads and stores. In Mojo, the most robust interoperability mechanism we identified is through the `external_call` construct from the `sys.ffi` module. However, the Mojo toolchain version available at the time of our experiments did not support direct raw-address pointers (e.g., a constructor such as `UnsafePointer.from_address(...)`), nor did it provide a stable volatile API. Attributes such as `@extern` or `@inline` were also not yet implemented. To work around these limitations, we implemented privileged MMIO operations in small C functions compiled into the kernel and invoked them from Mojo. The following C shim reads from and writes to a 32-bit MMIO register:

```
/* mmio_shim.c */
#include <stdint.h>
uint32_t mmio_read32(uint64_t addr) {
    return *(volatile uint32_t *)(uintptr_t)addr;
}
void mmio_write32(uint64_t addr, uint32_t v) {
    *(volatile uint32_t *)(uintptr_t)addr = v;
}
```

UART driver implementation. To demonstrate end-to-end integration, we implemented a basic Universal Asynchronous Receiver–Transmitter (UART) driver in Mojo. The driver uses the C shim for volatile register access, with memory-mapped addresses defined as constants in Mojo:

```
# uart.mojo
from sys.ffi import external_call

alias UART_BASE:    UInt64 = 0x0900_0000
alias UART_DR_ADDR: UInt64 = UART_BASE + 0x00
alias UART_FR_ADDR: UInt64 = UART_BASE + 0x18
alias UART_FR_TXFF: UInt32 = 1 << 5

fn mmio_read32(addr: UInt64) -> UInt32:
    return external_call["mmio_read32", UInt32, UInt64](addr)

fn mmio_write32(addr: UInt64, value: UInt32) -> None:
    external_call["mmio_write32", NoneType, UInt64, UInt32](addr, value)

@export("uart_send", ABI="C")
fn uart_send(c: UInt8) -> None:
    while (mmio_read32(UART_FR_ADDR) & UART_FR_TXFF) != 0:
        pass
    mmio_write32(UART_DR_ADDR, UInt32(c))
```

This driver transmits a single byte by polling the UART status register until the transmit FIFO is not full, then writing the byte to the data register.

Build system integration. Mojo was integrated into the build process using CMake, with custom rules to compile each `.mojo` source file to an object file (passing `-target-triple` and `-mcpu`) before linking it alongside C and assembly code. Linker scripts, startup routines, and other board-specific initialisation remain managed by the C/ASM toolchain (`GCC`/`LD`), as Mojo currently provides no first-class support for these embedded concerns.

Summary of observations. In its current state, Mojo can be used to implement compute-intensive, self-contained functions for embedded systems, compiled as relocatable objects and linked into existing bare-metal codebases. However, the absence of stable APIs for raw pointer manipulation and volatile memory access, combined with the lack of direct support for linker scripts, interrupt handling, and inline assembly, limits its ability to fully replace C in low-level device drivers. Bridging these gaps would require stabilising pointer and volatile semantics, expanding cross-target documentation, and providing explicit system-level features.

System-Level Control. On a standard Linux runtime, we found that setting thread priorities, adjusting CPU affinity, or configuring `/dev/cpu_dma_latency` directly from Mojo was not possible due to the absence of bindings for low-level system calls. The only available approach is to invoke these calls indirectly through the C library using `sys.external_call`. While functional, this indirection complicates debugging, as standard error-reporting mechanisms such as `errno` are not directly accessible, making failed system calls difficult to detect

and handle reliably. As a workaround, such parameters can be configured externally (e.g., via `chrt` or `taskset`), but this breaks self-containment.

We also observed unexpected behavior for certain device interactions. For instance, opening `/dev/cpu_dma_latency` in write mode using the standard `open()` API results in the following error: "`unable to remove existing file /dev/cpu_dma_latency`". This behavior stems from the use of Mojo-specific symbols such as `KGEN_CompilerRT_IO_FileOpen` that are present in the runtime's file-opening logic, which is part of a closed-source component of the Mojo runtime. Without access to its implementation, further diagnosis is not possible. These limitations restrict Mojo's ability to adapt scheduling parameters at runtime, which is often essential in real-time workloads requiring precise execution placement and latency isolation.

4.2 Quantitative Findings

Our quantitative evaluation addresses three research questions derived from the objectives in Section 3:

RQ1: *Can Mojo deliver predictable latency under real-time scheduling policies?* This is assessed using CPU microbenchmarks and `cyclictest` under `SCHED_FIFO` with and without interference.

RQ2: *How does Mojo's performance compare to established systems languages such as C and Rust?* We compare execution time, variability, and binary size across equivalent kernel implementations.

RQ3: *How does Mojo's GPU execution model behave in terms of timing stability?* We measure execution time variability for GPU-accelerated workloads, focusing on matrix multiplication, and compare it to CPU execution.

The experiments follow the methodology outlined in Section 3, presenting results in the same order: CPU microbenchmarks (RQ1 and RQ2), `cyclictest` (RQ1), and GPU execution variability (RQ3).

Microbenchmarks (RQ1, RQ2). To address **RQ1** (predictable performance) and **RQ2** (comparison to other compiled languages), we evaluated three CPU-bound microbenchmarks implemented equivalently in Mojo, C, C++, and Rust: (i) *Mandelbrot* (compute-intensive, embarrassingly parallel), (ii) *Matrix Multiplication* with statically sized operands and no explicit SIMD intrinsics (compute/memory mix), and (iii) *StringSearch* using the Knuth–Morris–Pratt (KMP) algorithm (branch-heavy, pointer/byte-wise processing). None of the benchmarks perform dynamic allocation in the hot path, and external dependencies are avoided whenever possible.

Experimental setup. Benchmarks were run in isolation on a "big" P-core of an Intel Core i9-13900H (Raptor Lake) under the Ubuntu 24.04 distribution (Linux `6.14.0-27-generic`), with no other user processes running on the benchmark core (using the `isolcpus` kernel boot argument). We compiled C and C++

variants with both Clang and GCC, and Rust with `rustc` and `cargo`, each at `-O1`, `-O2`, and `-O3` optimization levels. Mojo was compiled in AOT mode with the corresponding optimization flags. We report mean runtime over 20 iterations, along with standard deviations as error bars, in Figure 2.

Mandelbrot (compute-dominated). For both problem sizes (256 and 512), Mojo achieves the shortest runtimes across all optimization levels, outperforming C, C++, and Rust. At `-O3`, its performance advantage remains consistent, with narrow error bars indicating stable and repeatable execution. This strong result likely reflects the efficiency of Mojo's standard library and its LLVM-based math intrinsics, which generate highly optimized code for arithmetic-intensive workloads.

Matrix Multiplication (SIMD-limited variant). For the smaller case (256×256), Mojo performs close to the best C (Clang) baseline, ranking second overall and outperforming C (GCC), C++, and Rust across all optimization levels. In the larger case (512×512), however, Mojo's runtime increases substantially, exceeding all compiled baselines. This disparity suggests that Mojo's current code generation lacks some of the optimizations leveraged by mature C/C++ compilers. Runtime variability is low across all languages, confirming stable and repeatable performance measurements.

StringSearch (branch-heavy). Mojo exhibits a pronounced performance deficit, running roughly 4–8× slower than C, C++, and Rust across all optimization levels and input sizes. The KMP benchmark's irregular control flow, pointer chasing, and branch-dependent logic likely expose current weaknesses in Mojo's code generation for branch-heavy kernels. Optimization flags (`-O1`–`-O3`) have little effect. Although runtime variability remains low, the consistently higher runtimes indicate that this class of workloads is not yet well optimized in Mojo.

Takeaway. Mojo's performance is competitive for arithmetic-heavy kernels (i.e., Mandelbrot) and reasonable for moderate-size, mixed compute/memory workloads (i.e., small MatMul), but degrades for larger data sets and branch-intensive algorithms (i.e., KMP). Predictability across runs is generally good, with Mojo performing on par with, or only slightly worse than, the other languages. Absolute performance remains sensitive to code patterns that rely on mature compiler optimizations and SIMD exploitation.

Binary file sizes. Figure 3 reports the file sizes of the microbenchmark executables. Executable size is an important factor for embedded systems, and in this respect, Mojo consistently produces larger binaries than the other languages. Even in its best case—Rust at `-O0` for the KMP benchmark—Mojo's output was still 90% larger. Mojo embeds substantial portions of its standard library into every binary, resulting in significant code size even for small programs. In our benchmarks, this includes large SIMD-related jump tables likely originating from Mojo's `DType` implementation.

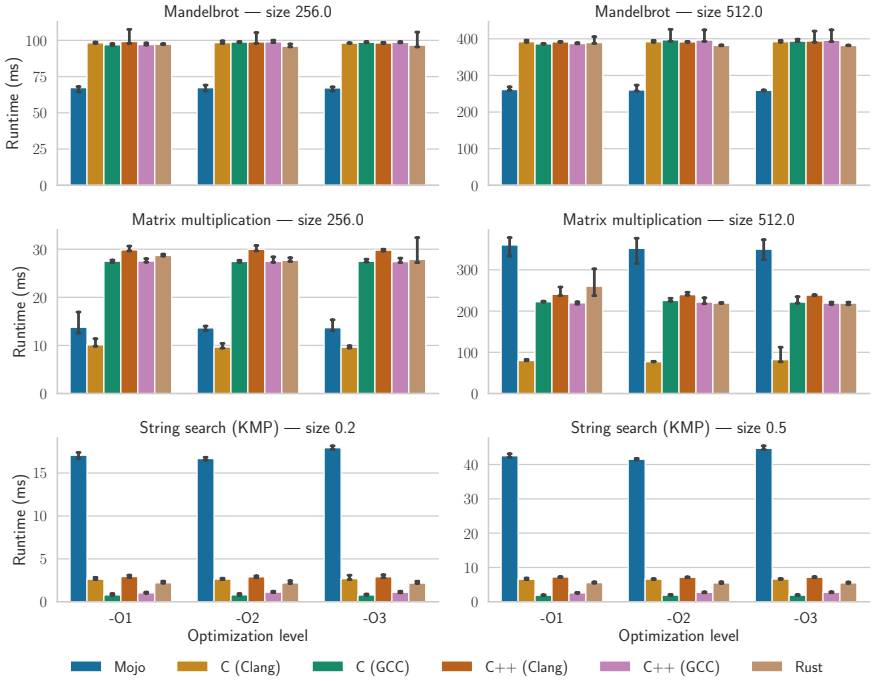


Fig. 2. Mean runtimes for microbenchmarks at different optimization levels. Error bars indicate standard deviation across runs. Mojo is competitive for Mandelbrot, reasonable for small MatMul, and slower for branch-heavy StringSearch.

Cyclictest Reimplementation (RQ1). To address **RQ1** (predictable latency under real-time scheduling), we reimplemented the well-known `cyclictest` [39] tool in Mojo and compared it against the canonical C version. This benchmark measures wake-up latency by repeatedly sleeping until an absolute wake-up time using `clock_nanosleep`, then recording the deviation from the expected wake-up. Since `cyclictest` latency is influenced mainly by hardware, firmware, and kernel scheduling, any differences between the Mojo and C results can be attributed to language runtime overheads, system call invocation paths, or implementation differences.

Implementation details. Due to the size of the original `cyclictest` code base, we implemented a reduced version covering a representative subset of its command-line options. We support the equivalent of: `cyclictest -nsecs -vnm -i 100 -p 99 -t X -duration=1m -mlockall`. We chose these arguments to follow best practices of prior work [21,13,29,1,14]. As direct system call bindings are not yet available in Mojo (see Section 4.1), we used the `sys.external_call` facility to invoke C functions for `clock_gettime` and `clock_nanosleep`. This external call might introduce some latency compared to the C version. It is, however, the only available method in Mojo at present to achieve the same functionality.

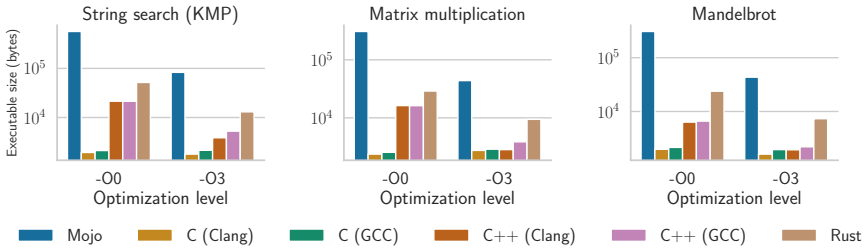


Fig. 3. Executable file sizes (in bytes) measured with `du -b`. Only `-O0` and `-O3` optimization levels are shown, as they represent the extremes. The y-axis uses a logarithmic scale to accommodate the much larger Mojo binaries compared to the other languages.

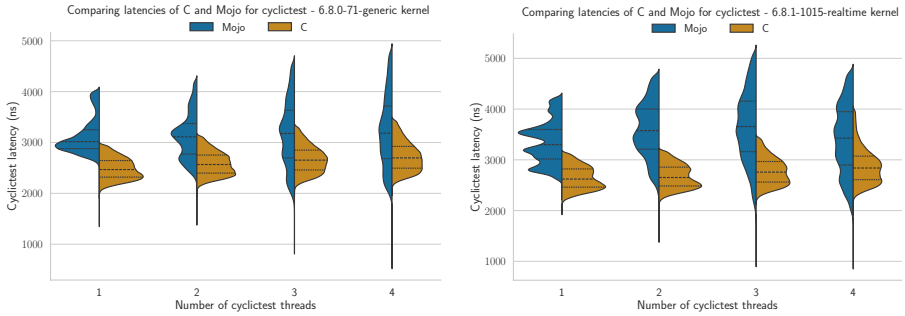
Concurrency was implemented using Mojo's `parallelize` construct, which spawns a specified number of work items. Empirical observation suggests that the current Mojo runtime employs a preallocated thread pool limited to four threads, constraining the maximum degree of concurrency in our tests. Thread priority was configured via `sched_setscheduler` by using an external call. The `SCHED_FIFO` was set at priority 99 to match the original `cyclictest` settings.

In addition, to reduce latency spikes from major page faults, we implemented a memory-locking mechanism analogous to `cyclictest`'s `--mlockall` option. The program invokes the `mlockall` system call at startup and `munlockall()` on exit using Mojo's `sys.external_call` interface. This ensured that all current and future memory allocations remained resident in RAM for the duration of the test.

Finally, `cyclictest` typically reduces device latency by opening the file `/dev/cpu_dma_latency` and writing a zero value for the duration of the test. As this operation is not currently possible in Mojo (see Section 4.1), we implemented a workaround by wrapping the Mojo `cyclictest` invocation in a small Python script that performs the write and keeps the file open for the entire execution. Although this solution introduces an external dependency and is not ideal, it preserves parity in kernel configuration between the Mojo and C measurements.

Background load. To assess latency robustness under CPU contention, all measurements were conducted while maintaining a constant background load. The load was generated using `stress-ng`, a widely used tool for exercising and benchmarking various system subsystems [23,21,14]. We used the following configuration: `stress-ng -cpu 0 -cpu-method all -t 65`. This configuration continuously stresses all available CPU cores with a variety of computational methods, ensuring that the kernel scheduler and runtime contend with realistic interference while still meeting the `cyclictest` deadlines.

Experimental setup. We ran `cyclictest` on the same hardware platform as the microbenchmarks. Instead of pinning the `cyclictest` processes, we disabled all "small" E-cores using the `isolcpus` boot argument (i.e., `isolcpus=12-19`).



(a) 6.8.0-71-generic kernel (b) 6.8.1-1015-realtime kernel

Fig. 4. Wake-up latencies for C vs. Mojo `cyclictest` with `lock_memory_alloc` enabled on (a) generic and (b) real-time kernels (with `PREEMPT_RT`). Mojo shows higher median latency and greater variability across all thread counts.

Results. Figures 4a and 4b compare wake-up latencies for the C and Mojo implementations of `cyclictest` with `lock_memory_alloc` enabled, on Linux `6.8.0-71-generic` and `6.8.1-1015-realtime` (i.e., with the `PREEMPT_RT` patch enabled) kernels, respectively. In both kernels, Mojo exhibits consistently higher median latency and greater variability than the C baseline across all tested thread counts (1–4). The difference is most pronounced in the single-threaded case, where Mojo's median latency exceeds C's by several hundred microseconds and its tail latencies are significantly higher, indicating more variance. The relative gap between Mojo and C is substantial, suggesting that page-fault avoidance does not fully offset Mojo's runtime overheads. The broader latency distributions for Mojo—especially visible in the real-time kernel results—point to additional sources of unpredictability, such as runtime scheduling effects, FFI indirection for system calls, and potentially less deterministic internal threading. These findings reinforce our earlier conclusion: while Mojo can handle soft real-time workloads, its current runtime behavior limits its suitability for hard real-time systems where tight latency bounds are critical.

GPU Execution Variability (RQ3). To address **RQ3** (timing stability of Mojo's GPU execution model), we implemented a square matrix multiplication kernel in Mojo for both CPU and GPU targets. The CPU version uses a straightforward triple-nested loop over `Int` values, while the GPU version is launched via Mojo's host–device API (`DeviceContext` and `enqueue_function`) using a grid–block–thread execution model (`block_idx`, `thread_idx`, `block_dim`, `grid_dim`) with device buffers for inputs and outputs. We compared execution-time distributions between CPU and GPU runs for different matrix sizes to assess variability and potential sources of unpredictability.

Experiments were conducted on a Dell Precision 5480 equipped with an Intel Core i7–13800H CPU running the `6.14.0-33-generic` Linux kernel and an

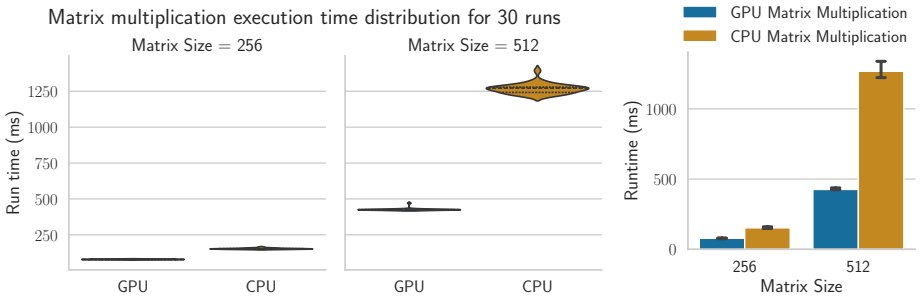


Fig. 5. GPU vs. CPU matrix multiplication in Mojo. Left: runtime distributions over 30 runs at sizes 256×256 and 512×512. Right: aggregated runtimes with error bars showing variability.

NVIDIA RTX 2000 Ada GPU with CUDA 13.0 and driver version 580.95.05. We measured 30 executions for each matrix size (256×256 and 512×512). Figure 5 reports the resulting distributions and aggregated execution times.

Findings. For 256×256, GPU and CPU runtimes are of the same order, with the GPU showing slightly *lower* variance. For 512×512, the GPU is faster on average and maintains a narrower runtime distribution than the CPU. This stability suggests that most variability comes from CPU-side factors rather than device scheduling or host–device transfers in our setup.

Current tooling limitations. Mojo's GPU API currently lacks clearly documented controls that are important for predictability, such as: (i) selecting a specific GPU device in multi-GPU setups, (ii) setting stream or queue priorities, and (iii) overlapping or explicitly scheduling memory transfers. In the absence of these controls or their documentation, bounding worst-case execution times remains challenging even when GPU performance is superior.

5 Related Work

5.1 Languages for Real-Time and Embedded Systems

Real-time systems impose strict timing and predictability requirements. The choice of programming language can significantly affect a system's ability to meet deadlines, avoid unpredictable delays, and facilitate timing analysis. The real-time research community has studied how language features impact determinism, scheduling analyzability, and worst-case execution time (WCET) predictability.

A suitable real-time language should support predictable timing and bounded resource usage [9]. Important criteria for hard real-time languages include determinism, bounded time/space, and responsiveness to external events without undue latency. Meeting these criteria often means avoiding features that introduce

unbounded or hard-to-predict delays (e.g., garbage collection pauses, dynamic dispatch) and restricting dynamic memory allocation or unbounded recursion. Coding standards such as MISRA C [31] or the Ravenscar Profile for Ada [8] embody these restrictions.

C and C++. C remains the de facto standard for real-time and embedded systems, due to its efficiency, low-level hardware access, and minimal runtime overhead [16]. While it allows precise control over timing, it provides no built-in real-time semantics, placing the burden on the programmer to ensure predictability. Guidelines such as MISRA C help mitigate unsafe patterns, and formally verified toolchains like CompCert [27] further improve trustworthiness. However, challenges related to memory safety and analyzability remain.

Ada. Ada was designed with built-in real-time features, including a tasking model, priority scheduling, timing events, and protected objects. The Ravenscar Profile [8] defines a restricted subset of Ada tasking to guarantee determinism and analyzability. Ada has been widely adopted in safety-critical domains for its strong typing and integrated real-time primitives.

Real-Time Java. Java's original memory model and garbage collection mechanisms conflicted with hard real-time needs. The Real-Time Specification for Java (RTSJ) introduced scoped memory, `NoHeapRealtimeThreads`, and fully preemptive priority-based scheduling [6]. Studies such as Pizlo et al. [33] showed that scoped memory significantly reduces variability compared to real-time garbage collection, at the cost of increased programmer responsibility.

Rust and emerging languages. Rust offers memory safety without garbage collection, using a compile-time ownership model. Recent work [10] has explored mapping real-time task scheduling patterns to Rust's type system, enabling safe concurrency in embedded contexts. Zig and similar emerging system languages emphasize manual memory management and minimal runtime overhead—features that align with real-time system requirements. However, to the best of our knowledge, there are no peer-reviewed empirical evaluations of Zig in hard real-time or safety-critical settings.

Domain-Specific Languages. Beyond these mainstream languages, synchronous programming languages (e.g., Lustre, Esterel, Signal) [7] and the Logical Execution Time paradigm [18] demonstrate that restricting concurrency and timing to formal models can yield analyzable real-time programs.

5.2 Mojo in Systems Programming

Mojo [24], introduced by Modular Inc. in 2023, seeks to bridge Python's accessibility with the control and performance of systems languages. Outside of AI and high-performance computing, published work on Mojo is scarce. A small number of academic studies exist, such as MojoBench [36], a benchmark suite

for LLM code generation, and MojoFrame [19], a dataframe library showcasing performance optimizations for relational operations. Practitioner sources—including Modular blog posts [34] and community discussions [35]—cover early experiments and anecdotal performance claims. However, none of these works assess Mojo's suitability for hard real-time constraints, embedded deployment, or WCET analysis. Our work is, to our knowledge, the first evaluation in this space.

5.3 Benchmarking Methodologies for Language-Level Evaluation

Cross-language benchmarking requires careful control of algorithmic equivalence, compiler settings, and confounding factors to produce valid comparisons. Ivanov [20] evaluates Rust and C++ implementations of everyday computational routines, finding that Rust can match or even exceed C++ performance when similar low-level optimizations are applied. Alomari et al. [2] compare six popular languages across metrics such as execution time, memory usage, and code size, revealing substantial trade-offs between performance and other software qualities. Nanz and Furia [32] use the Rosetta Code repository to conduct a large-scale comparison of eight languages, measuring conciseness, runtime efficiency, and failure rates across a diverse set of tasks. While these studies focus primarily on average-case throughput and developer productivity, our methodology adapts such comparative frameworks to real-time contexts, emphasizing metrics critical to predictability: worst-case latency, variance under interference, and binary size.

5.4 GPU Predictability in Real-Time Systems

GPU acceleration has been integrated into real-time domains such as robotics and signal processing, but studies show substantial execution-time variability due to device scheduling, stream interference, and transfer overheads [22,37]. Approaches to mitigate this include partitioning [4,5] and priority-based scheduling [17,15]. While these have been studied for CUDA and OpenCL, GPU Mojo's abstractions have not been evaluated for timing predictability.

5.5 Timing Tests and `cyclictest` Studies

`cyclictest` [39] is the de facto standard for measuring OS-level wake-up latency. It has been used to assess PREEMPT_RT Linux in multiple contexts [11,14]. However, no prior work has reimplemented `cyclictest` in another language to isolate language runtime effects on latency. By doing so in Mojo, we quantify the additional latency and variability introduced by the language's runtime and interoperability layers.

6 Conclusion

This paper presented the first systematic evaluation of the Mojo programming language for real-time and embedded system applications. We introduced a reproducible, language-level methodology to assess readiness for time-sensitive domains, covering latency, execution-time variability, binary size, GPU control, and predictability under scheduler and interference conditions. The methodology was applied to representative CPU microbenchmarks in Mojo, C, C++, and Rust; to a reimplementation of `cyclictest` in Mojo compared against the canonical C version; and to GPU kernels measuring execution-time stability. Our results indicate that Mojo can match C and Rust in arithmetic-heavy workloads, offers stable GPU runtimes for regular kernels, and provides high-level abstractions that may lower the entry barrier for developers from Python-centric backgrounds.

Our experiments also exposed a number of limitations in the current state of Mojo that restrict its applicability in hard real-time and low-level embedded contexts. The language and toolchain do not yet offer fine-grained control over scheduling parameters, CPU affinity, or device latency settings, and lack stable primitives for pointer manipulation, volatile and direct memory access, and interrupt handling. Compiler optimizations fall short for branch-heavy and large memory-bound workloads, while GPU support omits predictability-oriented features such as device selection and stream priorities. Official targets remain limited to `x86_64` Linux, with only experimental cross-compilation to other architectures and no RTOS integration yet.

Beyond the language itself, our study has methodological limitations. The `cyclictest` reimplementation was carried out only in Mojo, meaning language-level runtime effects were compared solely against C. The feasibility studies for robotics and embedded integration were exploratory rather than full production deployments, WCET analysis was not performed, and CPU/GPU experiments focused on a small, regular kernel. The evaluation also did not consider the commercial Modular MAX runtime, and all benchmarks were run on `x86_64` platforms, leaving open questions about generality across architectures.

Future work can proceed along two lines. On the Mojo side, extending the standard library and FFI to expose low-level system calls, providing stable embedded-oriented primitives, and integrating with WCET analysis tools would strengthen its real-time potential. Optimizing for irregular control flow and memory-intensive kernels, expanding official deployment targets to Arm, bare-metal, and RTOS environments, and improving GPU predictability controls are likewise promising. On the methodological side, applying the proposed evaluation framework to other emerging systems languages (such as Zig or Carbon) and to established languages in novel execution contexts would broaden its relevance in real-time systems research.

In summary, while Mojo already demonstrates the potential to bridge high-level expressiveness with low-level performance, it requires significant evolution before it can be considered a mature option for safety-critical, timing-sensitive systems. The methodology developed here provides both a roadmap for that

evolution and a template for evaluating other languages in the same demanding domain.

Acknowledgments. This work was supported by the Cybersecurity Research Program Flanders (CRPF). We thank Chris Lattner and his team at Modular for their constructive feedback on the manuscript.

References

1. Adam, G.K., Petrellis, N., Doulos, L.T.: Performance Assessment of Linux Kernels with PREEMPT_RT on ARM-Based Embedded Devices. Electronics **10**(11), 1331 (2021)
2. Alomari, Z., Halimi, O.E., Sivaprasad, K., Pandit, C.: Comparative Studies of Six Programming Languages (2015), https://arxiv.org/abs/1504.00693
3. Antonio Paolillo and Aaron Bogaert: MojoRT: Evaluation Artifacts for "Is Mojo Ready for Real-Time? A Language Evaluation for Time-Sensitive System Software". https://github.com/apaolillo/mojort
4. Bakita, J., Anderson, J.H.: Hardware Compute Partitioning on NVIDIA GPUs. In: 2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 54–66 (2023). https://doi.org/10.1109/RTAS58335.2023.00012
5. Bakita, J., Anderson, J.H.: Hardware Compute Partitioning on NVIDIA GPUs for Composable Systems. In: Mancuso, R. (ed.) 37th Euromicro Conference on Real-Time Systems (ECRTS 2025). Leibniz International Proceedings in Informatics (LIPIcs), vol. 335, pp. 21:1–21:25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2025). https://doi.org/10.4230/LIPIcs.ECRTS.2025.21
6. Belliardi, R., Brosgol, B., Dibble, P., Holmes, D., Wellings, A., Bollella, G., Furr, S., Gosling, J., Hardin, D., Turnbull, M.: The Real-Time Specification for Java, Version 1.0.2. The Real-Time for Java Expert Group, Pittsburgh, PA, USA (2006), first printing, January 2006
7. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., de Simone, R.: The Synchronous Languages 12 Years Later. Proceedings of the IEEE **91**(1), 64–83 (2003). https://doi.org/10.1109/JPROC.2002.805826
8. Burns, A., Dobbing, B., Vardanega, T.: Guide for the use of the Ada Ravenscar Profile in high integrity systems. Ada Lett. **XXIV**(2), 1–74 (Jun 2004). https://doi.org/10.1145/997119.997120
9. Burns, A., Wellings, A.J.: Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX (2001)
10. Carvalho, T., Silva, H., Miguel Pinho, L.: A Real-Time Parallel Programming Approach for Rust. Ada Lett. **43**(2), 57–61 (Jun 2024). https://doi.org/10.1145/3672359.3672366
11. Cerqueira, F., Brandenburg, B.B.: A Comparison of Scheduling Latency in Linux, PREEMPT_RT, and LITMUSRT. In: Proceedings of the 9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2013). Paris, France (Jul 2013), held in conjunction with the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)
12. Cherickal, T.: Meet Mojo: The Language That Could Replace Python C and CUDA – The Reality Check: What Mojo Can't Do Yet – But Will With Time (June 2025), https://hackernoon.com/meet-mojo-the-language-that-could-replace-python-c-and-cuda, accessed: 2025-08-14

13. Delgado, R., Choi, B.W.: New Insights Into the Real-Time Performance of a Multicore Processor. Ieee Access **8**, 186199–186211 (2020)
14. Dewit, W., Paolillo, A., Goossens, J.: A Preliminary Assessment of the real-time capabilities of Real-Time Linux on Raspberry Pi 5. In: Proceedings of the 18th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2024). pp. 7–12. Lille, France (July 2024), https://www.ecrts.org/wp-content/uploads/2024/07/ospert24-proceedings.pdf
15. Discepoli, A., Huygen, M.L., Paolillo, A.: Compute Kernels as Moldable Tasks: Towards Real-Time Gang Scheduling in GPUs. In: Proceedings of the 19th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2025). pp. 29–33. Brussels, Belgium (July 2025), https://www.ecrts.org/wp-content/uploads/2025/07/ospert25-proceedings.pdf
16. Ebert, C., Jones, C.: Embedded Software: Facts, Figures, and Future. Computer **42**(4), 42–52 (2009). https://doi.org/10.1109/MC.2009.118
17. Fan, R., Ren, T., Xie, M., Gao, S., Shu, J., Lu, Y.: GPREEMPT: GPU Preemptive Scheduling Made General and Efficient. In: Proceedings of the 2025 USENIX Annual Technical Conference (ATC '25). USENIX Association, Boston, MA, USA (July 2025)
18. Henzinger, T., Horowitz, B., Kirsch, C.: Giotto: A Time-Triggered Language for Embedded Programming. Proceedings of the IEEE **91**(1), 84–99 (2003). https://doi.org/10.1109/JPROC.2002.805825
19. Huang, S., Li, Z., Werner, D., Park, Y.: MojoFrame: Dataframe Library in Mojo Language (2025), https://arxiv.org/abs/2505.04080
20. Ivanov, N.: Is Rust C++-fast? Benchmarking System Languages on Everyday Routines (2022), https://arxiv.org/abs/2209.09127
21. Jo, Y.H., Choi, B.W.: Performance evaluation of real-time linux for an industrial real-time platform. International journal of advanced smart convergence **11**(1), 28–35 (2022)
22. Kato, S., Lakshmanan, K., Rajkumar, R., Ishikawa, Y.: TimeGraph: GPU scheduling for real-time multi-tasking environments. In: Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference. p. 2. USENIXATC'11, USENIX Association, USA (2011)
23. King, C.I.: stress-ng. https://github.com/ColinIanKing/stress-ng, accessed 2025-10-29
24. Krill, P.: Mojo language marries Python and MLIR for AI development (5 2023), https://www.infoworld.com/article/2338436/mojo-language-marries-python-and-mlir-for-ai-development.html, accessed: 2025-08-14
25. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., Zinenko, O.: MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 2–14 (2021). https://doi.org/10.1109/CGO51591.2021.9370308
26. Lattner, C. and Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004. pp. 75–86 (2004). https://doi.org/10.1109/CGO.2004.1281665
27. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: CompCert - A Formally Verified Optimizing Compiler. In: ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress. SEE, Toulouse, France (Jan 2016), https://inria.hal.science/hal-01238879

28. Lindvig, A.P., Iturrate, I., Kindler, U., Sloth, C.: ur_rtde: An Interface for Controlling Universal Robots (UR) using the Real-Time Data Exchange (RTDE). In: 2025 IEEE/SICE International Symposium on System Integration (SII). pp. 1118–1123 (2025). https://doi.org/10.1109/SII59315.2025.10871000
29. Litayem, N., Saoud, S.B.: Impact of the Linux real-time enhancements on the system performances for multi-core intel architectures. International Journal of Computer Applications **17**(3), 17–23 (2011)
30. Modular Inc.: Install Mojo (Sep 2025), https://docs.modular.com/mojo/manual/install/, online installation guide for the Mojo programming language
31. Motor Industry Software Reliability Association: MISRA-C:2012 – Guidelines for the use of the C language in critical systems, 3rd edn. (2019)
32. Nanz, S., Furia, C.A.: A Comparative Study of Programming Languages in Rosetta Code. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. p. 778–788. IEEE (May 2015). https://doi.org/10.1109/icse.2015.90, http://dx.doi.org/10.1109/ICSE.2015.90
33. Pizlo, F., Vitek, J.: An Emprical Evaluation of Memory Management Alternatives for Real-Time Java. In: 2006 27th IEEE International Real-Time Systems Symposium (RTSS'06). pp. 35–46 (2006). https://doi.org/10.1109/RTSS.2006.9
34. Prasanna, S.: Fast k-means clustering in Mojo: a guide to porting Python to Mojo for accelerated k-means clustering (5 2024), https://www.modular.com/blog/fast-k-means-clustering-in-mojo-guide-to-porting-python-to-mojo-for-accelerated-k-means-clustering, accessed: 2025-08-14
35. Ragazzi, D.: Creating a tiny operating system in Mojo (July 2024), https://github.com/modularml/mojo/discussions/3334, accessed: 2025-08-14
36. Raihan, N., Santos, J.C.S., Zampieri, M.: MojoBench: Language Modeling and Benchmarks for Mojo (2024), https://arxiv.org/abs/2410.17736
37. Shen, Y., Mynsbrugge, J.V., Roshandel, N., Bouchez, R., FirouziPouyaei, H., Scholz, C., Cao, H.L., Vanderborght, B., Joosen, W., Paolillo, A.: SentryRT-1: A Case Study in Evaluating Real-Time Linux for Safety-Critical Robotic Perception. In: Proceedings of the 19th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2025). pp. 35–41. Brussels, Belgium (July 2025), https://www.ecrts.org/wp-content/uploads/2025/07/ospert25-proceedings.pdf
38. TechSkillGuru: Advantages and Disadvantages of Mojo Programming (2024), https://techskillguru.com/mojo/advantages-and-disadvantages-of-mojo, accessed: 2025-08-14
39. The-Linux-Foundation: Cyclictest — realtime documentation howto tools. Online (2025), https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start, online; Accessed 2025-08-07