Prototype Preemptive Threading in Wasmtime

Jerry Yang
CSE M.S.
University of California, San Diego
San Diego, California
jiy112@ucsd.edu

Blake Muxlow
CSE M.S.
University of California, San Diego
San Diego, California
bmuxlow@ucsd.edu

***Abstract*—WebAssembly (Wasm) is increasingly used in environments requiring robust concurrency, yet most standalone runtimes offer limited scheduling capabilities. Wasmtime, a leading Wasm runtime, provides cooperative threading, which relies on guest programs to explicitly yield control. This paper presents the design and implementation of a preemptive scheduler for Wasmtime's lightweight fibers, enabling automatic context switching without guest cooperation.**

We explore two independant mechanisms for forcing preemption: epoch-based interruption (time-slicing) and fuel-based metering (gas-instrumentation). Our implementation repurposes these existing runtime features to suspend a running fiber and return control to a round-robin scheduler, which then resumes the next fiber in its queue. A key challenge was enabling yields from synchronous contexts, which we solved by using thread-local storage to track and suspend the active blocking context.

We evaluate the performance overhead of both preemption strategies across three distinct workloads: recursive (Fibonacci), memory-intensive (Dot Product), and mixed computation (Matrix Multiplication). Our results demonstrate that the overhead is highly workload-dependent. Epoch-based preemption excels for recursive tasks with near-zero overhead, while fuel-based preemption is more efficient for compute-heavy matrix operations. Both methods introduce significant overhead for tight, memory-bound loops. This work validates the feasibility of preemptive scheduling in Wasmtime and concludes that the optimal preemption strategy depends on the target application, justifying our configurable design.

***Index Terms*—WebAssembly, Multithreading, Preemption, Cooperative Threading, Wasmtime, Fibers**

I. Introduction

Preemptive scheduling is a cornerstone of modern operating systems and runtimes, ensuring fairness and bounded latency by forcibly interrupting running tasks to allocate resources to others. This prevents any single compute-heavy or misbehaving task from monopolizing the system. As WebAssembly (Wasm) expands from a browser sandbox to server-side and cloud-native environments, the demand for robust, preemptive concurrency models has intensified.

However, the majority of standalone WebAssembly runtimes, including Wasmtime, primarily support cooperative multitasking. In this model, tasks must explicitly and voluntarily yield control. While simpler to implement, this approach cannot guarantee fairness, as a non-cooperative guest program can starve all other tasks of execution time. The WebAssembly specification itself does not define threading or scheduling mechanisms, leaving these critical features to the host environment.

This paper addresses these limitations by designing and implementing a preemptive threading capability within Wasmtime, a leading runtime developed by the Bytecode Alliance. Our work leverages Wasmtime's lightweight, stack-switching fibers to build a scheduler that can automatically preempt guest code without requiring any modification or explicit yield points. We explore and compare two distinct interruption mechanisms to trigger preemption:

- **Epoch-based interruption**, which enables time-based slicing.
- **Fuel-based metering**, which provides deterministic, instruction count-based slicing.

Our contributions demonstrate that preemptive scheduling is feasible at the fiber level within Wasmtime's architecture. We present a detailed analysis of the performance trade-offs between the two approaches, showing that the optimal strategy is workload-dependent. This work provides a practical foundation for building more advanced, isolation-focused scheduling policies in future WebAssembly runtimes.

II. Background

A. The WebAssembly Concurrency Model

WebAssembly is a portable, sandboxed binary instruction format designed as a high-performance compilation target for languages like C++, Rust, and Go. While the core Wasm specification is single-threaded, the WebAssembly Threads proposal extended it with concurrency primitives. Crucially, this proposal focuses exclusively on enabling parallel computation through shared memory. It introduces WebAssembly.SharedMemory and a set of atomic instructions (Atomics) that mirror those found in modern CPUs, providing the necessary tools for developers to build thread-safe applications.

However, the specification deliberately leaves the management of threads—their creation, scheduling, and lifecycle—to the host environment (e.g., a web browser or a standalone runtime like Wasmtime). This design choice makes Wasm highly flexible but also places the burden of implementing a robust concurrency model on the runtime. Without a host-provided scheduler, Wasm modules can only achieve concurrency through cooperative multitasking, where guest code must be explicitly written to yield control, or by relying on host-provided asynchronous APIs.

B. The Wasmtime Runtime and its Fiber Architecture

Wasmtime is a leading, open-source WebAssembly runtime maintained by the Bytecode Alliance. It is engineered for performance and security, compiling Wasm modules to highly optimized native machine code using its Cranelift JIT compiler. A key architectural feature of Wasmtime, and the foundation upon which our work is built, is its use of lightweight, stack-switching **fibers**.

Unlike traditional operating system threads, which require kernel-level context switching, fibers are managed entirely in user space. Each fiber maintains its own separate stack and register context. This allows Wasmtime to suspend the execution of a Wasm function by saving its current state and switch to another fiber by restoring its state, all with minimal overhead. This model is ideal for managing many concurrent Wasm instances efficiently, but by default, a switch only occurs when the guest code performs an await on an asynchronous host operation. A running fiber executing a tight, compute-bound loop will monopolize its OS thread indefinitely, highlighting the need for a mechanism to force a context switch.

C. Host-Controlled Interruption Primitives

While Wasmtime's default model is cooperative, it provides two powerful, host-controlled primitives for interrupting Wasm execution. These tools were designed for timeout and resource control, but we repurposed them to serve as the triggers for our preemptive scheduler.

a) *Epoch-Based Interruption:* An "epoch" is a global, 64-bit counter on the Wasmtime engine. The host can increment this counter at any time, typically on a regular interval using a timer. Cranelift instruments the compiled Wasm code to check the value of this counter at strategic points, such as function prologues and loop backedges. The host can set a deadline; if the Wasm code detects that the current epoch has surpassed the deadline, it triggers a trap or invokes a host-defined callback. This mechanism was introduced by Jamey Sharp primarily to implement execution timeouts with very low performance overhead, estimated to be around 10%. [1]

b) *Fuel Metering:* Fuel metering provides a more deterministic method of controlling execution. The host provides a Wasm Store with a fixed amount of "fuel." Cranelift instruments the compiled code to decrement a fuel counter at every function call and loop iteration, with the amount consumed being proportional to the estimated instruction cost of the code block. If the fuel counter drops to zero, a trap is triggered. This mechanism is ideal for preventing denial-of-service attacks and for metered billing (akin to "gas" in blockchain systems), as it deterministically ties execution to a quantifiable resource. However, this determinism comes at a cost; the additional instrumentation incurs a significant performance overhead, measured by Sergei Shulepov to be between 24-34% compared to unmetered execution. [2]

III. Design and Implementation

Our primary design goal was to evolve Wasmtime's concurrency model from a purely cooperative to a preemptive one, without altering the guest Wasm contract. This required building a scheduler that could forcibly interrupt and context-switch fibers. The implementation journey involved designing a scheduler architecture, exploring two preemption strategies, and solving a critical bug in Wasmtime's context management.

A. Scheduler Architecture

We began by introducing a PreemptiveThreads struct integrated directly into the Wasmtime Store. This location was chosen because the scheduler needs intimate access to the store's fiber management and execution state. The struct contains the core components of a simple scheduler: a HashMap mapping thread IDs to StoreFiber objects for efficient lookup, and a VecDeque acting as a FIFO run queue to implement a round-robin policy.

To make this functionality accessible, we exposed a clean public API on the Store:

- spawn_wasm_thread(): Takes a Wasm function and its parameters, creates a new fiber to execute it, assigns the fiber a unique ID, and enqueues it for scheduling.
- run_wasm_threads_for(): Activates the main scheduler loop for a specified duration. This loop continuously dequeues a fiber, resumes its execution, and, upon a yield, re-enqueues it.
- shutdown_wasm_threads(): Provides a graceful exit path, halting the scheduler and calling dispose() on all active fibers to release their stack memory and prevent resource leaks.

B. An Evolution of Preemption Strategies

The core implementation challenge was forcing a running fiber to yield control. We repurposed Wasmtime's two interruption mechanisms, leading to a configurable, dual-mode design.

a) *First Attempt: Epoch-Based Preemption:* Our initial prototype used epoch-based interruption, as it is the lower-overhead mechanism. We configured a deadline and installed a callback that would trigger a yield. To drive the interruptions, we spawned a background "ticker" thread responsible for periodically calling engine.increment_epoch().

However, this approach revealed several fundamental problems:

1) **Unpredictable Timeslicing:** The actual preemption interval depended on the execution speed of our scheduler loop and the OS scheduling of the ticker thread, not on a consistent wall-clock time.
2) **Background Thread Overhead:** The ticker thread consumed system resources even when no Wasm code was running.
3) **Implementation Complexity:** Managing the lifetimes and state-sharing for the callback was complex and error-prone.

These issues made the epoch-based approach non-deterministic and inefficient for our purposes, motivating a search for a better solution.

b) *Second Attempt: Fuel-Based Preemption:* We then turned to fuel metering. The key insight was to leverage the

fuel_async_yield_interval feature not for its intended purpose of trapping, but to trigger a non-trapping, asynchronous yield. This provided the determinism that epochs lacked.

This pivot dramatically simplified the design. We removed the background ticker thread entirely. The scheduler loop no longer needed to manually manage epochs. Instead, we modified the out_of_gas libcall, which is invoked automatically when a fiber's fuel slice is consumed. Inside this handler, we prevented the default Trap::OutOfFuel and instead performed two actions: we refilled the fiber's fuel to u64::MAX to allow it to continue later, and then we invoked a custom preemptive_yield() function to suspend the fiber and return control to the scheduler. This approach was simpler, more self-contained, and provided reproducible, instruction-count-based preemption.

C. Solving the Blocking Context Problem

With the fuel-based approach, we encountered a subtle but critical bug: threads were not actually yielding. The out_of_gas libcall executes in a synchronous context, but Wasmtime's fiber suspension API was designed to be called from an async context. Our preemptive_yield function would check for an async context, find none, and silently return without suspending the fiber. The symptom was that each thread would run for one timeslice and then appear to "complete" because it was never re-enqueued.

We engineered a robust solution by modifying Wasmtime's core context management using **thread-local storage (TLS)**.

1) We declared a new thread-local Cell to hold a raw pointer to the current BlockingContext.
2) We instrumented the with_blocking function, which sets up a synchronous execution environment, to store a pointer to its context in this TLS variable upon entry and restore the previous value upon exit. This correctly handles nested contexts.
3) We updated our preemptive_yield function with a crucial fallback: if it fails to find an async context, it now checks the TLS for a BlockingContext. If one is present, it safely uses that context to suspend the fiber.

This TLS-based solution bridged the gap between Wasmtime's synchronous and asynchronous worlds, ensuring that preemptive yields could be triggered reliably from any execution state.

IV. Evaluation

To measure the performance impact of our scheduler, we conducted benchmarks using three distinct workloads, each designed to stress a different aspect of the runtime:

- **Fibonacci:** A recursive, CPU-bound task (fib(32)).
- **Dot Product:** A memory-intensive, tight-loop operation on two vectors of 4096x4096 f64 elements.
- **Matrix Multiplication:** A mixed computation and memory-access task (512x512 f64 matrices).

We compared the execution time of both Fuel and Epoch preemption modes against a baseline of direct, uninterrupted execution.

A. Results

Our benchmark results reveal that the performance overhead of preemption is highly dependent on the nature of the workload.

TABLE I
Performance overhead of preemptive scheduling modes across different workloads. Lower overhead is better. Best-performing preemption mode for each workload is marked in italics.

Workload	Mode	Mean Time (ms)	Overhead
Fibonacci	Baseline	36.554	1.00x
	Fuel	43.519	1.19x
	Epoch	35.404	*0.97x*
Dot Product	Baseline	37.543	1.00x
	Fuel	55.691	1.48x
	Epoch	55.838	1.49x
Matrix Multiply	Baseline	417.007	1.00x
	Fuel	520.691	*1.25x*
	Epoch	630.010	1.51x

As shown in Table I, the results highlight a clear trade-off:

- For the recursive **Fibonacci** workload, epoch-based interruption is remarkably efficient, exhibiting virtually no overhead. In fact, it consistently performed slightly faster than the baseline, an anomaly likely due to measurement noise. Fuel-based preemption incurred a consistent 19% overhead.
- For the memory-bound **Dot Product** workload, both preemption methods performed poorly, adding nearly 50% overhead. This suggests that the cost of the frequent interruption checks in a tight, memory-intensive loop outweighs the benefits of preemption.
- For the mixed-computation **Matrix Multiplication** workload, fuel-based preemption was significantly more efficient, with 25% overhead compared to over 50% for the epoch-based method.

These findings strongly support our design choice to make the preemption mode configurable, as the optimal strategy is clearly application-dependent.

V. Related Work

The concept of threading in Wasmtime has evolved over time. Initially, concurrency relied on fully cooperative multitasking, where guest modules had to be written with async support and explicitly yield. The introduction of epoch-based interruption by Jamey Sharp was primarily for execution timeouts, not general-purpose scheduling. [1] Similarly, Tyler Rockwood refactored the fuel metering API to be more ergonomic for resource limiting. [3] Our work is the first to combine these mechanisms into a cohesive preemptive scheduler.

The performance overhead of fuel metering has been a topic of extensive discussion in the Wasmtime community. Measurements by Sergei Shulepov indicated an overhead of

24-34% for fuel, compared to 10% for epochs, which aligns with our findings for certain workloads. [2] Shulepov also proposed a "slacked fuel metering" optimization to reduce this cost, but it has not been implemented due to its complexity, which would involve intricate signal handling and potential ABI changes.

It is important to note that production systems built on Wasmtime, such as wasmCloud and Fastly's Compute@Edge, typically use epoch-based interruption. They favor its lower general-purpose overhead and accept its non-determinism, as their actor and serverless models often consist of short-lived computations where precise, instruction-level fairness is less critical. [4] While engineers from companies like Cosmonic have contributed to Wasmtime, their work has focused on areas like WASI and the component model, rather than fundamental fuel optimizations.

VI. Conclusion and Future Work

We have successfully designed and implemented a preemptive, fiber-based scheduler in Wasmtime, demonstrating that fair, automatic scheduling is viable without requiring guest code cooperation. Our exploration of both epoch-based and fuel-based preemption revealed a critical trade-off between performance, determinism, and workload suitability, underscoring the importance of a configurable approach. The technical solution of using thread-local storage to handle yields from synchronous contexts represents a key contribution to enabling robust preemption.

Our work provides a solid foundation for more advanced concurrency models in WebAssembly. Several avenues for future work remain. First, the scheduler currently uses a simple round-robin policy; implementing priority-based or weighted fair queuing would allow for more sophisticated resource management. Second, the timeslice for preemption is currently a compile-time constant and should be exposed as a configurable runtime parameter. Finally, a known issue where the runtime panics on shutdown due to improper fiber state cleanup needs to be resolved before this implementation could be considered for upstream inclusion in Wasmtime.

Ultimately, this project confirms that preemptive scheduling can be effectively integrated into WebAssembly runtimes, paving the way for more resilient, predictable, and isolated execution environments.

References

[1] J. Sharp, "Implement epoch-based interruption." [Online]. Available: https://github.com/bytecodealliance/wasmtime/pull/6464

[2] S. Shulepov, "High overhead of fuel-based interruption." [Online]. Available: https://github.com/bytecodealliance/wasmtime/issues/4109

[3] T. Rockwood, "Refactor fuel API." [Online]. Available: https://github.com/bytecodealliance/wasmtime/pull/7298

[4] Cosmonic, "Gumball Deconstructed: A Performance Deep Dive." [Online]. Available: https://wasmcloud.com/blog/gumball-deconstructed-a-performance-deep-dive/