ChatGPT

Comprehensive Dual-Track Profiling & Benchmarking Strategy for `lit-bit`

Overview: Platform-Dual Profiling Approach

`lit-bit` is a Rust actor system designed for both **embedded (`no_std`)** and **host (`std`)** environments. Profiling and benchmarking must therefore address two distinct tracks:

- **Embedded (`no_std`) Profiling:** Focus on memory usage (static and dynamic), real-time latencies, and power-oriented metrics, all with minimal runtime overhead.
- **Host (`std`) Profiling:** Deep analysis of memory allocation patterns, fragmentation, cache usage, and multi-threading effects in an async runtime.

Benchmark Harness Requirements: To compare sync vs. async and embedded vs. host fairly, the benchmarking setup should be **runtime-agnostic** and consistent across environments. This means using lightweight executors and avoiding Tokio-specific issues (like nested runtime panics). Harnesses should allow running the same actor workloads on different runtimes (e.g. Tokio, futures-lite, Embassy) for apples-to-apples comparisons.

Below we detail the strategy for each track, recommend tools/techniques, and highlight best practices (including insights from Akka/Erlang actor systems).

1. Embedded (`no_std`) Profiling Goals

For embedded targets (e.g. Cortex-M microcontrollers or RISC-V MCUs), the profiling must be extremely low-overhead and often use hardware support. Key goals include **static memory analysis, peak heap usage, interrupt/task latency,** and **power implications**:

1.1 Static Memory Usage Analysis

- **What to Measure:** Flash and RAM footprint of the actor system – sizes of code (`.text`), static data (`.data`), and zero-initialized data (`.bss`). Also worst-case stack usage per actor/task.
- **Tools & Methods:**
- Use `cargo size` or `llvm-objdump` to inspect binary section sizes. This gives a breakdown of static memory usage (e.g. ensuring the core `lit-bit` library remains lightweight for MCU flash).
- Leverage static analysis tools like `cargo-call-stack` to estimate maximum stack usage [1]. This tool analyzes call graphs to detect deep recursion or large stack frames, crucial for preventing stack overflows in `no_std` contexts.
- Monitor **unsafe code** and hidden allocations: Ensure no unexpected std usage creeping in (the CI uses a “heap-safety-scan” to confirm no `std` in `lit-bit-core` unless feature-gated [2]).
- **Best Practices:**

- **Feature-gate allocations:** Keep the core actor framework free of heap usage unless an `alloc` feature is enabled, so static memory use is predictable.
- **Memory Map review:** Regularly review the memory map to catch large symbols (using tools like `nm` or `cargo-bloat`) and optimize or feature-gate them as needed.

1.2 Peak Heap Usage Tracking (if using `alloc`)

- **What to Measure:** If heap allocations are enabled (e.g. via `alloc` crate or allocator support), track the **peak heap memory** used and allocation patterns over time. This reveals if the actor system stays within memory budgets and helps catch leaks or fragmentation on embedded.
- **Tools & Methods:**
- **Custom Global Allocator:** Implement a lightweight allocator wrapper to instrument allocations. For example, create a global allocator that counts current allocated bytes and records the **high-water mark** (peak) at runtime [1]. This is analogous to Rust's `dhat` heap profiler but for `no_std` (since `dhat` itself requires `std`) [1]. Each `alloc` and `dealloc` can update counters (careful to use atomic or interrupt-safe operations).
- **Heap Measuring in Tests:** In an integration test (on device or in QEMU), allocate typical actor workloads (spawn actors, send messages) and then query the allocator's counters for peak usage. For example, after running a message flood test, log the peak heap usage via a debug interface (ITM, semihosting, or UART logging with `defmt`).
- **Minimal Overhead:** The allocator instrumentation should be as simple as incrementing/decrementing counters – this adds negligible overhead and can be left enabled in test builds. Since no external profiler can be attached easily on bare-metal, this self-instrumentation is crucial.
- **Best Practices:**
- **Wrap `alloc` carefully:** If using a third-party allocator (like `heapless` or `buddy_alloc`), consider forking or wrapping it to expose internal stats (e.g. remaining free memory, largest free block) to help detect fragmentation.
- **Zero allocation mode:** Also test with `alloc` *disabled* to verify the system can run without heap (heap usage profiling tools should be gated behind a feature to avoid breaking no-alloc builds).

1.3 Interrupt Latency & Task Execution Latency

- **What to Measure:** The time from an **interrupt firing or an event occurrence to the corresponding actor/task actually running**. This includes ISR entry latency and any scheduling delay before the actor handles the event. Also measure **task execution latency** (the time an actor takes to process a message, possibly including scheduling jitter if multiple actors). For real-time systems, it's critical to know worst-case latency.
- **Tools & Methods:**
- **Hardware Cycle Counters:** Leverage on-chip counters for precise timing. ARM Cortex-M MCUs provide the DWT (Data Watchpoint and Trace unit) cycle counter, which can be enabled to count CPU cycles with ~0 overhead [3]. By reading the `DWT->CYCCNT` register at two points (e.g. at interrupt handler start and actor event handling start), you can compute the latency in cycles. Similarly, on RISC-V use the `mcycle` CSR (via inline assembly or the `riscv` crate) to timestamp events.
- **Instrument ISRs and Actor Loops:** For example, in a Cortex-M system, toggle a GPIO **HIGH** at the start of an ISR (or when posting a message to an actor) and toggle it **LOW** when the actor's `on_event` for that interrupt's message begins. Measure the pulse width with a logic analyzer or oscilloscope – this directly gives interrupt-to-task latency in real time (with nanosecond accuracy, effectively zero intrusion).

- **ITM Tracing:** Use the ARM ITM via the `itm` crate to emit timestamped events on the SWO pin [4]. For instance, emit an ITM event in the interrupt and another in the actor handler; tools like **RTIC-Scope** can capture these and calculate latency distributions. **Orbuculum**, an open-source trace toolkit, can be used on the host PC to collect and visualize ITM events, giving insight into timing behavior on Cortex-M [5].
- **Software Timing (fallback):** If hardware counters aren't available, use a high-resolution timer peripheral or even the SysTick (for coarse μs timing) to timestamp events. Keep in mind software timers add more overhead and lower precision.
- **Metrics:** Collect min/avg/max latency and ideally jitter (e.g. standard deviation). For critical real-time actors (e.g. an ISR-driven sensor actor), ensure the max latency stays within requirements (perhaps double-check by generating worst-case load).
- **Best Practices:**
- **Prevent Optimizer Removal:** When using cycle counters in code, perform a *volatile read* of the counter to ensure it's not optimized away [6].
- **One-at-a-time tests:** To measure pure ISR latency, test with minimal other load. Then separately measure under load (many actors active) to see scheduling impact.
- **Cooperative Scheduling Tuning:** Since `lit-bit` uses a cooperative scheduler (Embassy for embedded [7] [8]), consider the *yield frequency*. If an actor monopolizes CPU, it could delay others. Ensuring that each message handler is short or yields periodically (similar to Erlang's reduction count concept) will keep latencies low – this can be validated by the timing tests.

1.4 Power Consumption Implications

- **What to Measure:** How the actor system influences MCU power draw – e.g. CPU active vs sleep ratios, and if possible, estimate energy per message or the impact of different scheduling strategies on power. This is **optional ("if feasible")**, as direct power measurement is typically external to the MCU logic.
- **Tools & Methods:**
- **External Power Measurement:** The most accurate approach is using a power analyzer or a specialized tool (e.g. Nordic Power Profiler Kit or Monsoon) connected to the device's power supply. One can run a scenario (e.g. 1000 messages processed) and measure the total energy used. While not an automated "in-code" profiling tool, this provides ground truth power usage.
- **On-Board ADC Shunt Measurement:** If the hardware has a current-sense resistor and ADC, you can sample the voltage drop to estimate current draw. For instance, some dev boards let you measure Vcc via ADC – by toggling actors on/off, you might infer power differences. This requires calibration and is not very precise, but can show trends (within ~5-10% accuracy).
- **CPU Sleep Instrumentation:** Track how often the system is idle vs busy. On Cortex-M, you can use the DWT to count cycles spent in active execution vs WFI (Wait-For-Interrupt) low-power state. Alternatively, toggle an LED or pin when entering idle sleep and measure duty cycle on that pin as a proxy for CPU active percentage.
- **Profiling Tools:** Some RTOS or schedulers have built-in energy profiling; if `lit-bit` is running atop an RTOS, check if it provides hooks for tracking power states. Otherwise, rely on external measurement as above.
- **Best Practices:**
- **Benchmark Different Strategies:** For example, compare always-on busy polling vs. WFI sleep between messages. If `lit-bit` on embedded uses Embassy, it likely utilizes interrupts and sleeps when idle. Validate that by measuring near-zero CPU usage when actors are idle (which correlates to lower power).

- **Optimize for Sleep:** Ensure that the actor system doesn't inadvertently wake up too often. Profiling the power can catch issues like a misconfigured timer causing needless wakes.
- **Record Temperature/Voltage:** External factors affect power; ensure tests run under consistent conditions or at least note them.

Summary – Embedded Profiling: Use **hardware-assisted profiling** wherever possible to minimize overhead: cycle counters for timing (measuring in CPU cycles or microseconds with virtually no overhead) [3], ITM/SWO for trace logging of events [9], and custom allocators for memory tracking. Static analyses (size, call-stack) complement runtime measurements to give a full picture of memory safety and efficiency. The goal is to quantify that `lit-bit` on embedded meets memory constraints and real-time deadlines with minimal footprint.

Table: Key Embedded Profiling Tools & Techniques

Aspect	Tools/Methods (Embedded)	Overhead
Static Memory Usage	• `cargo size` / `objdump` for .text/.data/.bss sizes (flash/ RAM) <br> • `cargo-call-stack` for static **stack** analysis [1] <br> • Code audit for hidden `std` uses (CI "heap-safety" checks)	*Offline (build-time) – no runtime cost*
Heap Usage (alloc)	• **Custom Global Allocator** that tracks allocated bytes and peak [1] <br> • Expose allocator stats via logging or ITM <br> • Alternatively, use a bump allocator and measure high-water mark of `brk` pointer	*Low (couple of counters updated on alloc/free)*
Latency (ISR & Task)	• **Cycle counter timestamps** (ARM DWT CYCCNT, RISC-V mcycle) around critical sections [3] <br> • **GPIO toggling** at event boundaries (oscilloscope measurement) <br> • **ITM trace events** on ISR entry/exit and actor start [4]	*Cycle counter read = 0-1µs; ITM event ~ minimal (hardware)*
Power Usage	• **External power monitor** (e.g. measure mA with scope or specialized tool) <br> • **Idle pin toggle** to measure CPU active % <br> • Use MCU's low-power modes and verify via current draw	*External measurement (no code overhead)*; pin toggle adds tiny overhead

2. Host (`std`) Profiling Goals

On the host side (e.g. Linux or Windows, running the actor system with `std` and an async runtime), the focus shifts to **dynamic behavior**: heap allocation patterns, memory fragmentation, CPU cache efficiency, and multi-thread scheduling overhead. The environment allows use of powerful profiling tools (e.g. Valgrind, `perf`) but we must interpret results in the context of asynchronous actor workloads.

2.1 Allocation/Deallocation Patterns (Heap Profiling)

- **What to Measure:** How often and where does the system allocate memory? What is the allocation size distribution? Are there excessive short-lived allocations (which could cause fragmentation or GC-like behavior)? By profiling heap usage, we can identify hot spots (e.g. message creation, mailbox resizing) and memory leaks.

- **Tools:**
- **Valgrind – DHAT and Massif:** Valgrind's **DHAT (Dynamic Heap Analysis Tool)** records detailed heap allocation history and usage (object lifetimes, allocation sites) without modifying the program [10]. You can run the actor test under Valgrind/DHAT to see which code paths allocate the most and how memory usage evolves over time. **Massif** (another Valgrind tool) focuses on heap size over time (tracking the heap high-water mark and identifying when/where peak usage occurs).
- `dhat-rs` **Crate:** An alternative is the Rust `dhat` **crate**, which provides heap profiling in-process (inspired by Valgrind's DHAT) [11]. By integrating `dhat-rs` in a dev build, you can capture allocation traces or stats and then view them with the DHAT viewer. This avoids the heavy slow-down of full Valgrind, at the cost of embedding some profiling code. *Example:* wrap a benchmark in a `dhat::Profiler` guard – it will log all allocations and their sizes while active [12].
- **Heap Trackers:** Tools like **Heaptrack** or **Google's perftools heap profiler** can also be used. Heaptrack records all allocations and frees, providing a timeline and showing which functions are allocating. This is useful for visualizing if memory use climbs (fragmentation or leak) or stays stable.
- **Allocator Statistics:** If using a specific allocator (e.g. `jemalloc` via `tikv-jemallocator` or `mimalloc`), we can query its stats. For instance, jemalloc has `mallctl` calls to get total allocated memory, active memory, and fragmentation metrics (like ratio of active/allocated). Embedding such a call at the end of a benchmark can report memory fragmentation: e.g. if "allocated (reserved) = 1 MB" but "active (in-use) = 500 KB", we have 50% fragmentation at that point.
- **Metrics:** Focus on **total heap usage** at steady state vs peak, number of allocations per message processed, and any pattern of growth (leaks). Also track **max resident set size (RSS)** via OS (e.g. using `/usr/bin/time -v` or OS APIs) to see total memory footprint.
- **Interpretation:**
- If profiling shows frequent tiny allocations in the messaging hot path, consider using object pooling or arena allocation for messages to reduce allocator overhead.
- If peak memory scales with number of actors or messages linearly, ensure it matches expected per-actor overhead (we can compare to Erlang's ~300 words per process as a baseline [13] – Rust actors might use more, but should be on the order of kilobytes, not megabytes each).
- A **flat allocation profile** (no single spot dominating) is good, whereas a spike at a particular function (e.g. constructing a big `Vec` each time) indicates an optimization target. DHAT's output will highlight hotspots in allocation counts or bytes.
- **Best Practices:**
- Run heap profiling in **release mode** with typical workloads (optimizations on) to get realistic patterns, since debug mode may allocate very differently.
- **Automate regression checking:** Consider adding a CI job or a periodic test that runs a scenario and checks that total memory usage stays under X bytes or doesn't grow unbounded. This catches memory leaks early.
- Pair allocation analysis with **free analysis**: ensure that for every mailbox enqueue, the memory is freed when the message is processed (no accumulating buffer). Tools like Valgrind **Memcheck** can be run as well to catch any actual leaks or invalid frees.

2.2 Memory Fragmentation Tracking

- **What to Measure:** Fragmentation occurs when heap memory becomes "holes" due to many alloc/free of varying sizes, causing the process to hold more memory than it actually needs for live data.

We want to track if long-running actor systems suffer fragmentation (which can lead to memory bloat over time).

- **Tools & Methods:**
- **Fragmentation Metrics via Allocator:** As mentioned, `jemalloc` can report stats such as “allocated vs active vs resident”. *Active* memory is in-use by application, *Allocated* (or *Resident*) is total reserved from OS. A big gap implies fragmentation or internal fragmentation. After running a lengthy test (lots of actor churn and message traffic), query these stats.
- **Simulate Worst-case Fragmentation:** Use a deterministic workload that is known to fragment memory in other systems (e.g. allocate a range of message sizes, then free in a pattern). Observe with Heaptrack or DHAT if memory usage returns to baseline or if holes prevent reuse.
- **Valgrind Massif:** Massif can show the heap usage over time and after frees. If the heap size doesn’t drop after a large release of actors/messages, that could hint at fragmentation. Massif’s output also includes fragmentation estimates.
- **Best Practices:**
- **Use consistent allocation sizes for frequent ops:** If each actor’s mailbox always allocates the same size buffer (e.g. channel of N messages) or uses static buffers (`heapless::Vec` for embedded), fragmentation is minimized. This design principle can be validated by observing stable memory usage in profiling.
- **Periodic Compaction (if needed):** In extreme cases, one might consider triggering a “fake” allocation to force the allocator to merge free blocks (though Rust’s global allocators generally don’t support manual compaction). Instead, ensuring that large allocations (e.g. big inboxes) are reused rather than constantly created/destroyed is preferable.
- **Watch long-term stability:** If `lit-bit` runs for hours, does memory steadily climb? Use soak tests with memory profiling to ensure it plateaus.

2.3 CPU Cache Behavior and Core Profiling

- **What to Measure:** How efficiently does the actor system use the CPU and caches? Key metrics include **CPU cycles per message, cache miss rates, branch mispredictions,** and distribution of work across CPU cores. Cache behavior is important because actors do lots of message passing (accessing memory queues) which can lead to cache misses if not optimized. Core profiling refers to understanding how threads (e.g. async runtime worker threads) utilize CPU cores – are they balanced or contending?
- **Tools:**
- **Linux `perf` (Performance Counters):** The `perf` tool is invaluable for this. It can collect hardware performance counters with minimal overhead (since it leverages CPU’s built-in counters) [14] . For instance, run the actor benchmark and collect counters like:
    - `cycles`, `instructions` – to compute CPI (cycles per instruction) and see if the workload is CPU-bound.
    - `cache-references`, `cache-misses` – to measure cache miss rate. A high miss rate might indicate poor locality (perhaps messages bouncing between cores or large data structures).
    - `branches`, `branch-misses` – to gauge branch predictor efficiency (important if there are a lot of conditionals in message dispatch; though ideally in `lit-bit` the hot path is straightforward).
    - `context-switches` and `cpu-migrations` – to see how often threads are switching or moving between cores (excessive context switches can hurt performance).

Example command: `perf stat -e cache-misses,cache-references,context-switches -a ./lit-bit-benchmark` to get summary statistics. Perf's output will show totals and percentages.

- `perf record` **and Flamegraphs:** Use `perf record -g` to sample the program and then `perf flamegraph` (via `inferno` or `FlameGraph` scripts) to get a visualization of where CPU time is spent. This helps identify if the **actor execution vs runtime overhead** is dominating. For example, one might find that 30% of CPU is in `tokio::runtime::park` (idle waiting) or lock contention, versus actual user code. Such flamegraphs have shown, for instance, ~60% of time in Tokio runtime for certain workloads [15] . A flamegraph can reveal if cache misses cluster around certain functions (they appear flat if a function's self time is large possibly due to memory stalls).
- **Core Distribution:** Tools like `htop` or `top` (with thread view) can show CPU usage per thread in real time. Alternatively, `perf sched record/report` can produce a timeline of when each thread was running. This helps verify that, say, on a 4-core system, the work is spread evenly across runtime worker threads, or if one core is a bottleneck.
- **Cache Simulation (advanced):** Valgrind's **Cachegrind** can simulate L1/L2 cache and give detailed info on cache hits/misses per line of code. It's very slow (like 20-50x slowdown) so it's more of an offline analysis tool, but it can pinpoint poor locality in the code. Use it on smaller test cases if needed (e.g. one actor ping-pong message to see if those accesses hit in cache).
- **Metrics & Interpretation:**
- **Cache Miss Rate:** If L1 miss rate is high (e.g. >5-10%), consider investigating data structures. For instance, if mailboxes use a Vec/heap that causes non-contiguous access, maybe a ring buffer (heapless queue) is better (which `lit-bit` uses for no_std, and Tokio's mpsc for std which is cache-optimized).
- **Instructions per Cycle (IPC):** Compare IPC to the machine's max. A low IPC (much lower than 1 on a superscalar CPU) often indicates the code is memory-bound (waiting on memory). If IPC is high (~1 or 2), it's likely CPU-bound. Use this to decide whether to optimize algorithm (CPU bound) or data locality (memory bound).
- **Branch misses:** If branch mispredictions are significant, look at branching in actor message dispatch. However, Rust async/await state machines compile down to largely linear code; branch misses might come from unpredictable workloads. If, for example, one branch is taken 50% of time and not easily predicted, maybe reordering logic or using jump tables could help (though likely not a big factor here).

- **Thread/Core usage:** Ideally, in a high-throughput test, all cores are near 100% utilization (no single thread saturating while others idle). If one thread (core) is 100% and others 20%, you have a **load imbalance** – possibly a bottleneck in a single actor or a single lock. The actor system should distribute actors across threads. Tokio does work-stealing to balance tasks; profiling can verify if it's effective. If not, consider pinning certain actors to threads to avoid contention, or splitting heavy work.

- **Best Practices:**

- Use **release mode and CPU affinity** for consistent results. Pin the process to a CPU set if doing fine-grained cache measurements to avoid OS moving it around.
- **Warm up** the system before measuring: CPU cache behavior can vary in the first few iterations. In benchmarks, discard initial runs so caches, branch predictors, etc., are "warmed".

- Compare runs with **hyper-threading on vs off** if on x86, since HT can affect cache (shared L1/L2) and context switch metrics.
- If certain cache metrics are problematic, consider micro-optimizations: e.g., align frequently accessed data to cache lines, avoid false sharing between actor threads by padding structures (particularly if `Address` or mailboxes have atomic counters that different threads touch – ensure each such atomic is on its own cache line to avoid "ping-pong" between cores).

2.4 Thread Contention and Scheduling Overhead

- **What to Measure:** The cost of synchronization and context switching in the host environment. In an async actor system, sources of contention include: locking in the executor (e.g. task queue mutexes), competition for CPU between threads, and any messaging channels that use locks or atomic operations. **Scheduling overhead** refers to time spent managing tasks (pushing them in/out of queues, waking threads) instead of doing useful work. We want to quantify how much overhead the runtime introduces and if thread contention (e.g. multiple threads trying to send to the same actor mailbox) causes slowdowns.
- **Tools & Methods:**
- **CPU Profiling (runtime overhead):** As mentioned, flamegraphs or profiles can show how much time is in the runtime. In one example, ~60% of time was in `tokio::runtime` for a heavy task workload [15], meaning only 40% was actual work – we want to minimize this overhead if possible. If our flamegraph shows a large portion of time in scheduling (e.g. in `poll` functions of the executor, or in thread parking/unparking), that's the overhead. Tokio's instrumentation (with `tokio-metrics` or `console-subscriber`) can also give insight: e.g. number of tasks awoken, how long tasks wait in queue, etc.
- **Contention Analysis:** Use `perf c2c` (cache-to-cache transfer analysis) or simpler, use perf counters for atomic instructions and cache line bouncing. If a particular atomic (e.g. a reference count in Arc or a mutex) is heavily contended, it will show up as increased LLC misses or `cache-misses` when multiple cores modify it. Another trick: temporarily instrument a suspected lock with `parking_lot::deadlock_detection` or measure lock wait times by timestamping before/after lock acquisition in critical sections (though Tokio's internals aren't easily modifiable, you can wrap user-level locks if any).
- **Thread Sanitizer (TSan):** While primarily for data races, TSan can sometimes highlight heavy lock contention as it tracks mutex operations. It's more for debugging, but if your test triggers TSan warnings of high contention, that's a sign.
- **Scheduler Timeline:** Use `perf sched` to record a timeline of thread scheduling events. You can see if threads frequently context-switch (voluntarily or involuntarily). High context switch rate (hundreds or thousands per second) can indicate tasks bouncing around. Ideally, each worker thread should run tasks in batches rather than constant context switches.
- **Metrics:** Context switch count (via `perf stat` or `/proc/<pid>/status`), average run time per task, run queue lengths (OS-provided or via runtime metrics). If using Tokio, the **Tokio Console** (or console subscriber crate) can live-monitor tasks: how long they are sleeping, running, etc., which helps see if tasks are frequently yielding due to waiting on resources.
- **Interpretation:**
- If we find a lot of time in synchronization, consider using lock-free structures or per-thread resources. For example, if a single mailbox is contended by many senders across threads, perhaps sharding or an atomic queue might reduce lock contention (Tokio's mpsc is already pretty optimized with lock-free segments).

- **Thread Pool Sizing:** Check if having more threads than CPU cores is causing excessive context switches (threads preempting each other). It might be better to limit the runtime to equal or slightly above core count to reduce overhead.
- **Task Scheduling Overhead:** If an actor processes extremely fast messages (<1μs work each) but the runtime overhead to schedule that task is, say, 5μs, then the overhead dominates. In such cases, batching messages or processing multiple messages per wake-up can improve throughput. We can emulate what Erlang does: handle a certain number of messages in one actor invocation before yielding – this amortizes scheduling cost. The `lit-bit` actor loop can be tuned to process multiple inbox items in succession (if not already), which is something to verify in profiling (look at message latency vs throughput tradeoff).
- **Best Practices:**
- **Use lightweight synchronization:** wherever possible, use atomic operations instead of Mutex, and if a Mutex is needed (e.g. for a global log or stats), ensure it's not on the hot path of message handling.
- **Measure at various loads:** Contention might only show up at high concurrency. Profile with different numbers of actor threads and message volumes to find breaking points. For example, measure throughput scaling from 1 thread to N threads – if it plateaus or drops after a certain N, that's likely contention or overhead increasing.
- **Affinity & NUMA considerations:** On multi-socket systems, threads on different NUMA nodes accessing shared data incur extra latency. If relevant, keep actor threads on one NUMA node or use per-node actors to reduce cross-node traffic (perf can show NUMA locality events too).

Table: Key Host-Side Profiling Tools & Focus

Profiling Area	Tools & Commands (Host)	Insights Gained
Heap allocations	• **Valgrind DHAT/Massif** – `valgrind --tool=dhat ./actor_bench` <br> • **dhat-rs crate** – integrate `dhat::Profiler` in code [12] <br> • **Heaptrack** – GUI analysis of alloc/free by function	Find high-frequency allocation sites, peak memory usage, leaks. Identify if message passing creates too many allocs.
Memory footprint	• **/proc/PID/status** or OS tools for RSS <br> • Custom allocator stats (jemalloc) for in-use vs reserved memory	Check per-actor memory overhead, fragmentation (reserved - used). Ensure memory usage scales linearly (or better) with actors/ messages.
CPU hotspots	• **perf stat** – CPU counters (cycles, instructions, cache misses) [14] <br> • **perf record + Flamegraph** – visualize CPU usage across functions	Determine if bottleneck is CPU or memory. See how much time spent in `lit-bit` logic vs runtime overhead. Identify cache inefficiencies (high miss rate).

Profiling Area	Tools & Commands (Host)	Insights Gained
Cache behavior	• **perf stat -e cache-misses,LLC-misses,branch-misses** <br> • **Cachegrind** (Valgrind) for detailed per-line misses	Quantify cache miss % – if high, indicates poor locality in actor message access. Branch misses hint at unpredictable branches.
Thread scheduling	• **perf stat -e context-switches,cpu-migrations** <br> • **perf sched record/report** for timeline <br> • Tokio Console / metrics for task stats	Measure how often threads switch (if very high, overhead). See if tasks frequently migrate between cores (cpu-migrations). Identify any idle cores vs overworked cores.
Lock contention	• **ThreadSanitizer** (build with `-Zsanitizer=thread`) – will flag lock issues <br> • **Custom counters** – e.g. count how many times a mailbox queue is full (contention indicator) or how long a lock is held (via timestamps)	Reveal if locks (e.g. mailbox Mutex or atomic send) are contended. If tasks back up waiting for locks, throughput will suffer. Use this to consider lock-free alternatives or restructuring.

3. Async Runtime Choices: Trade-offs for Benchmarking

`lit-bit` is designed to run on multiple async runtimes (Tokio for std, Embassy or custom for no_std) [7]. For benchmarking, choosing the right executor/runtime is critical to get **accurate, unbiased measurements**. The goal is to avoid introducing noise or unfair advantages from a particular runtime. Key considerations:

- **Tokio (multi-threaded) vs others:** Tokio is the most mature and high-performance Rust async runtime, but its multi-thread scheduler can introduce non-deterministic variations and significant overhead for micro-tasks. Meanwhile, simpler executors (like **futures-lite** with **smol** or a custom single-thread executor) have lower overhead but may not reflect real-world multi-core performance. We must balance realism vs determinism in benchmarks.

Below is a comparison of runtime options and their impact on benchmarking:

Runtime Option	Characteristics for Benchmarking	Use Cases
Tokio (multi-thread)	High throughput on multi-core, uses work-stealing. However, each task spawn/awakening involves atomics and potential cross-thread handoff, adding overhead. Results can be noisy due to thread scheduling by OS. *Not nestable:* you cannot easily start a Tokio runtime inside another (would panic "Cannot start a runtime from within a runtime"). Need to run benches in a fresh runtime thread [16].	**Realistic throughput** on multi-core; measuring system at scale. Use for stress tests (many actors), but account for ~microsecond-level overhead per task wake. Avoid for pinpoint latency measurement (noise from thread scheduling).
Tokio (current-thread)	Single-threaded Tokio runtime (or using `LocalSet` for !Send tasks). Removes multi-thread overhead but retains Tokio's task scheduling logic. More deterministic since one thread runs all tasks. Still has some scheduling cost (waking tasks involves an atomic flag, but no thread handoff). Crucially, easier to embed in tests because it doesn't spawn background threads.	**Deterministic micro-benchmarks**; measuring exact processing latency without thread preemption. Good for comparing algorithmic overhead of actor handling vs sync code. Also simulates an embedded-like environment on host.
async-std	An alternative runtime with auto-spawned thread pool. Simpler API (spawn is global). Historically had higher overhead (in one user's flamegraph, ~79% in runtime vs 60% in Tokio for similar work [15]). Fewer knobs to control scheduling. No known nesting issues (but it starts its own threads implicitly).	Mostly for cross-checking Tokio. If we want to ensure `lit-bit` works on async-std or measure its performance relative to Tokio, we can include it. Generally, Tokio outperforms async-std in many cases, so it may not be the default choice for benchmarking.

Runtime Option	Characteristics for Benchmarking	Use Cases
futures-lite + smol	A minimal executor: can be run single-threaded or with a thread pool (smol). Very lightweight – tasks are polled in a simple loop, minimal synchronization when single-threaded. Lacks the complex scheduling of Tokio, so overhead per task is lower. However, it doesn't have Tokio's IO or timer features (not needed for pure computation). No nested runtime issues – you can simply call `block_on` or drive the executor directly.	**Apples-to-apples sync vs async:** Great for comparing an async actor vs a synchronous algorithm because the scheduling overhead is minimal. Use it to find the **pure** cost of `async/await` state machines and message passing without Tokio's optimizations. Also useful on embedded (smaller footprint executors).
Embassy (embedded)	Embassy provides a static event loop for no_std – single-threaded, interrupt-driven. In the host context, we can simulate it (as done in `PerformanceTestKit.spawn_actor_embassy()` [17]) by running the Embassy executor on a dedicated thread. Embassy has near-zero overhead when idle and cooperative scheduling like RTIC. No thread preemption at all.	**Embedded equivalence:** Use in benchmarks to mirror how the actor system would behave on embedded. For example, compare `spawn_actor_tokio` vs `spawn_actor_embassy` costs [18] [17] to ensure the design scales in both environments. Also helps catch any divergence in behavior between the two.

Avoiding Tokio Nesting Issues: To run Tokio-based benchmarks while the test harness (like Criterion) is itself an async context, the common solution is to **create a new Runtime** manually and use its handle to execute the benchmarked code. For example, the code snippet below (from our TestKit) shows using `tokio::runtime::Runtime::new()` and then running the async code via `rt.block_on` or Criterion's `to_async` adaptor [16] [19] :

```rust
let rt = tokio::runtime::Runtime::new().unwrap();
// Use Criterion's to_async to run on the runtime
b.to_async(&rt).iter(|| async {
    // ... benchmark body ...
    let mut testkit = PerformanceTestKit::new();
    let actor = ThroughputTestActor::new();
    let addr = testkit.spawn_actor_tokio(actor, size);
    // send messages, await replies...
});
```

This ensures the Tokio runtime used for benchmarking is separate and not already running on the current thread (preventing the "runtime inside runtime" error). **Bottom line:** always initialize a fresh runtime in each benchmark thread, or use single-thread Tokio in sync tests, to avoid nesting conflicts.

Choosing Runtimes for Benchmarks:

- For **throughput tests** (lots of messages, high concurrency), use **Tokio multi-thread** to see realistic performance on a multicore system. However, also run the same test on a single-thread executor to compute the scheduling overhead. For example, if single-thread processes X messages/sec and multi-thread processes 4X on a 4-core, we're scaling well; if only 2X, contention is hurting scalability.
- For **latency tests**, prefer a single-thread context (Tokio current-thread or futures-lite) to measure the baseline **message round-trip latency** without cross-thread noise. This gives the best-case numbers. You can then compare with multi-thread to see how much jitter is introduced by scheduling.
- For **CPU profiling**, a single-thread run is easier to analyze (no interleaving of events), but a multi-thread run is needed to profile contention. Use both modes and compare flamegraphs.
- For **embedded simulation**, incorporate Embassy or a `no_std` **executor** in the host benchmarks. The `PerformanceTestKit` already allows spawning on Embassy vs Tokio for direct comparisons [20] [17]. This apples-to-apples approach (same actor code, different executors) reveals the overhead of the runtime itself.

Trade-off Summary: Simpler executors (futures-lite, single-thread Tokio) give more consistent and minimal-overhead measurements, isolating the true cost of actor operations. Rich executors (Tokio multi-thread) include real-world effects like context switching and load balancing. We recommend **using both types**: start with a deterministic executor to baseline the performance, then verify on Tokio to ensure the behavior holds under real concurrent conditions. Keep the benchmarking harness flexible – e.g., feature-flag which runtime to use – so it's easy to switch and run on either without altering the test logic. This ensures the **"runtime-agnostic"** goal is met.

4. Insights from Akka & Erlang/OTP Benchmarking Methodologies

The actor model is well-established in Erlang/OTP and Akka (JVM/Scala). We can draw on their performance methodologies to guide our benchmarks in `lit-bit`. Key metrics these ecosystems emphasize include **throughput (messages/sec)**, **latency (message round-trip times)**, and **memory footprint per actor**, as well as patterns for testing these.

4.1 Throughput Benchmarking (Messages per Second)

Akka & Erlang Approaches: Both communities routinely measure how many messages an actor system can process per second. A classic throughput test is a **ping-pong benchmark** – two actors send a message back and forth as fast as possible, often for millions of messages, and measure the total time. For example, benchmarks have shown Akka achieving on the order of 1–2 million msg/s on a single machine, vs Erlang around 1 million msg/s on similar hardware [21]. A more recent comparison from the Proto.Actor framework (C# and Go actor library) showed:

- **Akka.NET:** ~350k msg/s (remote) and ~46 million msg/s (in-process)
- **Erlang:** ~200k msg/s (remote) and ~12 million msg/s (in-process) [22]
  ("Remote" is across network, "in-process" is within one runtime with no networking overhead.)

These numbers illustrate how throughput can vary by platform and scenario. Erlang's lower in-process count is partly due to per-process GC and perhaps stricter fairness, whereas Akka on the JVM benefitted from JIT optimizations in tight loops.

Methodology to Adapt: For `lit-bit`, define **throughput scenarios** similar to those in Akka/Erlang:

- **In-process Ping-Pong:** Two actors on the same runtime send a message back and forth N times. Measure total time and compute messages/sec. This gauges the raw speed of message handling (should be in hundreds of thousands or millions per second on a modern CPU if efficient). Our Criterion benchmarks already do something akin to sending 1000 messages and timing it [23]. We should scale that up (100k or 1M messages) to get stable throughput figures and possibly use bigger numbers for release builds (ensuring Criterion's overhead is negligible).
- **Multi-actor Throughput:** Create many actors (e.g. 100 actors in a ring or a star topology) and flood messages among them to see system-wide throughput. Akka's **Savina** benchmark suite (for actors) includes patterns like pipeline, throughput under various topologies, etc. We can borrow those patterns to ensure we measure not just a simple ping-pong but also scenarios with contention (many senders to one receiver, which tests mailbox backpressure).
- **Remote/Distributed:** If applicable, measure throughput across a network or thread boundary. Akka and Proto.Actor do "remote ping-pong" (two actors on separate nodes/machines). For us, "remote" could be akin to two separate executor threads communicating (though in-process, but maybe via an async channel to simulate remote). This tests the overhead of serialization or message copying. In embedded context, "remote" might be interrupt to thread communication, which we partially cover in latency.

Metrics & Goals: We want `lit-bit` to achieve **on the order of >100k messages/sec per core**, as hinted in the docs [24] (targeting >100k msg/sec/core). Our benchmarks should confirm this. For instance, if one core can do 200k msg/sec (which is 5 μs per message on average), that's good. If it's much lower, profile why – maybe the overhead per message is too high.

Also measure **scaling**: if 1 core = 200k msg/s, ideally 4 cores = ~800k msg/s (some loss is normal due to coordination overhead). If scaling is poor, look at thread contention (back to section 3/host profiling).

4.2 Latency Benchmarking (Message Round-trip Time)

Akka & Erlang Approaches: Latency is often measured as the round-trip time (RTT) of a message exchange, sometimes under various load conditions. Erlang, for instance, might use `timer:tc` to measure microsecond-level latencies of a ping-pong. They often collect **percentile latencies** – e.g., median, 95th, 99th percentile – to understand the distribution, not just the mean. The **Savina** suite includes latency-sensitive benchmarks (e.g., a one-hop round trip vs pipeline). Erlang's approach to latency is also to test under low load vs high load, because under load latency might increase (due to message queueing).

Methodology to Adapt: We have a **LatencyMeter** in our test utilities that already computes p50, p95, p99, etc. [25]. We should use it to gather latency stats for single message round-trips:

- **Isolated latency:** Measure the RTT of one message send and reply, when no other messages are in the system. This can be extremely low (sub-microsecond in optimized C, perhaps a few microseconds in Rust with async). Our goal might be, say, median latency < 50 μs, p99 < 500 μs on typical hardware

[26]. In fact, the docs mention a target of **<200 ns message latency** (which is very ambitious – likely referring to baseline overhead) [24]. Realistically, on a 3GHz CPU, 200 ns is ~600 cycles; we should verify if we approach that for an actor forwarding a message in-process. This might require using the single-thread executor to eliminate scheduling overhead.

- **Loaded latency:** Follow Akka's practice of not measuring latency in isolation only, but also when the system is processing other traffic. E.g., run a throughput test and simultaneously probe the latency of a specific message. This setup reveals queueing delays. Akka devs sometimes use a harness where one pair of actors does ping-pong while N other actors flood the system, then observe how the ping-pong latency changes. We can script a benchmark where one "latency probe" actor pair exchanges a message periodically while a bulk workload runs, and record the latencies (this requires careful synchronization to log times).
- **Timeouts and Service times:** Also measure any *actor service time* (the time spent in `on_event` handling a message) if relevant, because if an actor does heavier computation, that affects latency to the next message. Erlang uses *reductions count* (roughly function calls) to limit how long a process runs before yielding. In Rust, an actor could starve others if it doesn't yield. Our latency tests can help verify that no single actor can cause unbounded delays to others (especially in the single-thread case). If we see long-tail latencies, it might indicate a need for explicit yielding or breaking work into smaller messages.

Metrics: Use **LatencyMeter's output** to get mean, p95, p99, max. Ensure p99 is within acceptable range (e.g., if mean is 50µs but p99 is 5ms, something sporadically stalls – investigate GC pauses, OS scheduling, etc.). Our tests should assert latency targets (the example in our test code asserts p95 < 500µs, p99 < 1ms [26]). Those numbers can be adjusted based on hardware, but having a target is useful (just as Akka might claim e.g. "~1 ms p99 latency under load").

4.3 Memory Footprint per Actor

Akka & Erlang Approaches: Memory overhead per actor (or process) is a critical metric, as it determines how many actors can be spawned. Erlang is renowned for lightweight processes (~300 words ≈ 1.2 KB each initially) [13]. Akka actors run on the JVM, so each actor is just an object (a few dozen bytes) but they often share threads from a pool, so thread stack overhead is amortized. Akka's bigger cost can be mailbox queues and any user state.

The ecosystems often test by **creating a very large number of actors** (tens or hundreds of thousands, even millions) and measuring total memory. They also measure how memory scales with number of messages in mailboxes (to see if backpressure works or if memory usage explodes when mailboxes fill up).

Methodology to Adapt: We should include **benchmarks for actor creation and memory use**:

- **Mass Actor Spawn Test:** Spawn, say, 100k lightweight actors (with minimal message handling logic) and measure memory usage. This can be done by observing the process RSS before and after, or by summing up heap allocations through a profiling tool. If 100k actors use X MB, then per-actor overhead ~ X/100k. We can compare this to known baselines (if 100k Erlang processes ~ 120 MB, that's ~1.2 KB each, etc.). Our goal is likely to keep it in the low KB range per actor. The spawn benchmark already in our suite (`bench_actor_spawn_cost`) times the act of spawning on Tokio vs Embassy [27] [17], giving performance of spawning. But we should also record memory. We might enhance that test to record how much heap is used after spawning N actors.

- **Mailbox Memory Test:** Create one actor with a large mailbox (e.g. send it 10k messages without it processing immediately) and measure memory. This tests the overhead per enqueued message. If using Tokio's bounded MPSC, the channel may allocate buffers in chunks. We ensure no major leaks after the queue is drained. This is analogous to backpressure tests – e.g. ensure that an actor with a full mailbox doesn't crash memory (it should either block senders or drop messages as configured).
- **Supervision overhead:** If actors restart on failure (OTP-style), measure if that process accumulates any memory (it shouldn't; a restart should free/reallocate the actor's state cleanly). Perhaps simulate a rapid crash-restart loop to see if memory grows (which would indicate a resource not freed). This is more of a correctness test but has memory implications.

Metrics: Memory per actor (in steady state with empty mailbox), memory per message (overhead of messaging subsystem), total footprint of a typical actor system (e.g. 1 supervisor + 10 workers + some messages in queues). Also measure **spawn time** (our bench covers that, aiming for minimal spawn latency). Akka's JMH results (in one example, spawning a million actors had certain time cost – Rust should generally beat that significantly given no VM startup needed).

4.4 Tooling/Infrastructure from Akka/Erlang

Borrowed Tools/Techniques:

- **Test Harnesses:** Akka uses JMH (Java Microbenchmark Harness) for precise benchmarks (to avoid JVM warm-up effects). In Rust, **Criterion** plays a similar role (statistical robust measurements). We are already using Criterion in our `performance_tests` module [28]. We should follow JMH best practices: do warm-ups, multiple iterations, and use statistical summary (Criterion does this). This reduces variance and gives confidence in comparisons.
- **Savina Benchmark Suite:** This is a collection of actor benchmarks originally for academic comparisons (includes patterns like ping-pong, throughput, pipeline, concurrent search, etc.). It was referenced for Actix (Rust actor framework) as something to possibly port [29]. We can use Savina's scenarios as inspiration to ensure we're not missing a pattern. For example, Savina's **"Skynet"** benchmark (creating a million actors in a tree and summing results) is a good test of both spawn overhead and scheduling (Proto.Actor's site lists a variant of it, where Proto.Actor did it in ~0.5s vs Akka.NET 4.5s [22]). We could implement a Skynet in `lit-bit` and measure it; this touches recursion, messaging, and a huge number of tiny actors – a great stress test for memory and scheduler.
- **Actor System Metrics:** Erlang has built-in functions to get metrics like `erlang:memory()` (total memory breakdown) and `process_info(Pid, memory)` (memory used by a specific process) [30]. In our Rust context, we may expose similar metrics via the system: e.g., a global that tracks total messages sent, alive actor count, etc., only for testing. While not normally in a production library, having a feature to query such info can assist in benchmarks (for instance, after a test, assert that `actor_count == expected` and maybe memory usage is within bounds). This is analogous to how Erlang can introspect number of processes, message queue lengths, etc., to verify test outcomes.
- **Failure Injection:** Erlang/OTP has a strong focus on failure recovery (the "Let it crash" philosophy). A reliability benchmark mentioned in research involves injecting failures and measuring how throughput drops or recovers [31]. For performance, we might not need to go deep into that, but we could measure the overhead of supervision (e.g., does enabling supervision checks slow things?). Probably negligible, but worth noting if any locks around supervisor trees exist.

- **Profilers and Monitors:** Erlang has `observer` (GUI for monitoring processes, message queues in real time) and Akka has Cinnamon/Telemeter, etc., for monitoring actor systems. In Rust, using `tracing` with instrumentation is analogous – e.g., instrument actors to emit events when they start/stop or when mailboxes hit capacity. For benchmarking, we don't want the overhead of heavy tracing, but lightweight logging (like counting metrics) can be turned on to analyze a run after the fact. For instance, count how many context switches or yields happened in a million messages – if that count is >> million, maybe too many yields. This kind of telemetry can be inspired by what Akka devs look at (they often look at mailbox sizes, throughput per actor, etc., via their monitoring tools).

Infrastructure to Consider: Setting up a **continuous benchmarking pipeline** (as alluded in our docs [32]) similar to how Akka has nightly performance runs. This could use something like GitHub Actions to run Criterion and collect results, maybe graphing them over time (to catch regressions). Akka teams often have performance regression tests alongside correctness tests. We aim to do the same: include performance tests (throughput, latency, memory) as part of the test suite (possibly behind a feature flag so it doesn't run every `cargo test` by default). This ensures `lit-bit` meets its performance goals consistently.

4.5 Summary of Recommendations (Akka/Erlang Learnings)

- **Benchmark Patterns:** Implement **ping-pong** (for latency and single-pair throughput), **flooding** (many-to-one throughput), **broadcast** (one-to-many), and **Skynet-like actor creation tests**. These cover the common cases used to evaluate actor systems in academia and industry, allowing comparison with published results.
- **Throughput & Latency Metrics:** Always report both throughput (ops/sec) and latency (in a distribution). Akka/ Erlang emphasize that you cannot optimize just one – e.g., maximizing throughput can sometimes hurt tail latency [33]. So present both. Perhaps produce a table after running benchmarks: e.g., "Throughput: X msg/s, p99 latency: Y µs at Z msg/s load". This is something done in e.g. Akka's documentation or blog posts where they discuss performance.
- **Memory & Scalability:** Borrow the Erlang style of measuring how memory scales with number of actors. E.g., "each actor costs ~Y bytes", "system can spawn 1e6 actors in A seconds using B MB memory". This paints a clear picture of scalability. We have the tools (via profiling and OS introspection) to get these numbers.
- **Minimal Overhead in Measurement:** Both Akka and Erlang communities try to minimize observer effect in benchmarks: e.g., running on dedicated machines, disabling GC for short tests if possible, etc. In Rust, ensure that the performance tests are run with full optimizations and without debug hooks (except those explicitly for profiling). When measuring, do not enable `tracing` or debug logs (they can drastically skew results). In embedded, turn off debug asserts. Essentially, mirror their practice of "measure release performance in a production-like config".
- **Leverage OTP Concepts:** Erlang's isolation and per-process GC means memory and performance costs are distributed. In Rust, we don't have a GC pause, which is a plus, but we should test the "isolation" – one actor's heavy load shouldn't drop overall throughput too much. This can be benchmarked by having one actor do CPU-intensive work while others process messages; see if those others still get CPU time (on multi-core, they should – on single-thread, they won't unless the busy actor yields). Akka's fork-join pool tries to balance such scenarios; we may rely on Rust's thread pool for similar effect.
- **Reporting & Visualization:** Consider outputting results in a table or chart form. For example, an **A/B comparison table** of different runtimes or settings:

Configuration	Throughput (msg/s)	p99 Latency (µs)	Notes
Tokio (4 threads)	X (e.g. 1,000,000)	Y (e.g. 1000µs)	baseline multi-core
Tokio (1 thread)	X1	Y1	no thread switching
futures-lite (1 thread)	X2	Y2	minimal executor overhead
Embassy (simulated)	X3	Y3	embedded style

And a **Memory table**:

Scenario	Memory Used (MB)	Actors or Msg count	Approx per Actor (bytes)
100k actors, idle	M1 MB	100,000	(M1*1024*1024 / 100000) bytes each
100k actors with 1 msg	M2 MB	100,000	overhead per queued message = M2-M1
1 actor, 10k msgs enqueued	M3 MB	10,000 messages	M3/10000 per message in queue

This mirrors how one might present results in a report or README, inspired by how others present (Proto.Actor's table [22], Erlang docs giving exact memory words [13], etc.).

In conclusion, by combining **low-level embedded profiling** (cycle counts, custom allocators) with **robust host profiling** (perf, Valgrind), and by structuring benchmarks to reflect **real use cases and known actor-model patterns**, we can create a comprehensive picture of `lit-bit`'s performance. The strategy ensures that both tracks (embedded and host) meet their objectives: memory safety and real-time behavior on one hand, and high throughput with efficient resource usage on the other. Adopting proven practices from Akka/Erlang while leveraging Rust's unique advantages (zero-cost abstraction, no GC) will position `lit-bit` as a truly high-performance actor system across all platforms.

Sources:

- Embedded profiling tools and tips [34] [35] [3]
- Host profiling with perf and DHAT [14] [12]
- Tokio vs async runtime overhead observations [15]
- Proto.Actor benchmark comparisons (Akka vs Erlang) [22]
- Erlang process memory footprint documentation [13]
- `lit-bit` internal test guide (performance checklist) [36] and benchmarks [19] [20]
- Latency target assertions in tests [26] and actor performance goals [24].

---

[1] [4] [5] [9] [34] [35] Debugging and profiling embedded applications. : r/rust
https://www.reddit.com/r/rust/comments/126vgrx/debugging_and_profiling_embedded_applications/

[2] [7] [8] 2025-05-25.md
https://github.com/0xjcf/lit-bit/blob/6bda3c500be055cd10b938e63703fc93d35bbd0f/docs/PROGRESS/2025-05-25.md

[3] [6] Cycle Counting on ARM Cortex-M with DWT | MCU on Eclipse
https://mcuoneclipse.com/2017/01/30/cycle-counting-on-arm-cortex-m-with-dwt/

[10] Best memory profiler? - help - Rust Users Forum
https://users.rust-lang.org/t/best-memory-profiler/75571

[11] dhat - crates.io: Rust Package Registry
https://crates.io/crates/dhat/0.3.0

[12] [14] Profiling with perf and DHAT on Rust code in Linux • Ryan James Spencer
https://www.justanotherdot.com/posts/profiling-with-perf-and-dhat-on-rust-code-in-linux

[13] Erlang -- Processes
https://www.erlang.org/docs/22/efficiency_guide/processes

[15] Tokio is faster than async-std, but can I reduce overhead? - help - The Rust Programming Language Forum
https://users.rust-lang.org/t/tokio-is-faster-than-async-std-but-can-i-reduce-overhead/58967

[16] [17] [18] [19] [20] [23] [25] [26] [27] [28] [32] [36] test-guide.md
https://github.com/0xjcf/lit-bit/blob/6bda3c500be055cd10b938e63703fc93d35bbd0f/docs/test-guide.md

[21] Let it crash • Benchmark: Akka vs Erlang
https://letitcrash.com/post/14783691760/akka-vs-erlang

[22] Benchmarks | Proto.Actor
https://proto.actor/docs/performance/benchmarks/

[24] actor-overview.md
https://github.com/0xjcf/lit-bit/blob/6bda3c500be055cd10b938e63703fc93d35bbd0f/docs/actor-overview.md

[29] scala - Akka scalability and performance benchmark testcases - Stack Overflow
https://stackoverflow.com/questions/15833855/akka-scalability-and-performance-benchmark-testcases

[30] How to calculate the size of an Erlang process in memory?
https://stackoverflow.com/questions/7528199/how-to-calculate-the-size-of-an-erlang-process-in-memory

[31] [PDF] A Reliability Benchmark for Actor-Based Server Languages
https://www.dcs.gla.ac.uk/~trinder/papers/ReliableActorBenchmark.pdf

[33] Akka grpc performance in benchmarks
https://discuss.lightbend.com/t/akka-grpc-performance-in-benchmarks/8236