Gazebo Performance Profiling Report

Carlos Agüero

caguero@honurobotics.com

May 2026

Contents

1 Introduction **3**
- 1.1 Goals . . . 3
- 1.2 Approach . . . 3

2 Methodology and Reproduction Procedure **3**
- 2.1 Profiling Techniques . . . 3
- 2.2 Benchmark World Selection . . . 5
- 2.3 Hardware . . . 5
- 2.4 Software Versions . . . 5
- 2.5 Setup . . . 5
- 2.6 Benchmark Worlds . . . 6
- 2.7 Running Captures . . . 6
- 2.8 Publishing a Run . . . 7
- 2.9 Online Repository . . . 7

3 Loading-Time Analysis **8**
- 3.1 Loading Flamegraphs . . . 8
- 3.2 Key Loading-Time Findings . . . 9

4 Runtime Analysis Per World **10**
- 4.1 Runtime Flamegraphs . . . 11
- 4.2 3k_shapes_static (3000 static entities, headless, RTF=0) . . . 11
- 4.3 3k_shapes_dynamic (3000 dynamic entities, headless, RTF=0) . . . 13
- 4.4 sensors_nonrendering (IMU, mag, etc., headless, RTF=0) . . . 13
- 4.5 jetty_headless (complex world, headless, RTF=0) . . . 14
- 4.6 gpu_lidar (GPU lidar, headless-rendering, RTF=0) . . . 15
- 4.7 sensors_demo (6 rendering sensors, headless-rendering, RTF=0) . . . 15

5 Cross-Reference Analysis **16**
- 5.1 Gazebo-Owned Hotspots (optimization targets) . . . 16
- 5.2 External Library Costs (context, not directly actionable) . . . 17

6 CPU Cache Analysis **17**
- 6.1 Loading cache performance . . . 17
- 6.2 Runtime cache performance . . . 17
- 6.3 Per-function cache analysis . . . 18

7 Extended Analysis **18**
- 7.1 Off-CPU Analysis . . . 18
- 7.2 False Sharing Detection . . . 19
- 7.3 Scheduler Latency Analysis . . . 20
- 7.4 Per-Thread Flamegraphs . . . 20
- 7.5 Differential Flamegraphs . . . 21
- 7.6 Unified Summary . . . 21

8 Optimization Recommendations **22**
- 8.1 Priority 1: Replace EachNoCache in ProcessRecreateEntitiesRemove . . . 22
- 8.2 Priority 2: Eliminate per-step pose map rebuild in ChangedWorldPoses . . . 23
- 8.3 Priority 3: Cache texture/image decoding in jetty-class worlds . . . 23
- 8.4 Priority 4: Reduce broadphase cost for worlds with many static bodies . . . 24
- 8.5 Priority 5: Optimize BitmaskContactFilter and updateEngineData . . . 24
- 8.6 Priority 6: Reduce loading time for large entity counts . . . 25
- 8.7 Priority 7: Investigate ECS View and thread barrier overhead . . . 25

9 Validating Optimizations **25**

9.1 Validation tools 25
9.2 Validation workflow 26
9.3 Existing GZ_PROFILE scopes for each priority 26
9.4 Recommended benchmark worlds per priority 26

10 References **27**

1 Introduction

Gazebo is a widely used robotics simulator that integrates physics engines (DART, Bullet), a 3D rendering pipeline (Ogre), sensor simulation, and an Entity Component System (ECS) into a multi-threaded architecture coordinated by a main simulation loop. As simulation worlds grow in complexity — more models, more sensors, more detailed meshes — users frequently encounter real-time factor (RTF) degradation without clear visibility into what is consuming CPU time.

Prior profiling efforts in Gazebo have relied on instrumentation profiling via the built-in Remotery profiler (`GZ_PROFILE` macros). Instrumentation profiling measures only explicitly annotated code scopes, providing deterministic per-step timing for Gazebo's own subsystems. However, it cannot see inside third-party libraries (DART, Ogre) or system-level costs (memory allocation, thread synchronization, dynamic linking). This leaves significant blind spots when diagnosing performance problems.

This report introduces sampling profiling (CPU flamegraphs via Linux `perf`) as a complementary technique. Sampling profiling periodically captures the full call stack across all libraries — including DART's constraint solver, DART's ODE-based collision broadphase, Ogre's scene-graph traversal, and glibc's memory allocator — without requiring code annotations. By combining sampling profiling (for discovery) with instrumentation profiling (for validation), we can attribute CPU time to specific subsystems and identify optimization targets with high confidence.

1.1 Goals

- Understand where CPU time goes across representative simulation workloads, with emphasis on Gazebo-owned code rather than third-party physics engines
- Measure startup/loading time and identify bottlenecks in world initialization
- Identify concrete, actionable optimization opportunities within the Gazebo codebase
- Establish a reproducible profiling methodology and benchmark suite for ongoing performance work

1.2 Approach

We selected 6 benchmark worlds covering distinct stress axes: entity count scaling (3000 shapes, static and dynamic variants), GPU lidar rendering, multi-sensor rendering, non-rendering sensors, and a realistic mixed-use scenario (jetty demo). Worlds were run at maximum speed (RTF=0) to saturate CPU and eliminate real-time pacing overhead. Both steady-state runtime and startup/loading captures were collected. A mix of static and dynamic worlds was used to separate Gazebo framework overhead from physics engine cost — the static variant of 3k_shapes (all models `<static>true</static>`) isolates ECS, transport, and scene management overhead by removing all DART/ODE work.

The study follows four phases. The **discovery phase** uses sampling profiling (CPU flamegraphs via `perf`) to identify where time goes across the full stack and prioritize optimization targets. **CPU cache analysis** uses hardware performance counters (`perf stat`) to measure cache miss rates and Instructions Per Cycle (IPC), revealing whether hotspots are compute-bound or memory-bound. The **extended analysis phase** applies false sharing detection (`perf c2c`), scheduler latency measurement, and per-thread flamegraphs to characterize multi-threaded behavior; results are combined into a composite score via `gz_summary.sh`. The **validation phase** (Section 9) defines a methodology for measuring the before/after impact of each optimization using instrumentation profiling (`GZ_PROFILE` / Remotery) and differential flamegraphs.

---

2 Methodology and Reproduction Procedure

2.1 Profiling Techniques

This study uses two complementary profiling approaches to achieve full-stack visibility into Gazebo's CPU usage: sampling profiling and instrumentation profiling.

Sampling profiling (via Linux `perf`) periodically interrupts the running process using hardware performance counters at a fixed frequency (997 Hz in this study) and records the current call stack. These stack

samples are statistical — they approximate where time is spent without modifying the target code. The samples are collapsed and rendered as interactive SVG flamegraphs using Brendan Gregg's FlameGraph scripts [1]. In a flamegraph, the x-axis represents the proportion of total CPU samples (wider = more time), and the y-axis represents call stack depth (bottom = entry point, top = leaf function where time is actually spent). The resulting SVG is interactive: clicking any frame re-renders the chart with that frame as the new baseline, effectively zooming into that subtree. Flamegraphs capture *every* function in the call stack, including third-party libraries (DART, Ogre), system libraries (glibc malloc, pthreads), and kernel calls — making them ideal for identifying bottlenecks that span library boundaries.

Two types of flamegraph captures were performed:

- **Loading captures**: `perf` wraps the entire process launch (`perf record -- gz-sim-main --iterations 1 <world>`), capturing startup, SDF parsing, physics engine initialization, and scene construction. This reveals where loading time goes.
- **Runtime captures**: `perf` attaches to a running Gazebo process and samples for 30 seconds during steady-state simulation. This reveals where CPU goes during normal operation.

To produce readable flamegraphs, the code must be compiled with debug symbols (`-g` via `RelWithDebInfo`) so that `perf` can map instruction addresses to function names, and with frame pointers (`-fno-omit-frame-pointer`) so that `perf` can walk the call stack reliably. Without frame pointers, stack traces break and the flamegraph contains `[unknown]` frames. An alternative is `perf record --call-graph dwarf`, which uses DWARF debug info for stack unwinding but produces 3-5x larger capture files.

The sampling frequency of 997 Hz (rather than 1000 Hz) avoids lock-step aliasing with system timers that may fire at 1 kHz. The Linux `perf_event_paranoid` sysctl must be set to 1 or lower to allow user-mode CPU sampling.

Instrumentation profiling (via `GZ_PROFILE` / Remotery) uses explicit code annotations to measure exact timing. Developers insert `GZ_PROFILE("scope_name")` macros that record entry/exit timestamps and send them to a Remotery server. A browser-based visualizer connects via WebSocket and displays a real-time timeline of named scopes per thread. Unlike sampling profiling, instrumentation profiling measures only *explicitly annotated* scopes — it cannot see inside libraries that lack annotations. However, it provides deterministic, per-step timing data that is easy to interpret and correlate with specific Gazebo subsystems.

The `GZ_PROFILE` macros are compiled as no-ops unless `ENABLE_PROFILER=ON` is passed at CMake time, ensuring zero overhead in production builds. Four Gazebo libraries support this flag: gz-sim, gz-physics, gz-rendering, and gz-sensors.

For the discovery phase (sampling profiling), `ENABLE_PROFILER=OFF` is recommended. This eliminates the ~1.5% overhead from `_rmt_BeginCPUSample` calls and produces cleaner flamegraphs without Remotery frames cluttering the stacks.

For the validation phase (instrumentation profiling), `ENABLE_PROFILER=ON` should be used. This activates the Remotery scopes that provide deterministic per-step timing for each named code block. See Section 9 for the recommended validation workflow.

CPU Cache Analysis uses `perf stat` with hardware performance counters to measure cache miss rates and Instructions Per Cycle (IPC). While flamegraphs show *where* time is spent, cache analysis reveals *why* — a function with high inclusive time and low IPC indicates a memory-bound bottleneck where the CPU is stalled waiting for data from RAM, not executing instructions. This distinguishes algorithmic inefficiency (doing too much work) from data structure inefficiency (doing work that thrashes the cache), and informs whether the fix should be "do less" or "reorganize memory layout."

Extended Analysis Techniques. In addition to the core CPU flamegraph and cache analysis, the profiling framework includes several supplementary techniques that provide deeper insight into multi-threaded simulation performance:

- **Off-CPU flamegraphs** capture where threads *block* (mutexes, I/O, condition variable waits) using BCC's `offcputime-bpfcc` tool (eBPF-based, in-kernel aggregation). CPU flamegraphs only show where threads burn cycles; off-CPU flamegraphs show where they *wait*. Together they account for 100% of thread time, following Brendan Gregg's principle that "off-CPU analysis is complementary to

CPU analysis" [2].
- **False sharing detection** uses `perf c2c` to measure HITM (Hit-In-Modified) events — cross-thread cacheline contention where one thread writes to a cache line that another thread is reading. High HITM rates on a symbol indicate that threads sharing a data structure are thrashing each other's caches even if they access different fields.
- **Scheduler latency analysis** uses `/proc/PID/task/TID/schedstat` to measure per-thread scheduling delays: the time threads spend in the run queue waiting for a CPU core. Delays above 1ms can cause missed physics steps in real-time simulation.
- **Per-thread flamegraphs** split a single capture into one flamegraph per thread using `stackcollapse-perf.pl -tid`. This separates CPU-hot threads (physics, rendering) from I/O-idle threads (transport, polling) without requiring function name heuristics — the sample count per thread is the data-driven signal.
- **Differential flamegraphs** compare before/after captures using `difffolded.pl` to validate optimizations. Red frames indicate regression (more time than baseline), blue frames indicate improvement.

All techniques produce machine-readable TSV output that feeds into a unified summary script (`gz_summary.sh`), which combines evidence from all available dimensions into a single ranked priority list with composite scores (0–100 points).

2.2 Benchmark World Selection

Six benchmark worlds were selected to cover distinct stress axes. All worlds were modified to run at maximum speed (`<real_time_factor>0</real_time_factor>`) to saturate CPU and eliminate the real-time pacing overhead from the flamegraphs. A mix of static and dynamic worlds separates Gazebo framework overhead from physics engine cost:

- **Static worlds** (3k_shapes_static, sensors, jetty paused): isolate ECS iteration, transport publishing, and rendering pipeline cost by removing all DART/ODE physics work.
- **Dynamic worlds** (3k_shapes, jetty headless): show the full picture including physics. Comparing dynamic 3k_shapes with its static variant reveals the pure physics engine cost.
- **Rendering/sensor worlds** (gpu_lidar, sensors_demo): exercise the Ogre rendering pipeline and sensor simulation. These require `gz topic -e` subscribers to force the sensor system to actually render and publish data — without subscribers, the pipeline skips rendering work entirely.

2.3 Hardware

- **CPU**: Intel Core Ultra 9 285HX, 24 cores (no HT), x86_64
- **GPU**: NVIDIA RTX PRO 3000 Blackwell Generation Laptop GPU
- **NVIDIA Driver**: 580.126.09 (used for headless EGL rendering via Ogre2)
- **Kernel**: 6.17.0-20-generic
- **GCC**: 13.3.0

2.4 Software Versions

- **gz-sim**: 11.0.0-pre1 (workspace source build, main branch)
- **gz-physics**: source build (main branch)
- **DART**: system-packaged libdart 6.13.2 (no debug symbols)
- **perf**: 6.17.13
- **FlameGraph**: Brendan Gregg's scripts (https://github.com/brendangregg/FlameGraph)

2.5 Setup

```
# 1. Clone the profiling repository
git clone https://github.com/caguero/gz-profiling
cd gz-profiling

# 2. Clone FlameGraph scripts
git clone https://github.com/brendangregg/FlameGraph
```

```
export FLAMEGRAPH_DIR=$(pwd)/FlameGraph

# 3. Build your Gazebo workspace with debug symbols
cd <GZ_WS>
colcon build --merge-install \
  --cmake-args -DENABLE_PROFILER=OFF \
               -DCMAKE_BUILD_TYPE=RelWithDebInfo \
               -DCMAKE_CXX_FLAGS="-fno-omit-frame-pointer" \
               -DCMAKE_C_FLAGS="-fno-omit-frame-pointer"

# 4. Source workspace and set environment
source <GZ_WS>/install/setup.bash
export GZ_CONFIG_PATH=<GZ_WS>/install/share/gz:$GZ_CONFIG_PATH
export GZ_SIM_MAIN=<GZ_WS>/install/libexec/gz/sim/gz-sim-main
sudo sysctl kernel.perf_event_paranoid=1
```

Replace `<GZ_WS>` with the path to your Gazebo workspace (e.g., `~/rotary_ws`).

2.6 Benchmark Worlds

All benchmark worlds are in the repository's `worlds/` directory with `real_time_factor` set to `0` to saturate CPU. `3k_shapes_static.sdf` additionally has `static` set to `true` on all 3004 models. Sensor worlds have companion `.topics` files listing topics for subscriber setup.

2.7 Running Captures

From the `gz-profiling` repository directory:

```
# Capture all worlds (loading + runtime flamegraphs)
./scripts/capture_all.sh worlds/

# Single world loading capture
./scripts/gz_loading_flamegraph.sh worlds/jetty.sdf jetty

# Single world runtime capture
./scripts/gz_flamegraph.sh worlds/jetty.sdf jetty 30 headless

# Cache-miss flamegraph
./scripts/gz_cache_flamegraph.sh --runtime worlds/3k_shapes_static.sdf 3k_static

# Analyze hotspots from .folded file
./scripts/gz_hotspots.sh captures/runtime/3k_shapes_static.folded

# CPU vs cache-miss comparison
./scripts/gz_compare_cpu_cache.sh captures/runtime/3k_shapes_static.folded \
  captures/cache/3k_static_rt_cachemiss.folded

# Quick IPC / cache miss rate
./scripts/gz_cache_stats.sh --pid $PID 10

# Extended analysis (false sharing, scheduler, per-thread, summary)
./scripts/gz_false_sharing.sh --pid $PID jetty 5
./scripts/gz_sched_analysis.sh --pid $PID jetty 5
./scripts/gz_per_thread_flamegraph.sh captures/runtime/perf_jetty.data jetty
./scripts/gz_summary.sh captures/ jetty

# Full capture with all extended analyses
```

```
./scripts/capture_all.sh worlds/ --full

# Full pipeline: capture + analyze + cross-world ranking (one command)
./scripts/gz_full_pipeline.sh worlds/

# Post-capture analysis only (no simulation needed)
./scripts/gz_analyze.sh captures/

# Differential flamegraph (after optimization)
./scripts/gz_diff_flamegraph.sh baseline.folded optimized.folded my_fix
```

`gz_full_pipeline.sh` runs the complete profiling workflow end-to-end: launches each world, captures all dimensions (CPU, loading, cache-miss, c2c, sched), then runs `gz_analyze.sh` which performs hotspot analysis, per-thread splitting, cache comparison, per-world scoring, and cross-world merging into a single ranked list of optimization targets (`captures/summary/cross_world_summary.tsv`). Use `gz_analyze.sh` alone to re-run analysis on existing captures without re-capturing.

2.8 Publishing a Run

The profiling workflow has two stages: **capture locally** (iterate until satisfied), then **publish** (package results into a dated directory for the repository).

```
# 1. Full pipeline: capture + analyze + cross-world ranking
./scripts/gz_full_pipeline.sh worlds/

# 2. Publish when happy
./scripts/gz_publish_run.sh captures/

# 3. Commit and push
git add YYYY-MM-DD/ && git commit -m "Add profiling run" && git push
```

`gz_publish_run.sh` creates a dated directory with all SVG, folded, and TSV files (skipping large `perf_*.data` files that stay local), runs post-capture analysis if not already done, generates a per-run `index.html`, and updates the top-level index.

2.9 Online Repository

All interactive flamegraphs, capture scripts, benchmark worlds, and the PDF report are published at:

https://github.com/caguero/gz-profiling

Interactive flamegraphs are served via GitHub Pages at:

https://caguero.github.io/gz-profiling/

The repository contains everything needed to reproduce the study:

```
gz-profiling/
  scripts/                    # Reusable capture and analysis
    gz_flamegraph.sh          #   Runtime flamegraph capture
    gz_loading_flamegraph.sh  #   Loading/startup capture
    capture_all.sh            #   Captures all worlds in a directory
    gz_hotspots.sh            #   Gazebo hotspot analysis from .folded
    gz_cache_stats.sh         #   CPU cache miss/IPC measurement
    gz_cache_flamegraph.sh    #   Cache-miss flamegraph capture
    gz_compare_cpu_cache.sh   #   CPU vs cache-miss comparison
    gz_offcpu_flamegraph.sh   #   Off-CPU (blocking) flamegraph
    gz_false_sharing.sh       #   False sharing detection (perf c2c)
    gz_sched_analysis.sh      #   Scheduler latency analysis
    gz_per_thread_flamegraph.sh # Per-thread flamegraph split
    gz_diff_flamegraph.sh     #   Differential flamegraph
```

```
    gz_summary.sh           #   Unified multi-dimensional summary
    gz_analyze.sh           #   Post-capture analysis + cross-world merge
    gz_full_pipeline.sh     #   End-to-end: capture + analyze + rank
    gz_publish_run.sh       #   Package captures for the repo
  worlds/                   # Benchmark world SDFs (all RTF=0)
    3k_shapes.sdf           #   3000 dynamic entities
    3k_shapes_static.sdf    #   3000 static entities
    sensors.sdf             #   Non-rendering sensors
    jetty.sdf               #   Complex real-world scene
    gpu_lidar_sensor.sdf    #   GPU lidar rendering
    gpu_lidar_sensor.topics #   Sensor topics for subscriber
    sensors_demo.sdf        #   6 rendering sensors
    sensors_demo.topics     #   Sensor topics for subscriber
  2026-04-21/               # This benchmark run
    runtime/                #   Flamegraph SVGs + .folded files
    loading/                #   Loading flamegraph SVGs + .folded
    findings_report.pdf     #   This report
    findings_report.tex     #   LaTeX source
    figures/                #   Thumbnails for the report
    cache/                  #   Cache-miss flamegraphs + .folded
    index.html              #   Per-run index with links
  YYYY-MM-DD/               # Future runs
```

Flamegraph thumbnails in this report link to the corresponding interactive SVG on GitHub Pages. Click any thumbnail to open the full flamegraph in a browser with click-to-zoom, hover tooltips, and search functionality. The `?s=<regex>` URL parameter pre-highlights matching frames in magenta.

---

3 Loading-Time Analysis

World	Wall-clock	Samples	Key observation
3k_shapes_static	**26.0s**	25,962	Dominated by entity creation
3k_shapes_dynamic	**26.6s**	26,545	Nearly identical to static — physics init is cheap
jetty	**7.8s**	7,718	SdfModelSerializer::Serialize + plugin loading
sensors_nonrendering	2.4s	368	Fast — few entities
gpu_lidar	0.06s	5	Trivial — few entities, rendering initializes at runtime
sensors_demo	0.03s	5	Trivial — few entities, rendering initializes at runtime

3.1 Loading Flamegraphs

Click any thumbnail to open the full interactive flamegraph in a browser (supports click-to-zoom, hover tooltips, and Ctrl+F search).

World	Wall-clock	Flamegraph
3k_shapes_static	26.0s	

3k_shapes_dynamic	26.6s	
jetty	7.8s	
sensors_nonrendering	2.4s	*Fast — few entities, no significant loading cost*
gpu_lidar	0.06s	*Loads instantly — few entities*
sensors_demo	0.03s	*Loads instantly — few entities*

3.2 Key Loading-Time Findings

3k_shapes (26s)

Loading flamegraph shows `SceneBroadcasterPrivate:: SceneGraphAddEntities` as a top self-time leaf (3% of loading in static variant). The loading time is dominated by constructing 3000 ECS entities and DART skeletons. Static vs dynamic loading time is nearly identical (26.0 vs 26.6s), confirming that DART skeleton construction cost is per-entity, not per-dynamic-entity.

`SceneGraphAddEntities`
(gz-sim, SceneBroadcaster.cc) Constructs a directed acyclic graph tracking parent-child relationships for new entities and serializes their pose and component data into protobuf messages for transport publication. Scales linearly with entity count.

`DownloadAssets`
(gz-sim, ServerPrivate.cc) Despite no remote assets in any benchmark world, this function consumes a significant fraction of every world's loading time. The SDF file is parsed **twice** during startup by design:

World	% of loading	Estimated time
sensors_nonrendering	29.9%	0.7s
jetty	13.9%	1.0s
3k_shapes_static	9.1%	2.3s
3k_shapes_dynamic	8.3%	2.2s

1. **First parse** (Server.cc:93): `LoadSdfRootHelper` parses the full SDF quickly (1.3%, barely visible in the flamegraph). The server then **intentionally strips all models, actors, and lights** from the result (Server.cc:104–108, `ClearModels/ClearActors/ClearLights`) and creates simulation runners with an empty world shell. This allows the GUI to start immediately without waiting for asset downloads.
2. **Second parse** (ServerPrivate.cc:787, inside `DownloadAssets`): `LoadSdfRootHelper` is called **again** on a background thread, this time fully resolving all model references via `Root::Load → readFile → init → initDoc → initXml` (deeply recursive). After parsing, the complete world (now including any freshly downloaded Fuel models) is injected back into the runner via `SetWorldSdf` (line 840), and `SetCreateEntities` (line 844) triggers entity creation. This second parse is visible in the flamegraph of **every world** — not just 3k_shapes.

The architecture is intentional: parse once for fast GUI startup, parse again in the background to get the full models. However, for worlds without Fuel URIs (all benchmark worlds tested), the first parse already has all the models — they are discarded and then re-parsed for no reason, costing 0.7–2.3 seconds per startup. The optimization: when no Fuel URIs are queued, skip the

second parse and keep the models from the first parse instead of discarding them. Additionally, `FetchQueuedAssets` creates a local `WorkerPool(2)`, spawning 2 threads for an empty download queue — also skippable.

jetty (7.8s)

The bottleneck is `SdfModelSerializer::Serialize`, called from `SceneBroadcaster::PostUpdate` → `EntityComponentManager::AddEntityToMessage`. This serializer converts every model's SDF representation back to a string via recursive `Element::ToString` → `PrintValuesImpl` calls. The recursive depth (3-5 levels of `ToString`, 2-5 levels of `PrintValuesImpl`) amplifies the cost for complex models.

`SdfModelSerializer::Serialize`
: (gz-sim, EntityComponentManager.cc) Called only once — on the first simulation step when all entities are new. The call path is: `SceneBroadcaster::PostUpdate` checks `HasNewEntities()`, which triggers `ChangedState()` → iterates over all `newlyCreatedEntities` → calls `AddEntityToMessage()` for each → invokes `SdfModelSerializer::Serialize` which recursively converts the SDF to string via `Element::ToString` → `PrintValuesImpl`. On subsequent steps, `HasNewEntities()` is false so this code path is never reached again. The cost is a one-time per-loading spike, but for complex models with deeply nested SDF elements, it dominates the first step.

`do_lookup_x`
: (glibc dynamic linker, 4.8%) Resolves symbol references when shared library plugins are loaded. Each plugin triggers symbol relocation across all loaded libraries. Appears in top-10 for every world (2-9%).

`DownloadAssets`
: (gz-sim, ServerPrivate.cc, 13.9% in jetty) Same redundant second SDF parse as described under 3k_shapes above. In jetty, this costs ~1.0s of the 7.8s loading time — the SDF is re-parsed on a background thread despite no Fuel URIs being present.

Common observations

`do_lookup_x` (dynamic linker) appears in top-10 for every loading capture (2-9%).

`pybind11::initialize_interpreter`
: (gz-sim, Server.cc) The Python interpreter is initialized **unconditionally** at server construction (`Server::Server` → `pybind11::initialize_interpreter` → `Py_InitializeFromConfig`), even when no Python system plugins are referenced in the world file. None of the benchmark worlds use Python plugins, yet every loading capture pays a fixed ~200–260ms cost for interpreter initialization and module imports:

World	% of loading	Estimated time
sensors_nonrendering	7.5%	180ms
jetty	3.3%	257ms
3k_shapes_static	1.0%	260ms
3k_shapes_dynamic	0.9%	239ms

The fix: lazy-initialize the Python interpreter only when a `python_system_loader` plugin is actually requested by the world.

---

4 Runtime Analysis Per World

All runtime captures were recorded for 30 seconds at 997 Hz with the real-time factor target set to 0 (run as fast as possible). The measured RTF column shows the actual achieved performance.

World	Measured RTF	Samples	Key observation
3k_shapes_static	0.04	557K	Hashtable 26.8%, ODE broadphase 3.8% despite static
3k_shapes_dynamic	0.01	611K	EachNoCache 10.8% + unresolved DART 18.3%
sensors_nonrendering	1.94	603K	Thread sync overhead 4% (Barrier::Wait, pthread)
jetty_headless	0.11	578K	Collision 21.9% + JPEG texture decode 6%
gpu_lidar	1.44	868K	NVIDIA EGL driver 6.9%, Ogre2 culling 3%
sensors_demo	1.34	691K	Ogre2 scene-graph traversal 22%

4.1 Runtime Flamegraphs

Click any thumbnail to open the full interactive flamegraph in a browser (supports click-to-zoom, hover tooltips, and Ctrl+F search).

World	Stress Axis	Flamegraph
3k_shapes_static	3000 static entities	
3k_shapes_dynamic	3000 dynamic entities	
sensors_nonrendering	IMU, mag, etc.	
jetty_headless	Complex real-world scene	
gpu_lidar	GPU lidar rendering	
sensors_demo	6 rendering sensors	

4.2 3k_shapes_static (3000 static entities, headless, RTF=0)

Purpose: Isolate Gazebo framework overhead with zero physics work.

Bold rows are Gazebo-owned (actionable). Non-bold rows are external libraries (context).

Leaf function	%	Category
`std::_Hashtable` (Write)	**26.8**	**gz-physics: pose map rebuild**
`Graph::Vertices` (EachNoCache)	**13.5**	**gz-sim: full entity scan**

Leaf function	%	Category
std::_Rb_tree_increment	7.7	ordered map/set iteration
WorldForwardStep	**4.3**	**gz-physics: per-link loop**
Frame::getWorldTransform	4.4	DART kinematic chain traversal
dxHashSpace::collide	3.8	ODE broadphase on static geometry
DegreeOfFreedom::getPosition	3.6	DART joint accessor
Skeleton::getDof	2.5	DART skeleton accessor
CollisionGroup::updateSkeleton*	2.5	DART collision prep

Key finding: The dominant cost is std::_Hashtable operations at 26.8%. A DWARF-unwound recapture reveals the caller: SimulationFeatures::Write(ChangedWorldPoses) (gz-physics, SimulationFeatures.cc:176–214). Every simulation step, this function:

1. Creates a **fresh** unordered_map<size_t, Pose3d> newPoses (line 183)
2. Iterates all 3000 links, inserting each pose into newPoses[id] — even when the pose hasn't changed (line 206)
3. Replaces the previous map: prevLinkPoses = std::move(newPoses) (line 213)

With 3000 static entities where nothing ever moves, every step still allocates a new 3000-entry map, inserts 3000 entries (triggering multiple _M_rehash calls as the map grows), then destroys the old map. This allocate-populate-destroy cycle is the source of the 26.8%.

Fix: Pre-size newPoses with reserve(this->links.size()) to eliminate rehashing. Better yet, reuse prevLinkPoses in place — update entries that changed, remove entries for deleted links — instead of rebuilding the map from scratch every step.

A second major cost is Graph::Vertices() at 13.5%, called from ProcessRecreateEntitiesRemove (SimulationRunner.cc:1510). This function uses EachNoCache<Model, Recreate> which iterates over **all 3000 entities** via Entities().Vertices(), checking each one for Recreate components. In a static world with nothing to recreate, it scans 3000 vertices every step to find zero matches. The ECS header itself notes: *"The cached version, Each(), is preferred."* — but this call site uses the uncached variant.

Fix: Switch to the cached Each<Model, Recreate>(), or add an early-return when no Recreate components exist (e.g., check HasComponentType<Recreate> before iterating).

ODE broadphase collision also runs every step at 3.8% despite all models being static. DART accessors (getWorldTransform 4.4%, getPosition 3.6%) are called per-entity per-step.

WorldForwardStep
: (gz-physics, SimulationFeatures.cc) Main simulation entry point that orchestrates pre-step gravity, the DART world step, NaN checking, and pose output to the ECS. The 3.1% self-time is the per-link iteration over 3000 links.

dxHashSpace::collide
: (ODE) Broad-phase collision detection using spatial hashing. Sweeps all shapes in the hash space to find potentially overlapping pairs — including static-vs-static pairs that can never produce contacts.

DegreeOfFreedom::getPosition
: (DART) Returns the current generalized coordinate of a joint DOF. Called per-DOF per-step during skeleton state queries.

std::_Hashtable (26.8%)
: (gz-physics, SimulationFeatures.cc:176–214) Operations on unordered_map<size_t, Pose3d> inside Write(ChangedWorldPoses). A fresh map is allocated every step, 3000 entries inserted (with rehashing), then the old map is destroyed. The allocate-populate-destroy cycle dominates even though no poses change in a static world.

Graph::Vertices (13.5%)
: (gz-sim, EntityComponentManager.hh:310) Called from ProcessRecreateEntitiesRemove

every step via `EachNoCache<Model, Recreate>`. Returns the entire scene graph vertex map (3000 entries), which is then scanned linearly to find entities with `Recreate` components. Zero matches in a static world — pure overhead. The cached `Each()` variant avoids this full scan.

`Frame::getWorldTransform`
(DART) Computes world-space transform by traversing up the kinematic chain and accumulating parent transforms. Called per-link per-step inside `Write(ChangedWorldPoses)`.

4.3 3k_shapes_dynamic (3000 dynamic entities, headless, RTF=0)

Purpose: Full physics with 3000 bodies. Compare with static variant.

Leaf function	%	Category
`[libdart.so.6.13.2]` unresolved	18.3	DART (stripped)
`ProcessRecreateEntities`	**10.8**	**gz-sim: EachNoCache full scan**
`std::_Hashtable` (Write)	**7.5**	**gz-physics: pose map rebuild**
`BodyNode::isReactive`	4.2	DART accessor
`BoxedLcpConstraintSolver`	3.5	DART LCP solver
`BodyNode::getSkeleton`	3.3	DART accessor
`buildConstrainedGroups`	3.0	DART constraint setup

Static→dynamic delta: LCP solver appears (3.5%), `buildConstrainedGroups` appears (3.0%), `BodyNode::isReactive` appears (4.2%). The unresolved `libdart` bucket grows from 0% to 18.3%. `std::_Hashtable` drops from 26.8% to 7.5% as physics work dilutes the ECS overhead. The physics engine cost is the dominant factor with dynamic bodies.

Note that `WorldForwardStep` does not appear in the self-time table above, despite accounting for 55.8% of total time *inclusive*. In the static world, `WorldForwardStep` shows 3.1% self-time because its per-link loops (added-mass gravity, pose output) iterate over 3000 links and the loop overhead itself is measurable. In the dynamic world, the same loops still run, but DART's solver and collision detection dominate so heavily that the loop overhead rounds to 0% self-time — all time is attributed to its children. The flamegraph tables report self-time only, so `WorldForwardStep` disappears from the dynamic table even though it is the parent of all physics work.

`BodyNode::getSkeleton`
(DART) Returns a pointer to the skeleton that owns a body node. A simple accessor, but called per-body in the inner solver loop for every contact constraint — hence 4.5% with 3000 dynamic bodies.

`BoxedLcpConstraintSolver`
(DART) Solves contact constraints using a Linear Complementarity Problem (LCP) formulation. Determines contact forces to prevent interpenetration. Cost scales with the number of active contacts.

`buildConstrainedGroups`
(DART) Partitions contacts into constraint islands (connected components of interacting bodies) to reduce the LCP problem size. Each island is solved independently.

4.4 sensors_nonrendering (IMU, mag, etc., headless, RTF=0)

Leaf function	%	Category
`[unknown]`	34.6	kernel/stripped libs
`[libdart.so.6.13.2]`	5.4	DART
`[[vdso]]`	2.5	timing
`BodyNode::getSkeleton`	1.8	DART
`pthread_cond_wait`	1.8	thread sync (glibc)
`_rmt_BeginCPUSample`	1.5	Remotery overhead

Leaf function	%	Category
pthread_mutex_lock	1.1	thread sync (glibc)
Barrier::Wait	**1.0**	**gz-sim: thread barrier**
BaseView::ResetNewEntityState	**0.8**	**gz-sim: ECS view reset**

Key finding: Thread synchronization (pthread_cond_wait + pthread_mutex_lock + Barrier::Wait) totals ~4%. ECS view operations (BaseView::ResetNewEntityState) visible. This is framework overhead — not physics, not rendering.

Barrier::Wait
: (gz-sim, Barrier.hh) Thread synchronization barrier that blocks worker threads until all have reached the synchronization point, then releases them together. Used to coordinate PreUpdate/Update/PostUpdate system execution phases.

pthread_cond_wait
: (glibc) Underlying POSIX condition variable wait used by Barrier::Wait and other synchronization primitives. The thread sleeps until signaled.

BaseView::ResetNewEntityState
: (gz-sim, detail/BaseView.hh) Clears the "new entity" flag on all cached entities in an ECS view at the start of each step. Allows systems to distinguish newly created entities from existing ones. Cost scales with view size.

4.5 jetty_headless (complex world, headless, RTF=0)

Leaf function	%	Category
dxHashSpace::collide	21.9	ODE broadphase
ProcessRecreateEntities	**15.2**	**gz-sim: EachNoCache full scan**
[unknown]	10.7	kernel/stripped
stbi__YCbCr_to_RGB_simd	**3.1**	**gz-common: JPEG decode**
Frame::getWorldTransform	1.8	DART
stbi__jpeg_decode_block	**1.6**	**gz-common: JPEG decode**
[libdart.so.6.13.2]	1.5	DART
BitmaskContactFilter	**1.5**	**gz-physics: collision filter**
updateEngineData	**1.2**	**gz-physics: DART→ODE sync**
stbi__convert_format	**1.2**	**gz-common: image format**
gz::common::Image::Width	**0.9**	**gz-common: image query**

Key finding: JPEG texture decoding (stbi_* functions) consumes ~6% of CPU during headless runtime. The full call path reveals this happens inside SceneManager::CreateVisual — visuals are created lazily during simulation on the render thread, not at startup. Each new visual triggers Ogre2Material::SetTextureMapDataImpl → Image::RGBAData → stbi_* to decode all material textures from JPEG. BitmaskContactFilter and OdeCollisionObject::updateEngineData are Gazebo-owned at 1.5% and 1.2%.

dxHashSpace::collide
: (ODE) Broad-phase collision using spatial hashing. At 21.9%, it dominates jetty due to the large number of collision shapes in the marina environment.

stbi__YCbCr_to_RGB_simd
: (stb_image, in gz-common) SIMD-accelerated JPEG color-space conversion from YCbCr to RGB. Called during lazy texture decoding when SceneManager::CreateVisual loads material textures on the render thread.

stbi__jpeg_decode_block
: (stb_image) Reconstructs an 8×8 JPEG image block via inverse DCT. Works in tandem with the color-space converter above.

BitmaskContactFilter::ignoresCollision
(gz-physics, EntityManagementFeatures.cc) Checks whether two collision shapes should be filtered based on user-defined bitmask values. Called once per broad-phase pair before narrow-phase testing.

OdeCollisionObject::updateEngineData
(DART/gz-physics) Synchronizes DART skeleton pose state to ODE collision object wrappers. Called per collision object per step, even when the object hasn't moved.

4.6 gpu_lidar (GPU lidar, headless-rendering, RTF=0)

Leaf function	%	Category
[unknown]	21.4	kernel/stripped
[[vdso]]	7.2	timing/GPU sync
[libnvidia-eglcore]	6.9	NVIDIA driver
[libdart.so.6.13.2]	6.4	DART
BodyNode::getSkeleton	4.5	DART
BoxedLcpConstraintSolver	4.1	DART solver
MovableObject::cullFrustum	1.2	Ogre2 culling
SceneManager::cullFrustum	1.0	Ogre2 culling
BaseView::ResetNewEntityState	**0.9**	**gz-sim: ECS view reset**
SceneManager::updateAllLods*	0.8	Ogre2 LOD

Key finding: NVIDIA EGL driver at 6.9% — GPU driver overhead for headless rendering. Ogre2 frustum culling and LOD updates visible but small (~3% combined). Physics still dominates despite lidar being the target.

libnvidia-eglcore
(NVIDIA driver) GPU driver overhead for EGL-based headless rendering. Handles GPU command submission, buffer management, and synchronization without an X server.

Ogre::MovableObject::cullFrustum
(Ogre2) Tests whether a renderable object intersects the camera frustum. Called per-object per-camera to determine visibility.

Ogre::SceneManager::updateAllLodsThread
(Ogre2) Updates level-of-detail selection for all objects based on distance to camera. Runs on a worker thread.

4.7 sensors_demo (6 rendering sensors, headless-rendering, RTF=0)

Leaf function	%	Category
[unknown]	19.6	kernel/stripped
Node::_getDerivedPosition	8.0	Ogre2 scene graph
[libnvidia-eglcore]	5.6	NVIDIA driver
Node::_getDerivedOrientation*	5.5	Ogre2 scene graph
[libdart.so.6.13.2]	4.6	DART
[[vdso]]	4.1	timing
Node::updateFromParentImpl	2.9	Ogre2 scene graph
Frustum::updateWorldSpace*	2.6	Ogre2 frustum
RenderSystem::_makeRsProj*	2.4	Ogre2 projection
SceneManager::cullFrustum	1.5	Ogre2

Key finding: Ogre2 scene-graph transform traversal totals ~22% (_getDerivedPosition + _getDerivedOrientationUpdated + updateFromParentImpl + frustum updates). This is the

dominant cost for multi-sensor rendering — not shader execution, not GPU submission, but CPU-side scene-graph walking. NVIDIA driver at 5.6%.

`Ogre::Node::_getDerivedPosition`
(Ogre2) Computes world-space position of a scene node by traversing up the parent chain. Called per-node per-camera — with 6 cameras, each node is queried 6 times per frame.

`Ogre::Node::_getDerivedOrientationUpdated`
(Ogre2) Computes world-space orientation of a scene node, re-deriving from parent if the dirty flag is set. Companion to `_getDerivedPosition`.

`Ogre::Node::updateFromParentImpl`
(Ogre2) Propagates parent transform changes down to child nodes. Recursive tree traversal that updates local-to-world matrices when any ancestor moves.

`Ogre::Frustum::updateWorldSpaceCornersImpl`
(Ogre2) Computes the 8 frustum corner points in world-space for each camera. Used for visibility culling and shadow map computation.

`Ogre::ForwardClustered::collectLightForSlice`
(Ogre2) Assigns lights to screen-space tiles for Ogre2's forward+ rendering pipeline. Reduces per-pixel light iteration cost by pre-sorting lights into spatial clusters.

---

5 Cross-Reference Analysis

5.1 Gazebo-Owned Hotspots (optimization targets)

The following hotspots are in code maintained by the Gazebo project. These are the primary targets for optimization effort.

Function	Where	%	Actionable fix
`ProcessRecreateEntities*`	all worlds	1.7–45	Switch to cached `Each()`
`Write(ChangedWorldPoses)`	3k_static	26.8	Reuse pose map in place
`DownloadAssets`	all loading	9–30	Skip re-parse when no Fuel URIs
`stbi_*` JPEG decode	jetty	6	Cache decoded textures
`pybind11::initialize_interp*`	all loading	1–7.5	Lazy-init Python when needed
`BitmaskContactFilter`	jetty	1.5	Inline bitmask check
`updateEngineData`	jetty	1.2	Dirty-flag skip unchanged
`Barrier::Wait`	sensors	1–1.8	Lock-free barriers
`BaseView::ResetNewEntity*`	sensors	0.5–0.9	Skip when no entities changed
`SdfModelSerializer`	all loading	1st step	Lazy or binary serialization
`SceneGraphAddEntities`	3k loading	3 load	Batch entity insertion

5.2 External Library Costs (context, not directly actionable)

The following hotspots are in third-party libraries (DART, Ogre, NVIDIA driver, glibc). ODE appears as DART's internal collision detection backend. These cannot be optimized directly in the Gazebo codebase but provide context for understanding CPU distribution. In some cases, Gazebo can reduce how often these are called (e.g., skip broadphase for static bodies).

Function	Library	%	Context
`dxHashSpace::collide`	ODE	3.8–21.9	Broadphase; runs on static bodies too
`[libdart.so]` unresolved	DART	0–18.3	Stripped internals; rebuild with `-g`
`Node::_getDerived*`	Ogre2	0–22	Scene-graph; scales with sensor count
`libnvidia-eglcore`	NVIDIA	5.6–6.9	GPU driver for headless rendering
`BoxedLcpConstraintSolver`	DART	2.1–3.5	LCP solve; scales with contacts
`getSkeleton/isReactive`	DART	3.3–4.5	Per-body accessors in solver loop
`Frame::getWorldTransform`	DART	2.2–4.4	Kinematic chain traversal per link
`_int_free + malloc*`	glibc	1–2	Allocator churn from per-step allocs

---

6 CPU Cache Analysis

In addition to flamegraph-based CPU sampling, hardware performance counters were used to measure cache efficiency via `perf stat`. Instructions Per Cycle (IPC) indicates how efficiently the CPU executes: a modern CPU should achieve 2–4 IPC; values below 1 indicate the CPU is stalled waiting for data from RAM due to cache misses.

6.1 Loading cache performance

World	Load time	LLC misses	IPC	Assessment
3k_shapes_static	32.5s	306M	**0.73**	Memory-bound
3k_shapes_dynamic	32.2s	302M	**0.72**	Memory-bound
jetty	10.9s	96M	**0.79**	Memory-bound
gpu_lidar	2.8s	2.7M	1.08	Moderate
sensors_demo	3.0s	3.6M	0.85	Moderate
sensors	2.3s	1.6M	1.26	Healthy

The loading phase shows entity/model construction for large worlds (3k_shapes, jetty) involves building graph structures and hash maps that thrash the cache during startup.

6.2 Runtime cache performance

World	Cache miss	LLC miss	IPC	Assessment
3k_shapes_static	**72.6%**	**60.6%**	**0.27**	Severely memory-bound
3k_shapes_dynamic	**82.3%**	**75.3%**	**0.91**	Severely memory-bound
jetty	19.3%	11.2%	1.14	Moderate
sensors	5.0%	1.6%	1.95	Healthy
gpu_lidar	2.5%	0.7%	2.17	Healthy
sensors_demo	4.4%	1.9%	2.39	Healthy

The 3k_shapes worlds are **catastrophically cache-hostile**: 73–82% of cache accesses miss, 61–75% of L3 accesses go to main memory, and the CPU achieves only 0.27–0.91 IPC (stalled 70–90% of the time waiting for RAM). This means the `ProcessRecreateEntitiesRemove` scanning 3000 hash-map vertices and `Write(ChangedWorldPoses)` rebuilding a 3000-entry map aren't just algorithmically wasteful — they **thrash every level of the cache hierarchy**.

This adds a second optimization lever beyond algorithmic fixes: **data structure layout**. Switching from `unordered_map` to contiguous storage (e.g., a flat vector sorted by entity ID) would dramatically improve

cache performance by enabling sequential memory access instead of pointer-chasing through heap-allocated hash-map nodes.

6.3 Per-function cache analysis

Cache-miss flamegraphs (captured with `perf record -e cache-misses`) were compared with CPU flamegraphs to identify which Gazebo functions are *cache-hostile* (more cache misses than their CPU share) vs *cache-friendly* (algorithmic cost without memory-layout issues). The "cache ratio" is cache-miss% divided by CPU-time%: values above 1.3 indicate cache-hostile code; below 0.7 indicates cache-friendly code.

Cache-hostile Gazebo functions (ratio > 1.3)

These functions cause disproportionately many cache misses. Fixing them requires better **data layout** (contiguous storage, fewer pointer chases) in addition to algorithmic improvements.

Function	World	CPU %	Cache %	Ratio
`RenderUtil::UpdateFromECM`	gpu_lidar	5.2	10.6	2.0
`Sensors::PostUpdate`	gpu_lidar	5.6	14.7	2.6
`ProcessRecreateEntitiesRemove`	jetty	15.2	27.2	1.7
`PhysicsPrivate::Step`	sensors_demo	21.7	39.6	1.8
`UpdatePhysics`	3k_static	0.8	2.0	2.5
`UpdatePhysics`	sensors	3.9	5.5	1.4

Cache-friendly Gazebo functions (ratio < 0.7)

These functions are algorithmically expensive but their data fits in cache. The fix is to **reduce work** (skip calls, use cached results), not change data layout.

Function	World	CPU %	Cache %	Ratio
`ProcessRecreateEntitiesRemove`	3k_static	45.0	26.9	0.5
`ProcessRecreateEntitiesRemove`	3k_dynamic	10.7	7.2	0.6
`Barrier::Wait`	sensors	38.1	23.0	0.6
`RenderUtil::Update`	jetty	23.4	0.5	0.0
`Rendering pipeline`	jetty	13–24	~0	0.0

Key insight: `ProcessRecreateEntitiesRemove` is cache-friendly in 3k_shapes (ratio 0.5) but **cache-hostile in jetty** (ratio 1.7). The same function exercises different memory access patterns depending on the world's entity structure. The algorithmic fix (use cached `Each()`) benefits both worlds, but jetty would additionally benefit from improved data layout.

All cache-miss flamegraphs and `.folded` files are available in the repository under `2026-04-21/cache/` and can be analyzed with `gz_compare_cpu_cache.sh`.

---

7 Extended Analysis

The following analyses complement the CPU flamegraph and cache analysis to provide a complete picture of multi-threaded simulation performance. Each technique addresses a blind spot in CPU-only profiling.

7.1 Off-CPU Analysis

Off-CPU flamegraphs show where threads *block* — waiting on mutexes, condition variables, I/O, or inter-thread synchronization. A thread can be "slow" without appearing in CPU flamegraphs because it spends most of its time waiting, not computing.

Methodology. Captured using BCC's `offcputime-bpfcc` tool, which attaches an eBPF program to the kernel's context-switch path. Unlike `perf record -e sched:sched_switch` (which failed on perf 6.17.x due to a tracepoint decoding bug), `offcputime` performs in-kernel aggregation and outputs folded

stacks directly — no post-processing issues. Requires `sudo` for eBPF access. The `-U` flag captures user-space stacks only (cleaner output), and `-m 1` filters out sub-microsecond context switches.

Interpretation. In the off-CPU flamegraph, the x-axis represents total blocking time (in microseconds), not CPU time. Functions like `pthread_cond_wait`, `__futex_abstimed_wait`, and `poll` are expected to dominate for I/O threads. The interesting findings are blocking events in *CPU-bound* threads (physics, rendering) — these indicate synchronization bottlenecks that CPU flamegraphs cannot reveal.

Results. Off-CPU blocking was measured across all 6 runtime worlds (5s each).

World	Total off-CPU	Top blocker	Share	Lock wait
3k_shapes_static	28.6s	`poll` (transport)	50.8%	0%
3k_shapes_dynamic	28.4s	`poll` (transport)	51.1%	0%
sensors_nonrendering	47.7s	`futex_wait` (barriers)	67.7%	1.7%
jetty_headless	43.3s	`futex_wait` (barriers)	67.1%	0%
gpu_lidar	165.2s	`pthread_barrier_wait`	72.4%	0%
sensors_demo	39.1s	`futex_wait` (barriers)	62.3%	0%

Key findings:

- **No hidden contention**: the dominant off-CPU cost across all worlds is expected synchronization — `Barrier::Wait` (threads waiting for each other between PreUpdate/Update/PostUpdate phases) and `zmq_poll` (transport threads waiting for messages). These are not optimization targets.
- **gpu_lidar is barrier-dominated**: 72.4% of off-CPU time (119.7s across all threads) is in `pthread_barrier_wait`. This confirms the per-thread analysis finding that rendering threads spend most of their time waiting for the simulation step to complete.
- **Lock contention is negligible**: only `sensors_nonrendering` shows measurable lock wait (1.7%, 805ms in `__lll_lock_wait` on `EntityComponentManager::FindView`). All other worlds show zero lock contention — the ECS mutex is not a bottleneck.
- **3k_shapes worlds are transport-dominated**: 51% of off-CPU time is `poll` (transport threads). The simulation thread itself rarely blocks — it runs continuously until preempted.

7.2 False Sharing Detection

False sharing occurs when threads write to different variables that happen to share the same CPU cache line (typically 64 bytes). Each write invalidates the cache line on other cores, forcing expensive cache coherency traffic even though the threads access logically independent data.

Methodology. Captured using `perf c2c record -F 60000 -all-user`. The `c2c` (cache-to-cache) subcommand samples memory load/store events and records which loads hit modified cache lines from other cores (HITM — Hit-In-Modified). The report groups events by cache line address and shows which symbols and source locations trigger contention.

Key metrics:

- **Local HITM**: the load hits a modified cache line from a sibling core on the same socket.
- **Remote HITM**: the load hits a modified cache line from a core on a different NUMA node (more expensive).
- **Total HITM**: the sum of both, indicating overall cacheline contention.

Interpretation. High HITM rates on a specific symbol indicate true sharing or false sharing contention. The fix depends on the type: true sharing requires algorithmic changes (reduce shared writes), while false sharing requires data layout changes (padding, separation into different cache lines).

Results. False sharing was measured across 5 runtime worlds.

World	Total HITM	Top contention symbol	Share
3k_shapes_static	2	(negligible)	—
3k_shapes_dynamic	2	(negligible)	—
sensors_nonrendering	2,420	`_int_free` (libc)	14.0%
gpu_lidar	1,183	`detail::BaseView` (gz-sim)	11.4%

sensors_demo	903	`Ogre::Node::updateFromParent`	15.8%

Key findings: The headless-only worlds (3k_shapes variants) show virtually zero HITM events — they run single-threaded with no cross-thread contention. Multi-threaded worlds show moderate contention dominated by:

- **Memory allocator** (`malloc`, `_int_free`, `cfree`): 14–30% of HITM in sensor worlds. This is true sharing on heap metadata, common in multi-threaded allocators.
- **ECS Views** (`detail::BaseView`, `detail::View`): 5–11% of HITM. Multiple threads accessing entity component views concurrently.
- **Mutex operations** (`pthread_mutex_lock`, `pthread_cond_wait`): 6–11%. Expected for lock-based synchronization.
- **Ogre rendering** (`Ogre::Node::updateFromParent`, `ForwardClustered`): 10–16% in rendering worlds. Render thread contention on scene graph nodes.

No severe false sharing was detected — the HITM counts are moderate (900–2,400 events over 5s). The contention is primarily true sharing on synchronization primitives and shared data structures, not data layout issues.

7.3 Scheduler Latency Analysis

Scheduling delays occur when a thread is ready to run but no CPU core is available. For real-time simulation, even millisecond-scale delays can cause missed physics steps or jitter in the simulation step rate.

Methodology. Measured using `/proc/PID/task/TID/schedstat`, which provides cumulative CPU time, runqueue wait time (time spent ready-to-run but waiting for a CPU core), and context switch count per thread. Two snapshots are taken at the start and end of the measurement interval, and the delta gives per-thread scheduling behavior during that period. This approach is more reliable than `perf sched` across kernel versions and requires no special permissions.

Interpretation. A thread with average scheduling delay above 1ms is flagged (configurable via `SCHED_THRESHOLD_MS`). This threshold is appropriate for physics engines running at 1kHz step rates. Persistent high delays suggest CPU oversubscription or thread priority issues.

Results. Scheduling delays were measured across 5 runtime worlds (5s each).

World	Main CPU (ms)	Main wait (ms)	Max avg delay	Flagged
3k_shapes_static	4,967	14	1.73ms	YES
3k_shapes_dynamic	5,004	6	0.64ms	NO
sensors_nonrendering	3,230	252	1.81ms	YES
gpu_lidar	1,106	248	0.48ms	NO
sensors_demo	1,999	229	0.31ms	NO

Key findings:

- The headless-only worlds (3k_shapes variants) show the main simulation thread consuming nearly 100% of one CPU core (5,000ms of 5,000ms). Auxiliary threads have low CPU but one thread in each static/nonrendering world shows average delays above 1ms — likely a Remotery or transport thread competing for CPU time on the saturated core.
- Sensor worlds show lower main-thread CPU (1–3s of 5s) because work is distributed across rendering and PostUpdate threads. Wait times are higher (229–252ms total) but spread across many more context switches, keeping average delay well below 1ms.
- **No critical scheduling bottleneck** was found. The flagged threads in 3k_shapes_static and sensors_nonrendering are low-priority auxiliary threads, not the physics simulation thread.

7.4 Per-Thread Flamegraphs

Standard flamegraphs merge all threads into a single view, mixing CPU-bound work (physics, rendering) with I/O threads that accumulate few samples. Per-thread flamegraphs separate these, making each thread's hot path visible.

Methodology. Post-processing on existing `perf_*.data` files using `stackcollapse-perf.pl --tid`. No re-capture needed. Each thread is identified by its comm name and TID. A summary TSV shows sample distribution across threads, identifying which threads are CPU-hot.

Results. Per-thread analysis was run on all 6 runtime captures.

World	Threads	Main thread	CPU %	Top function
3k_shapes_static	1	SimulationRunner	100%	`Hashtable::find` (21.6%)
3k_shapes_dynamic	3	SimulationRunner	99.3%	`libdart` (18.5%)
sensors_nonrendering	12	SimulationRunner	54.6%	`libdart` (10.6%)
jetty_headless	14	SimulationRunner	72.0%	`dxHashSpace::collide` (30.3%)
gpu_lidar	33	SimulationRunner	49.2%	`libdart` (13.0%)
sensors_demo	37	SimulationRunner	28.9%	`libdart` (15.9%)

Key findings:

- **Headless worlds are effectively single-threaded**: 3k_shapes_static has only 1 thread; 3k_shapes_dynamic has 3 but 99.3% of CPU is in the main thread. All optimization effort in these worlds should target the main simulation loop.
- **Sensor worlds distribute work across many threads**: gpu_lidar has 33 threads and sensors_demo has 37. The SimulationRunner thread drops to 29–49% of total CPU. RenderThread accounts for 18–25% in these worlds.
- **Jetty is rendering-dominated**: the RenderThread consumes 25.4% of CPU (texture/mesh loading via `libz` decompression), second only to the main SimulationRunner at 72%.
- **PostUpdate threads show high `[unknown]`**: 52–72% of PostUpdate thread samples are in `[unknown]` stacks (likely Remotery profiler or barrier wait code without debug symbols).

7.5 Differential Flamegraphs

Differential flamegraphs compare two captures (typically before and after an optimization) and highlight what changed. This is the standard technique for validating that an optimization actually reduced CPU time in the target function without introducing regressions elsewhere.

Methodology. Uses `difffolded.pl -n` (with normalization to account for different run durations) from the FlameGraph toolkit. The resulting SVG shows red frames for regressions and blue frames for improvements. A companion TSV lists the top changes ranked by absolute delta.

Results. No differential results are included in this report because no optimizations have been applied yet. Once an optimization from Section 8 is implemented, `gz_diff_flamegraph.sh` should be used to generate a before/after comparison and validate the improvement. See Section 9 for the recommended workflow.

7.6 Unified Summary

The `gz_summary.sh` script combines all available analysis dimensions into a single ranked list of optimization targets. Each Gazebo function receives a composite score (0–100 points) based on:

Dimension	Max Points	Thresholds
CPU time	40	>10% → 40, 5–10% → 25, 1–5% → 10
Cache hostility	20	ratio >2.0 → 20, >1.3 → 10
Off-CPU blocking	15	top-5 blocker → 15, top-10 → 8
False sharing (HITM)	15	>5% HITM → 15, 1–5% → 8
Scheduler delay	10	max >1ms on CPU-hot thread → 10

The summary degrades gracefully: if a dimension is unavailable, its contribution is omitted and the maximum possible score is reduced accordingly. The `evidence_sources` column in the TSV output lists exactly which analyses contributed to each function's score, ensuring traceability.

Cross-world results. The output of `gz_analyze.sh` across all 6 benchmark worlds, filtered to the most actionable targets (excluding call-chain wrappers like `SimulationRunner::Step` that only accumulate child cost):

Function	Score	Worlds	Worst case	CPU range
`ProcessRecreateEntitiesRemove`	40	6/6	3k_dynamic (10.8%)	1.7–45.0%
`EachNoCache<Model, Recreate>`	40	6/6	3k_dynamic (10.8%)	1.5–45.2%
`SensorsPrivate::RenderThread`	45	3/6	gpu_lidar (5.4%)	5.4–23.6%
`Barrier::Wait`	40	3/6	sensors (38.2%)	1.9–38.2%
`Graph<>::Vertices`	40	5/6	3k_static (13.6%)	1.5–13.6%
`EntityMatches`	40	2/6	3k_static (28.1%)	6.6–28.1%
`SimulationFeatures::Write`	25	4/6	3k_static (7.1%)	0.6–7.1%
`BitmaskContactFilter`	25	3/6	3k_static (5.4%)	1.8–6.6%
`loadMesh / CreateVisual`	40	1/6	jetty (13.5%)	jetty only
`ComponentImplementation`	25	3/6	3k_dyn (0.9%)	0.5–1.5% (cache 3.1)

Rows 1–2 are the same call path (`ProcessRecreateEntitiesRemove` calls `EachNoCache`). Row 3 scores highest (45) due to cache hostility in gpu_lidar (ratio 8.1). Rows 5–6 are also part of the `EachNoCache` call chain. Row 10 has low CPU but severe cache hostility (ratio 2.6–3.1), indicating a data layout problem in the ECS component storage.

The full cross-world summary with all 148 functions is in `captures/summary/cross_world_summary.tsv`.

---

8 Optimization Recommendations

The following priorities are ranked using the composite scoring system from Section 7.6. Each function is scored 0–100 across CPU time, cache hostility, off-CPU blocking, false sharing, and scheduler impact. The table below shows the highest composite score each priority achieves across all benchmark worlds, plus the world where it scores highest.

#	Target	Score	Peak world	Evidence
1	`ProcessRecreateEntitiesRemove` / `EachNoCache`	50	jetty (14.8%, cache 1.8)	cpu, cache
2	`SimulationFeatures::Write` (ChangedWorldPoses)	35	3k_static (7.1%, cache 1.5)	cpu, cache
3	Texture decoding (`stbi_*` via `SceneManager`)	40	jetty (23.4%)	cpu
4	Broadphase collision (`dxHashSpace`)	—	jetty (21.9%)	external (ODE)
5	`BitmaskContactFilter`	25	jetty (6.6%)	cpu, cache
6	Loading time (`DownloadAssets`, `SdfModelSerializer`)	—	3k_shapes (26s)	loading-only
7	`Barrier::Wait` / ECS thread barriers	40	sensors (38.2%)	cpu, cache

Priorities 4 and 6 are not scored by `gz_summary.sh`: Priority 4 targets an external library (ODE) that is outside Gazebo's namespace filter, and Priority 6 is a loading-time issue that only appears in startup flamegraphs (not runtime). Priority 7 scores as high as 40 in sensor worlds due to `Barrier::Wait` consuming 38.2% of CPU; however, this is thread synchronization overhead (threads waiting for each other to complete PostUpdate work), not wasted computation — the fix requires architectural changes to the threading model rather than a localized code change.

8.1 Priority 1: Replace EachNoCache in ProcessRecreateEntitiesRemove

- **Evidence**: `ProcessRecreateEntitiesRemove` consumes a major fraction of CPU in **every world**:

World	% of runtime
3k_shapes_static	**45.0%**
jetty_headless	**15.2%**
3k_shapes_dynamic	10.8%
sensors_nonrendering	4.7%
sensors_demo	2.4%

gpu_lidar	1.7%

- **Impact**: Largest Gazebo-owned hotspot across all worlds. Scales linearly with total entity count.
- **Location**: `gz-sim/src/SimulationRunner.cc:1510-1524` calls `EachNoCache<Model, Recreate>`, which iterates **all entities** via `Entities().Vertices()` (EntityComponentManager.hh:310), checking each one with `EntityMatches`. With zero `Recreate` components in any benchmark world, the full entity graph is scanned every step for nothing.
- **Fix**: Switch to the cached `Each<Model, Recreate>()` which uses pre-built component indices instead of scanning all vertices. Alternatively, add an early-exit: `if (!entityCompMgr.HasComponentType<Recreate>()) return;` before the iteration.

8.2 Priority 2: Eliminate per-step pose map rebuild in ChangedWorldPoses

- **Evidence**: `SimulationFeatures::Write` contributes 26.8% in 3k_shapes_static (via `_Hashtable` operations), 7.5% in 3k_shapes_dynamic
- **Impact**: Second-largest Gazebo-owned hotspot in entity-heavy worlds. Scales linearly with link count.
- **Location**: `gz-physics/dartsim/src/SimulationFeatures.cc`, lines 176–214 (`Write(ChangedWorldPoses)`)
- **Root cause**: Every step allocates a fresh `unordered_map<size_t, Pose3d>`, inserts all 3000 link poses (triggering multiple rehashes as the map grows from 0 to 3000 entries), then destroys the old map. Even when no poses change (static world), every link writes into the new map.
- **Fix options**:
  - **Quick**: Add `newPoses.reserve(this->links.size())` after line 183 to pre-allocate and eliminate rehashing.
  - **Better**: Reuse `prevLinkPoses` in place instead of rebuilding from scratch. Update only entries whose pose changed, remove entries for deleted links.
  - **Best**: For static links, skip the pose comparison entirely — if the link is immobile, its pose cannot change.

8.3 Priority 3: Cache texture/image decoding in jetty-class worlds

- **Evidence**: `stbi__YCbCr_to_RGB_simd` + `stbi__jpeg_decode_block` + `stbi__convert_format` at ~6% in jetty headless

- **Impact**: ~6% CPU reduction for mesh-heavy worlds

- **Call path**:

```
RenderThread → SensorsPrivate::RunOnce → RenderUtil::Update
  → SceneManager::CreateVisual → LoadGeometry → CreateMesh
    → Ogre2MeshFactory::LoadImpl → BaseMaterial::CopyFrom
      → Ogre2Material::SetTextureMapDataImpl
        → gz::common::Image::RGBAData
          → Image::Implementation::DataWithChannels
            → stbi__convert_format / stbi__YCbCr_to_RGB_simd
```

- **Root cause**: Visual creation happens **lazily during simulation** — `SceneManager::CreateVisual` is called from `RenderUtil::Update` on the render thread, not at startup. Each time a new visual is created, all its material textures are decoded from JPEG via `Ogre2Material::SetTextureMapDataImpl → gz::common::Image::RGBAData → stbi_*`. This means texture decoding competes with simulation on the render thread.

- **Location**: `gz-rendering/ogre2/src/Ogre2Material.cc` (`SetTextureMapDataImpl`), `gz-common/src/Image.cc` (`RGBAData`, `DataWithChannels`)

- **Fix options**:
  - Cache decoded `Image::RGBAData` results so the same texture isn't decoded twice across materials that share it
  - Move visual creation to a pre-simulation loading phase instead of lazy creation during `RenderUtil::Update`

  - Pre-decode textures at world load time on a background thread, so the render thread never blocks on JPEG decode

8.4 Priority 4: Reduce broadphase cost for worlds with many static bodies

- **Evidence**: `dxHashSpace::collide` at 2.6% in 3k_shapes_static (no dynamic bodies), 21.9% in jetty
- **Impact**: The broadphase dominates collision-heavy worlds. In mixed worlds (jetty: 300+ static models, 1 mobile robot), ODE sweeps all shapes including static-vs-static pairs that can never produce new contacts.
- **Call path**:

```
SimulationRunner::Step → UpdateSystems → Physics::Update
  → PhysicsPrivate::Step
    → SimulationFeatures::WorldForwardStep
      → dart::simulation::World::step                   ← called unconditionally
        → ConstraintSolver::solve
          → ConstraintSolver::updateConstraints
            → GzOdeCollisionDetector::collide            ← Gazebo-owned wrapper
              → OdeCollisionDetector::collide
                → dxHashSpace::collide                   ← sweeps ALL shapes
```

- **Current state**: gz-physics uses a **single collision group per world** (via `getConstraintSolver()` → `getCollisionGroup()`). All bodies — static and dynamic — go into the same group. `BitmaskContactFilter` filters by user-defined bitmask only and never checks whether a body is static. gz-sim *does* track static entities in `PhysicsPrivate::staticEntities` (Physics.cc:365), but this information never reaches the collision layer.
- **Location**:
  - `gz-physics/dartsim/src/EntityManagementFeatures.cc` (BitmaskContactFilter)
  - `gz-physics/dartsim/src/SimulationFeatures.cc` (WorldForwardStep)
  - `gz-sim/src/systems/physics/Physics.cc` (staticEntities tracking)
- **Fix options (incremental)**:
  1. **Quick win — filter static-vs-static in `BitmaskContactFilter::ignoresCollision`**: Add an early-return when both bodies belong to immobile skeletons (DART already exposes `Skeleton::isMobile()`). This skips narrowphase and contact generation for static-vs-static pairs but doesn't eliminate the broadphase sweep itself.
  2. **Architectural — separate collision groups**: Create two collision groups: one for dynamic bodies, one for static. Run broadphase as dynamic-vs-dynamic and dynamic-vs-static only, skipping static-vs-static entirely. This would require changes to how gz-physics constructs and manages DART collision groups, and would need to handle bodies transitioning between static and dynamic.
  3. **Full-static shortcut**: In `WorldForwardStep`, if all bodies in the world are static, skip `world->step()` entirely. Edge case but cheap to implement.
- **Note**: The broadphase sweep (`dxHashSpace::collide`) is ODE-internal and runs before any Gazebo filter is called. Options 1 and 3 help but don't touch the broadphase itself. Only option 2 eliminates the static-vs-static broadphase pairs at the ODE level.

8.5 Priority 5: Optimize BitmaskContactFilter and updateEngineData

- **Evidence**: Combined 2.5-2.7% in collision-heavy worlds (jetty)
- **Impact**: Small per-world, but these are called per-pair and per-object — scales with scene complexity
- **Location**: `GzCollisionDetector.cc` and `EntityManagementFeatures.cc` in `gz-physics/dartsim`
- **Fix**: Inline bitmask check; dirty-flag on updateEngineData to skip unchanged objects

8.6 Priority 6: Reduce loading time for large entity counts

- **Evidence**: 3k_shapes takes 26s to load; `SceneGraphAddEntities` visible in 3k_shapes, recursive `Element::ToString` → `GetExplicitlySetInFile` visible in jetty
- **Impact**: Critical for large-scale simulation startup
- **Location**: `gz-sim/src/systems/scene_broadcaster/SceneBroadcaster.cc`, `gz-sim/src/EntityComponentManager.cc`
- **Fix**: The bottleneck is `SdfModelSerializer::Serialize`, which converts every entity's SDF to string via recursive `Element::ToString` on first PostUpdate. Consider lazy serialization (serialize only when a subscriber requests), caching the serialized form so it isn't recomputed, or replacing the string-based serialization with a binary format. Additionally, `ServerPrivate::DownloadAssets` wastes 9% of loading time. The server parses the SDF twice by design — once for fast GUI startup (discarding models), then again on a background thread to resolve Fuel URIs and recover the models. When no Fuel URIs are present, the first parse already has everything needed. The fix: detect worlds with no Fuel URIs and keep the models from the first parse instead of discarding and re-parsing them. Finally, `pybind11::initialize_interpreter` adds a fixed ~200–260ms to every server startup by unconditionally initializing the Python interpreter (1–7.5% of loading depending on world complexity). Lazy initialization would eliminate this cost for the majority of worlds that don't use Python plugins.

8.7 Priority 7: Investigate ECS View and thread barrier overhead

- **Evidence**: `BaseView::ResetNewEntityState` (0.5-0.9%), `Barrier::Wait` (1% self-time in merged flamegraph; 38.2% of PostUpdate thread CPU in per-thread analysis), `pthread_cond_wait` (1.8%) in sensor worlds
- **Impact**: Small per-world, but represents framework tax on every step
- **Location**: `gz-sim/src/detail/View.hh`, `gz-sim/src/Barrier.cc`
- **Fix**: Investigate if view state reset is needed when entities haven't changed; consider lock-free barriers

---

9 Validating Optimizations

The analysis in this report identified optimization targets using sampling profiling and multi-dimensional scoring. Validating that each optimization delivers the expected improvement requires both *targeted measurement* (did the specific function speed up?) and *regression detection* (did anything else get worse?). The profiling framework provides complementary tools for each.

9.1 Validation tools

Instrumentation profiling (`GZ_PROFILE` / Remotery)
: Best for *targeted per-step timing*. Named scopes provide deterministic timestamps per simulation step (e.g., "this function dropped from 4.5ms to 0.01ms per step"). Requires `ENABLE_PROFILER=ON` at build time. Use for measuring the direct impact of a code change on a specific function.

Differential flamegraphs (`gz_diff_flamegraph.sh`)
: Best for *regression detection*. Compares two `.folded` captures and highlights every function that changed — red for regression, blue for improvement. Catches unintended side effects that targeted measurement might miss. Does not require rebuilding; works with `ENABLE_PROFILER=OFF`.

Unified summary (`gz_summary.sh`)
: Best for *cross-dimensional verification*. Compares composite scores before/after to verify the optimization didn't shift the problem to a different dimension (e.g., reducing CPU time but introducing cache thrashing). Run on both baseline and optimized captures.

Cache analysis (`gz_compare_cpu_cache.sh`, `gz_false_sharing.sh`)
: Use when validating data layout changes (Priority 2 hash map, Priority 4 collision groups). Verify that the cache ratio decreased and HITM events dropped.

Scheduler analysis (gz_sched_analysis.sh)
Use when validating threading changes (Priority 7 barriers). Verify that scheduling delays didn't increase for CPU-bound threads.

9.2 Validation workflow

For each priority optimization:

1. **Baseline**: Capture the benchmark world *before* the change:
   - `gz_flamegraph.sh` to get baseline `.folded`
   - `GZ_PROFILE` with `ENABLE_PROFILER=ON` for per-step timing of the target scope
   - For cache-related fixes: `gz_cache_flamegraph.sh` + `gz_false_sharing.sh`
2. **Optimize**: Apply the code change. Rebuild.
3. **Re-measure**: Run the same benchmark world. Capture the same data as baseline.
4. **Compare**:
   - `gz_diff_flamegraph.sh baseline.folded optimized.folded` — check for regressions
   - Remotery before/after per-step timing — quantify the improvement
   - `gz_summary.sh` on both captures — verify composite score improved
5. **Publish**: `gz_publish_run.sh` to store the optimized capture alongside the baseline for future reference.

9.3 Existing GZ_PROFILE scopes for each priority

As part of this study, 17 new `GZ_PROFILE` points were added to `gz-physics/dartsim` to break the previously opaque `WorldForwardStep` call into observable sub-phases: pre-step gravity, DART world step, NaN checking, pose output, collision detection wrappers, kinematics queries, and SDF model construction methods.

The following scopes are already in place to measure each optimization target:

Priority	Optimization target	GZ_PROFILE scope
1	`ProcessRecreateEntitiesRemove`	`SimulationRunner::ProcessRecreateEntitiesRemove`
2	`Write(ChangedWorldPoses)`	`dartsim::Write::ChangedWorldPoses`
3	Texture decode	Scope needed — add to `Ogre2Material.cc`
4	Broadphase for static bodies	`dartsim::OdeCollide`
5	`BitmaskContactFilter` + `updateEngineData`	Scope needed — add to `EntityManagementFeatures.cc`
6	`DownloadAssets` + SDF re-parse	Scope needed — add to `ServerPrivate.cc`
7	`Barrier::Wait` + ECS views	Scope exists in `SimulationRunner`

Scopes marked "needed" should be added before beginning the optimization work so that the baseline measurement uses the same instrumentation as the post-optimization measurement.

9.4 Recommended benchmark worlds per priority

- **Priority 1** (ProcessRecreateEntities): Use `3k_shapes_static.sdf` (45%) or `jetty.sdf` (15.2%)
- **Priority 2** (Write ChangedWorldPoses): Use `3k_shapes_static.sdf` (26.8%)
- **Priority 3** (texture decode): Use `jetty.sdf` (6%)
- **Priority 4** (broadphase): Use `jetty.sdf` (21.9%)
- **Priority 5** (BitmaskContactFilter): Use `jetty.sdf` (1.5%)
- **Priority 6** (loading time): Use `3k_shapes_static.sdf` (26s loading)
- **Priority 7** (ECS/barriers): Use `sensors.sdf` (4.7%)

All benchmark worlds with `RTF=0` are available in the repository's `worlds/` directory.

---

10 References

[1] B. Gregg, “Flame Graphs,” *ACM Queue*, vol. 14, no. 2, pp. 91-110, 2016. Software available at https://github.com/brendangregg/FlameGraph

[2] B. Gregg, “Off-CPU Flame Graphs,” 2015. Available at https://www.brendangregg.com/FlameGraphs/offcpuflamegraphs.html

[3] J. Mario, “C2C — False Sharing Detection in Linux Perf,” 2016. Available at https://joemario.github.io/blog/2016/09/01/c2c-blog/

[4] KDAB, “Hotspot — the Linux perf GUI for performance analysis.” Available at https://github.com/KDAB/hotspot