Gazebo Server Performance Profiling Report

gz-sim PR #3447 (EnTT ECM) with gz-transport Zenoh 1.8 + shared memory (PRs #868, #842)

Carlos Agüero

caguero@honurobotics.com

September 2026

Contents

1 Introduction

This is the third study in the Gazebo profiling series published at https://caguero.github.io/gz-profiling/. The first run (April 2026) profiled the headless server on `main`; the second (July 2026) profiled the GUI process. This run returns to the server, on a substantially different code base:

- **gz-sim PR #3447** (ECM implementation on top of EnTT), which replaces the Entity Component Manager storage and view machinery.
- **gz-transport PR #868** (Zenoh 1.8.0 integration, part 2) merged with **PR #842** (Zenoh shared memory transport), run with `GZ_TRANSPORT_IMPLEMENTATION=zenoh`.
- Every other Gazebo library at the tip of its `main` branch on 2026-09-01.

The benchmark suite is the April one plus the `moving_robots_and_sensors` world that gz-sim PR #3846 added as a runtime benchmark: two driven vehicles (differential and Ackermann), a two joint arm following a trajectory, four rendering sensors and six non rendering ones. This is closer to what a typical user world looks like than any of the synthetic worlds.

1.1 Goals

- Re-baseline the server on the EnTT ECM and the Zenoh transport, and check whether the April hotspots that were Gazebo owned (ECS views, scene broadcaster serialization at load time, texture decoding at runtime) are still present.
- Measure loading time and steady state real time factor (RTF) for every world.
- Characterize the new mixed world (physics, controllers, sensors, transport) and identify what dominates it.
- Rank the remaining Gazebo owned optimization targets.

1.2 Approach

Every flamegraph in this PDF is a static rendering; the caption of each figure links to the interactive SVG on the site (click frames to zoom, Ctrl+F to search), and the run page https://caguero.github.io/gz-profiling/2026-09-01/ lists all of them, including the per thread splits.

Each world runs headless at `<real_time_factor>0</real_time_factor>` so the server is CPU bound. Two captures are taken per world: a 30 s steady state runtime capture with `perf` attached to the server process, and a loading capture that wraps the whole process launch with `--iterations 1`. Worlds with rendering sensors run with `--headless-rendering` and one `gz topic -e` subscriber per sensor topic, because rendering sensors do no work without a subscriber. A separate unprofiled pass measures RTF and loading wall clock without `perf` overhead, and a per thread split of every runtime capture separates the simulation thread from the sensor and transport threads.

2 Methodology and Reproduction Procedure

2.1 Profiling Technique

CPU flamegraphs are built from Linux `perf` samples at 997 Hz with DWARF call graphs, collapsed with Brendan Gregg's FlameGraph scripts [1]. Every frame in the stack is captured, including DART, ODE, OgreNext, the NVIDIA driver, glibc and the kernel, so third party cost is attributed to the Gazebo function that triggers it. All libraries are built `RelWithDebInfo` with `-fno-omit-frame-pointer`, and `ENABLE_PROFILER=OFF` so the Remotery scopes do not appear in the profiles.

Two methodology changes with respect to April:

- Sampling uses the `task-clock` software event rather than the default `cycles` event. The Core Ultra 9 285HX is a hybrid part; with `cycles`, `perf` opens one event per PMU (`cpu_core` and `cpu_atom`) and `perf script` silently emits only one of them, dropping roughly half of the samples. The July GUI study established this; the server scripts now do the same. As a consequence the weights in the `.folded` files are nanoseconds of CPU time rather than sample counts.
- Step rates are the steady state rate between two `/world/<name>/stats` snapshots taken 10 s and 30 s after launch, never `sim_time / real_time` from a single snapshot. A free running server (`gz-sim-main -s -r` without `--iterations`) starts its loop before the entities exist (`SimulationRunner::Run` only waits for asset creation when an iteration count is given) and steps the still empty world at 200k to 300k iterations per second for the first one to three seconds. For `3k_shapes_static` that burst is about 900k iterations against 239 real steps per second

afterwards, so a single snapshot would report 50x the true rate. Profiles, thread splits, cache counters and loading times are taken after the burst. The capture script records both snapshots.
- Sensor subscriber topics were audited. In April `sensors_demo` subscribed to `/rgbd_camera` and `/segmentation_camera`; neither topic exists (the RGBD sensor publishes `/rgbd_camera/{image,depth_image,points}` and there is no segmentation camera in that world, but there is a second plain camera on `/camera_alone`). Two of the six sensors were therefore idle in April. This run subscribes to all eight real topics.

2.2 Benchmark World Selection

World	Mode	Stress axis
`3k_shapes_static`	headless	3000 static models: framework overhead with no dynamics
`3k_shapes`	headless	3000 dynamic models: physics scaling
`sensors`	headless	non rendering sensors (IMU, magnetometer, altimeter, air pressure, force torque)
`jetty`	headless	complex real world scene: collision heavy, many meshes
`gpu_lidar_sensor`	headless rendering	one GPU lidar, 1 subscriber
`sensors_demo`	headless rendering	six rendering sensors, 8 subscribers
`moving_robots_and_sensors`	headless rendering	PR #3846 world: 2 driven vehicles, arm trajectory, 4 rendering + 6 other sensors, 12 subscribers

The `moving_robots_and_sensors` world needs commands to actually move. The capture script runs a companion `moving_robots_and_sensors.setup.sh` after the world has loaded; it publishes the same `Twist` to both vehicles and the same 1000 point `JointTrajectory` that `BM_MobileRobot` in `test/benchmark/server_run.cc` publishes, so the profiled scenario matches the upstream benchmark.

2.3 Hardware

- **CPU**: Intel Core Ultra 9 285HX, 24 cores (8 P + 16 E, no HT), power profile `performance`
- **GPU**: NVIDIA RTX PRO 3000 Blackwell Laptop, driver 580.159.03 (EGL, headless rendering)
- **Memory**: 125 GB
- **Kernel**: 6.17.0-35-generic, perf 6.17.13
- **Compiler**: GCC 13.3.0

2.4 Software Versions

Workspace `~/rotary_bench_ws`, created with `vcs import` from the Rotary collection on 2026-09-01. Every repository is at the tip of `main` except:

Package	Revision	Note
gz-sim 11.0.0~pre1	`eec8c3fdb`	PR #3447 `luca/entt_playground` (ECM on EnTT)
gz-transport 16.0.0~pre1	`438ddec1`	local merge of PR #868 (`7f6cb3c0`, Zenoh 1.8.0 part 2) and PR #842 (`ca429b33`, Zenoh shared memory)
gz-physics 10.0.0~pre1	`4121e3e`	main
gz-rendering 11.0.0~pre1	`9cd640c`	main
gz-sensors 11.0.0~pre1	`bd1bd7a`	main
gz-common	`7111f3d`	main (includes lazy texture loading for Assimp meshes)
gz-msgs	`ca37940`	main
sdformat	`2ace536`	main

Package	Revision	Note
DART	6.13.2 (`libdart6.13`, OSRF packages)	ODE broadphase via `libdart-collision-ode`
OgreNext	2.3.1 (`libogre-next-2.3`)	
Zenoh	zenoh-c / zenoh-cpp 1.8.0	`transport/shared_memory/enabled=tru` by default (mode `lazy`)

PR #842 conflicted with the head of PR #868 in seven files because #868 had since been simplified (#952). The merge keeps the #952 structure and re-applies the shared memory delta; the full gz-transport unit and integration suites pass under both backends (21/21 unit and 19/19 integration each for ZeroMQ and Zenoh, including the SHM large message test).

Transport configuration used for all captures: `GZ_TRANSPORT_IMPLEMENTATION=zenoh`, default Zenoh 1.8 config (shared memory enabled), default gz-transport SHM threshold (128 KB) and pool (48 MB). Every image, point cloud and the 3k world state messages exceed the threshold and go through shared memory.

2.5 Build Command

```
cd ~/rotary_bench_ws
colcon build --merge-install \
  --cmake-args -DCMAKE_BUILD_TYPE=RelWithDebInfo \
               -DCMAKE_CXX_FLAGS=-fno-omit-frame-pointer \
               -DCMAKE_C_FLAGS=-fno-omit-frame-pointer \
               -DENABLE_PROFILER=OFF -DBUILD_TESTING=OFF
```

2.6 Setup and Running Captures

```
source ~/rotary_bench_ws/install/setup.bash
export GZ_CONFIG_PATH=~/rotary_bench_ws/install/share/gz:$GZ_CONFIG_PATH
export GZ_SIM_MAIN=~/rotary_bench_ws/install/libexec/gz/sim/gz-sim-main
export GZ_SIM_RESOURCE_PATH=<jetty_demo>/models:$GZ_SIM_RESOURCE_PATH
export GZ_TRANSPORT_IMPLEMENTATION=zenoh
export FLAMEGRAPH_DIR=~/FlameGraph
sudo sysctl kernel.perf_event_paranoid=1

git clone https://github.com/caguero/gz-profiling
# Runtime + loading flamegraphs for every world (reads <world>.topics and
# runs <world>.setup.sh automatically)
./gz-profiling/scripts/capture_all.sh gz-profiling/worlds/
# Unprofiled RTF pass (writes <label>_stats.txt only)
SKIP_PERF=1 OUTPUT_DIR=captures/rtf ./gz-profiling/scripts/gz_flamegraph.sh \
    gz-profiling/worlds/jetty.sdf jetty 30 headless
# Per thread split of a runtime capture
./gz-profiling/scripts/gz_per_thread_flamegraph.sh captures/runtime/perf_jetty.data jetty
```

The runtime capture waits 40 s after launch, starts the subscribers, runs the setup script if present, records 30 s, then snapshots `/world/<name>/stats` so sim time, real time and iteration count are stored next to the flamegraph.

3 Executive Summary

The seven worlds were captured on 2026-09-01 (runtime and loading flamegraphs, per thread splits, cache counters and steady state step rates). The headline numbers:

World	RTF (steady state)	Steps/s	Cores busy	Sim thread	Render thread	Loading wall clock
3k_shapes_static	0.24	239	1.00	100%	none	2.05 s
3k_shapes	0.014	14	1.00	100%	none	2.11 s
sensors	50.6	50,644	1.00	100%	none	0.82 s
jetty (4 ms step)	1.98	495	1.00	99.9%	none	1.55 s
gpu_lidar_sensor	50.2	50,156	1.27	73%	26%	1.22 s
sensors_demo	5.3	5,335	1.40	32%	66%	1.19 s
moving_robots_and_sensors	3.0	3,015	1.49	64%	34%	1.22 s

Steps/s and RTF are the steady state rate between two `/world/*/stats` snapshots taken 10 s and 30 s after launch (unpinned, dartsim, RTF 0). "Cores busy" is the CPU time the server consumed during the 30 s capture divided by 30 s. The simulation thread is always saturated; the rendering sensor thread adds a partial second core, and the Zenoh transport threads never exceed 1.2% of the process.

Top findings:

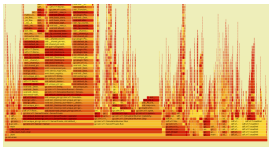
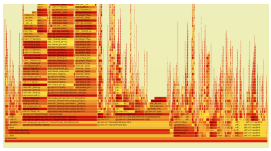
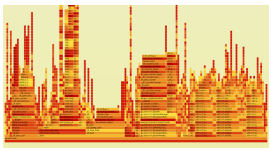
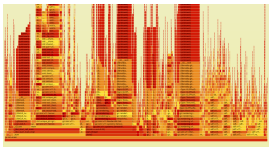
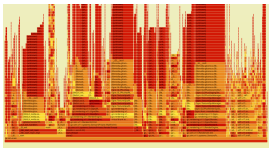
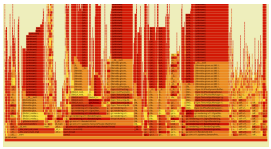
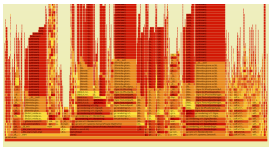
1. **Loading is an order of magnitude faster than in April.** The 3k_shapes worlds load in 2.1 s (26 s in April) and jetty in 1.6 s (7.8 s). The SceneBroadcaster serialization that dominated April's loading profile is gone from the critical path, the ECM construction cost on EnTT is small, and the remaining loading CPU is 0.15 to 1.5 s per world. The rendering worlds are dominated by NVIDIA driver initialization (60 to 70% of loading samples under `ioctl`).
2. **The Gazebo framework has largely disappeared from the steady state profiles.** In `3k_shapes_static` the physics system's `Step` is 97.6% of CPU and everything else in gz-sim (ECM iteration, level manager, transport, scene broadcaster) is below 2%. In `3k_shapes` the ECM is 0.14%. The April framework items (`BaseView::ResetNewEntityState`, `Barrier::Wait`, Remotery) are gone or unmeasurable.
3. **What is left is the physics engine, and two Gazebo owned interfaces to it.** DART's `World::step` is 63% (static), 93% (dynamic), 95% (jetty) and 68% (sensors) of the single threaded worlds. Two costs sit in Gazebo code around it:
   - gz-physics writes a `WorldPoses` vector for every link on every step (`getWorldTransform` per link, plus a 3000 entry vector allocation), then writes `ChangedWorldPoses` doing the same transform query again. gz-sim only reads `ChangedWorldPoses`. In `3k_shapes_static` the double pass plus the allocation is roughly 10% of CPU.
   - gz-sim's `PhysicsPrivate::UpdateSim` and `UpdatePhysics` copy `gz::physics::EntityPtr` handles for every link and joint query, which shows up as `shared_ptr` atomic refcount churn. In the `sensors` world this is 13% of CPU under `UpdateSim` alone.
4. **jetty is a broadphase benchmark.** 86% of jetty's CPU is collision detection, 43% is self time in ODE's `dxHashSpace::collide`, and a further 16% is the per pair contact filter (`BitmaskContactFilter::ignoresCollision` through DART's `BodyNodeCollisionFilter`), most of which is `BodyNodePtr` reference counting. The LCP solver is 3%. The April texture decoding (`stbi_*`, 6% of jetty) is gone after gz-common's lazy texture loading.
5. **Rendering sensors cost about half a core per camera heavy world and are GPU sync bound on the driver side.** On the render thread of `moving_robots_and_sensors`, 50% of the time is inside the NVIDIA EGL driver (35% plus 15% spinning on `clock_gettime`), 15% is OgreNext's per frame scene graph transform update (`Node::updateFromParentImpl`), 9% is `memcpy` of image data, and 35% inclusive is the Forward Clustered light assignment. The CPU side of a plain 640x480 camera is as expensive as the physics of the whole world in `sensors_demo` (Camera 32%, RGBD 20%, thermal 9%).
6. **Zenoh with shared memory is not a measurable cost.** Transport threads (`tx-0`, `rx-*`, `net-0`, `app-0`, `acc-0`) plus the SHM watchdogs sum to 0.6 to 1.2% of the process in the sensor worlds; in the sim thread `gz::transport::` is 0.5 to 3.5% inclusive, most of it protobuf serialization of sensor messages. The 128 KB SHM threshold routes every image and point cloud through shared memory; `zenoh_shm::watchdog` threads are visible in the per thread tables.
7. **Per step fixed overhead of `EachNew<>` views is the one EnTT related item.** In the fast stepping small worlds (`gpu_lidar_sensor` and `sensors` at about 50k steps/s) `entt::` frames sum to 5 to 6% of CPU, spread over

dozens of `EachNew<...>` calls that each construct a view every iteration. It is a small, flat cost and does not grow with entity count.

4 Loading Analysis

4.1 Loading Flamegraphs

Click any thumbnail to open the full interactive flamegraph in a browser (click to zoom, hover tooltips, Ctrl+F search; the link pre-highlights the frames discussed in the text).

World	Wall clock	Flamegraph
3k_shapes_static	2.05 s	
3k_shapes	2.11 s	
sensors	0.82 s	
jetty	1.55 s	
gpu_lidar_sensor	1.22 s	
sensors_demo	1.19 s	
moving_robots_and_sensors	1.22 s	

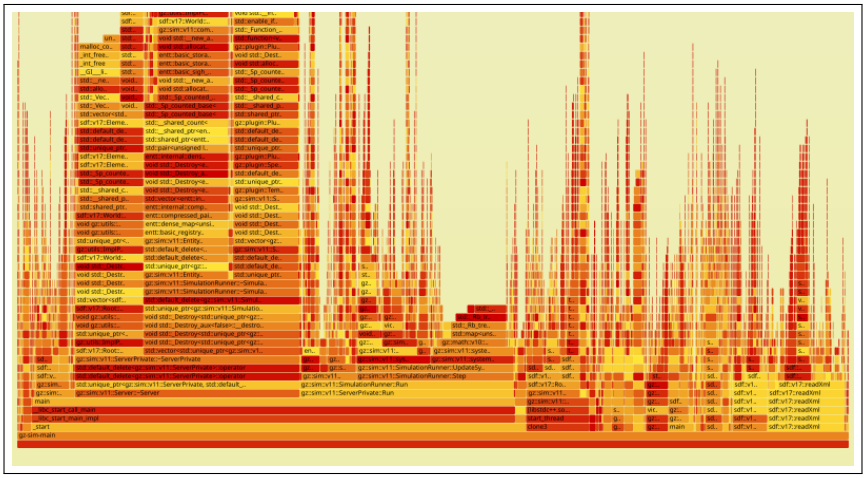
Loading is measured two ways: the wall clock of `gz-sim-main -s -r --iterations 1 <world>` without `perf` (median of three runs) and the CPU time of the same launch under `perf` (which inflates the wall clock to 6 to 15 s because of DWARF stack copying, so only the CPU split is used from it).

World	Wall clock 2026-09	Wall clock 2026-04	CPU under perf	Dominant cost
3k_shapes_static	2.05 s	26.0 s	1.36 s	`sdf::` 69%, SceneBroadcaster 21%, malloc 42%
3k_shapes	2.11 s	26.6 s	1.46 s	same as static
jetty	1.55 s	7.8 s	0.86 s	render engine init 61% (`gz::rendering::`), NVIDIA 37%

World	Wall clock 2026-09	Wall clock 2026-04	CPU under perf	Dominant cost
sensors	0.82 s	2.4 s	0.15 s	dynamic linker 45%, plugin loading 20%
gpu_lidar_sensor	1.22 s	n/a	0.55 s	NVIDIA driver 64%, Ogre init 51%
sensors_demo	1.19 s	n/a	0.56 s	NVIDIA driver 66%, Ogre init 54%
moving_robots_and_sensors	1.22 s	new	0.59 s	NVIDIA driver 62%, Ogre init 50%

The April numbers for the two rendering worlds (0.06 s and 0.03 s) were measurement artifacts and are not comparable.

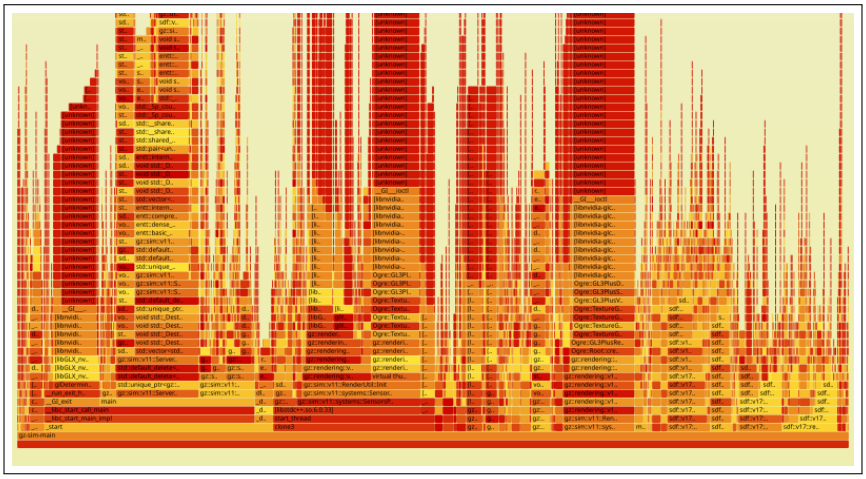
3k_shapes: the 26 s of April are down to 2.1 s. The loading flamegraph still shows the same shape as before, only 12x smaller: 69% of the CPU is under `sdf::` (parsing the 3004 model file and creating entities from it), 21% is `SceneBroadcaster` building its scene graph on the first step, and 40% of all samples are inside `malloc`/`free` (the `gz::math::graph` red black trees that back the scene graph, `shared_ptr` control blocks and SDF element strings). `entt::` accounts for 14% and `SdfEntityCreator` for 5%. The static and dynamic variants load in the same time, so DART skeleton construction is not the bottleneck.



3k_shapes_static loading. SDF parsing and entity creation on the left, SceneBroadcaster scene graph construction in the middle.

Click to open interactive flamegraph

jetty: 1.6 s. Mesh loading is no longer visible (2% under `MeshManager`/Assimp). The loading profile is the OgreNext render engine initialization (`Ogre2RenderEngine`, Hlms shader cache, 29%) and the NVIDIA driver (37%), because jetty carries a rendering sensor and creates an EGL context even headless.



jetty loading. Render engine and driver initialization dominate; SDF and model creation are a small band on the left.

Click to open interactive flamegraph

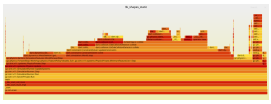
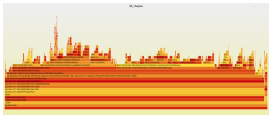
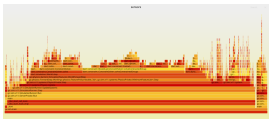
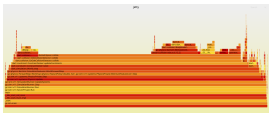
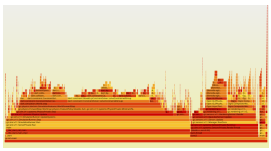
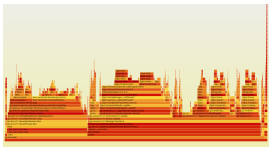
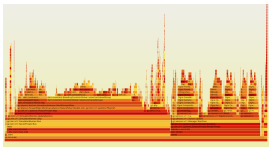
Rendering sensor worlds: 1.2 s each, 60 to 70% of the samples are `[unknown]` frames under `__GI___ioctl`, that is the kernel side of the NVIDIA driver setting up the context and compiling shaders (kernel symbols are not available at `perf_event_paranoid=1`). This is a fixed cost per process and is not Gazebo code.

Common: `do_lookup_x` (dynamic linker symbol resolution while loading system plugins) is 4 to 20% of loading CPU in every world and is the largest item in the small `sensors` world. Prelinking or reducing exported symbols in the system plugins would shave a fraction of a second everywhere.

5 Runtime Analysis Per World

5.1 Runtime Flamegraphs

Click any thumbnail to open the full interactive flamegraph in a browser (click to zoom, hover tooltips, Ctrl+F search; the link pre-highlights the frames discussed in the text).

World	Steady rate	Flamegraph
3k_shapes_static	239 steps/s	
3k_shapes	14 steps/s	
sensors	50,644 steps/s	
jetty	495 steps/s	
gpu_lidar_sensor	50,156 steps/s	
sensors_demo	5,335 steps/s	
moving_robots_and_sensors	3,015 steps/s	

All percentages are of the process's total CPU time during the 30 s capture (all threads). "Self" is the time in the function body itself, "inclusive" includes callees.

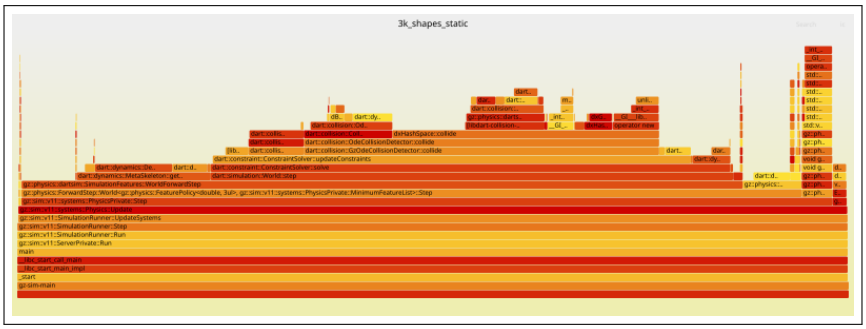
5.2 3k_shapes_static (3000 static models, 1.00 cores, 239 steps/s)

Purpose: framework overhead with no dynamics. Every model is `<static>true</static>`, so DART has nothing to integrate.

Function	Self %	Category
`dart::dynamics::DegreeOfFreedom::getPosition`	9.3	DART accessor
`dart::dynamics::Frame::getWorldTransform`	9.1	DART accessor
`dxHashSpace::collide`	8.8	ODE broadphase
`SimulationFeatures::WorldForwardStep`	6.6	gz-physics dartsim
`CollisionGroup::updateSkeletonSource`	6.2	DART collision prep
`[libdart.so.6.13.2]` (no symbols)	4.8	DART
`dart::dynamics::Skeleton::getDof`	4.2	DART accessor

Function	Self %	Category
`_int_malloc` + `unlink_chunk` + friends	12.5 (inclusive)	glibc, 70% from `dxHashSpace::collide`, 26% from the `WorldPose` vector
`Joint::isKinematic`	3.1	DART
`SkeletonRefCountingBase::incrementReferenceCount`	2.8	DART `BodyNodePtr` refcount

Inclusive: `PhysicsPrivate::Step` 97.6%, of which DART `World::step` 62.8% (collision detection 49.4%, no LCP work) and the pose write back 3.7% plus the `WorldPose` allocations; `PhysicsPrivate::UpdatePhysics` 1.7%; ECM `Each` 1.7%; transport 0.07%; SceneBroadcaster 0.03%.



3k_shapes_static runtime. The whole width is `PhysicsPrivate::Step`; the two towers are ODE broadphase on the left and DART's per link `getWorldTransform` on the right.

Click to open interactive flamegraph

Findings: with the framework gone, the static world exposes two per step full scans that do not need to happen for static bodies:

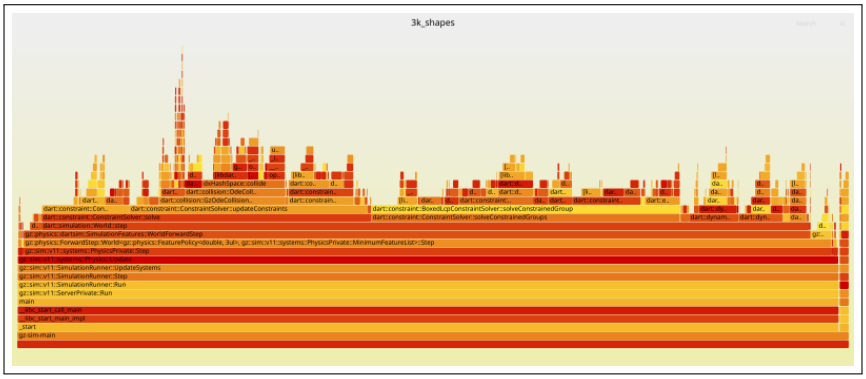
- DART's ODE collision detector rebuilds the hash space and re-collides all 3000 static geometries every step (`dxHashSpace::collide` and `CollisionGroup::updateSkeletonSource` calling `getWorldTransform`). Static vs static pairs are discarded later, but only after the broadphase and the contact filter have run.
- gz-physics `WorldForwardStep` writes `WorldPoses` (all 3000 links) and `ChangedWorldPoses` (all 3000 links compared to their previous pose) on every step, calling `getWorldTransform` twice per link and allocating the 3000 entry `WorldPoses` vector each step. gz-sim only consumes `ChangedWorldPoses`. This is the `getWorldTransform` + `getPosition` + malloc block, roughly 10% of the world.

Compared with April, `dxHashSpace::collide` went from 2.6% to 8.8% of the profile simply because the surrounding framework cost (54% of April's samples were demangling artifacts of ECS view templates) is gone.

5.3 3k_shapes (3000 dynamic models, 1.00 cores, 14 steps/s)

Function	Self %	Category
`[libdart.so.6.13.2]` (no symbols)	18.2	DART
`dart::dynamics::BodyNode::getSkeleton`	10.0	DART accessor
glibc malloc/free (inclusive)	16.6	called from DART's solver and collision
`ConstraintSolver::buildConstrainedGroups`	3.8	DART
`BoxedLcpConstraintSolver::solveConstrainedGroup`	3.2	DART LCP
`BodyNode::isReactive`	2.7	DART
`ConstrainedGroup::getConstraint`	2.5	DART
`Frame::getWorldTransform`	2.0	DART
`BodyNode::getArticulatedInertia`	1.9	DART

Inclusive: `World::step` 93.4%, `ConstraintSolver::solve` 77.0% (`BoxedLcpConstraintSolver` 37.2%), collision 16.0%, forward dynamics 12.5%. Gazebo owned: `UpdateSim` 0.62%, `ChangedLinks` 0.65%, `UpdateModelPose` 0.44%, ECM 0.14%.



3k_shapes runtime. DART's constraint solver dominates; gz-sim is the thin band at the bottom.

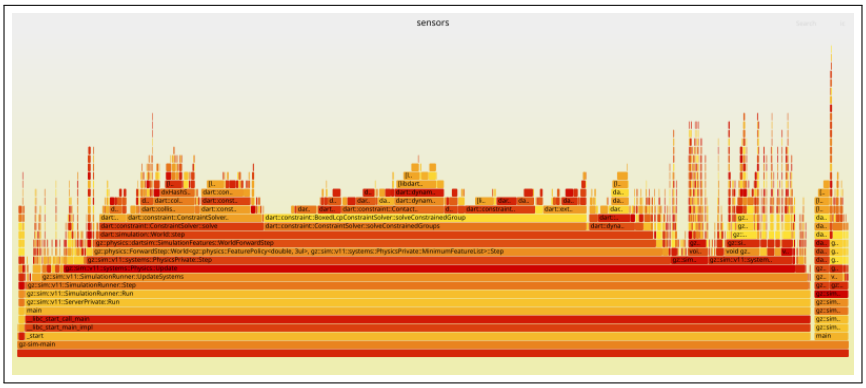
Click to open interactive flamegraph

Findings: the static to dynamic delta (LCP solver, articulated inertia, `buildConstrainedGroups`) is the pure physics cost of 3000 resting bodies and matches April. At 14 steps per second (70 ms per step, 17x slower than the static variant) the dynamic world is by far the most expensive of the suite: DART does not put resting bodies to sleep, so the constraint solver runs on all 3000 contacts every step. Nothing in this world is actionable from gz-sim; it is a DART benchmark.

5.4 sensors (non rendering sensors, 1.00 cores, 50,644 steps/s)

Function	Self %	Category
`[libdart.so.6.13.2]`	18.8	DART
`BodyNode::getSkeleton`	11.5	DART accessor
`BodyNode::isReactive`	3.4	DART
`BoxedLcpConstraintSolver::solveConstrainedGroup`	3.3	DART LCP
`std::_Sp_counted_base<...>`	3.1	shared_ptr refcount
`__exchange_and_add_dispatch`	2.5	shared_ptr refcount
`_int_free`	2.6	glibc
`ConstrainedGroup::getConstraint`	2.4	DART
`BodyNode::getArticulatedInertia`	2.2	DART

Inclusive: `World::step` 68.5%; **`PhysicsPrivate::UpdateSim` 13.2%**, `PhysicsPrivate::UpdatePhysics` 4.5%; ECM `Each` 14.8% (77% of it under `UpdateSim`, 17% under `UpdatePhysics`); transport 3.5%; non rendering sensor `Update` calls 3.6%; SceneBroadcaster 1.2%; `entt::` 5.2%.



sensors runtime. The right hand third is `PhysicsPrivate::UpdateSim` querying link kinematics through gz-physics.

Click to open interactive flamegraph

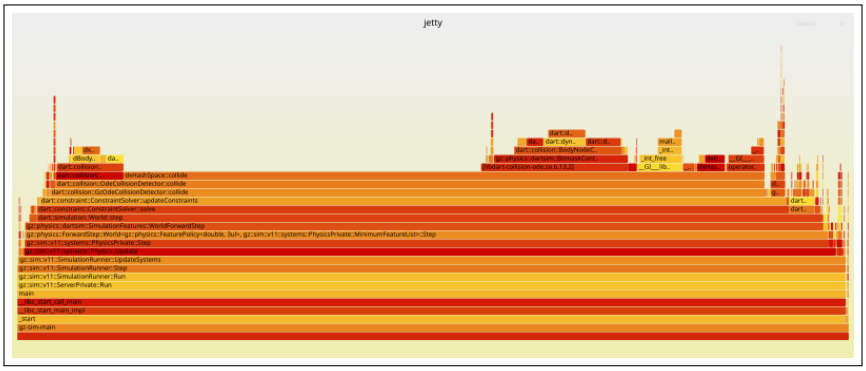
Findings: this small world steps about 50,000 times per second, so per step fixed costs are visible. The largest Gazebo owned item is `UpdateSim`: its `Each<Pose, LinkTag, WorldLinearVelocity ...>` loops call `Link::FrameDataRelativeToWorld`, `WorldLinearAcceleration`, and similar gz-physics queries once per link per step. Each query goes through `gz::physics::EntityPtr`, whose copy semantics

increment and decrement a `shared_ptr` control block, and through `FrameSemantics::Frame` resolution (`KinematicsFeatures::SelectFrame`, `Frame::getSpatialVelocity`, `getLinearAcceleration`). The leaf profile under `UpdateSim` is 12% `_Sp_counted_base`, 11% `__exchange_and_add_dispatch`, then Eigen and DART frame math. April saw `Barrier::Wait`, `pthread_cond_wait` and `BaseView::ResetNewEntityState` here (about 4%); those are gone and `pthread_*` is 1.2%.

5.5 jetty (complex real world scene, 1.00 cores, 495 steps/s, 4 ms steps)

Function	Self %	Category
`dxHashSpace::collide`	**43.0**	ODE broadphase
`SkeletonRefCountingBase::incrementReferenceCount`	4.6	DART `BodyNodePtr`
`SkeletonRefCountingBase::decrementReferenceCount`	4.3	DART `BodyNodePtr`
glibc malloc/free (inclusive)	12.3	70% called from `dxHashSpace::collide`
`[libdart.so.6.13.2]`	2.9	DART
`BodyNode::getSkeleton`	2.6	DART
`gz::physics::dartsim::BitmaskContactFilter::ignoresCollision`	2.4	gz-physics
`dxGeom::computePosr`	2.4	ODE geometry transform
`Frame::getWorldTransform`	2.3	DART
`dxSafeNormalize4`	2.1	ODE

Inclusive: `World::step` 94.6%, collision detection 86.3% (all of it `OdeCollisionDetector::collide`), `dxHashSpace::collide` 77.1% (56% of that is its own body, 23% the per pair near callback in `libdart-collision-ode`, 13% malloc/free), `BitmaskContactFilter::ignoresCollision` 16.2%, `BoxedLcpConstraintSolver` 3.0%, gz-sim `Sensors::PostUpdate` 0.8%, ECM 0.6%.



jetty runtime. One tower: ODE's hash space broadphase under DART's `OdeCollisionDetector::collide`.
Click to open interactive flamegraph

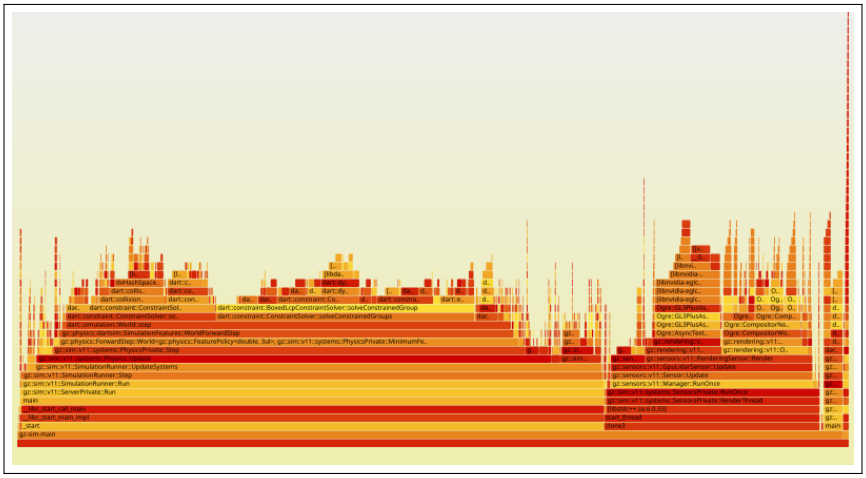
Findings: jetty is now a pure broadphase benchmark. Its 408 collision geometries (360 boxes, 39 meshes, 46 cylinders, most of them static warehouse furniture) are inserted into ODE's hash space, which recomputes every geometry's AABB (`computePosr`) and re-hashes the whole space every step; the per pair callback then runs the contact filter for every candidate pair. The filter path is the Gazebo owned part: `BitmaskContactFilter::ignoresCollision` (gz-physics) first defers to DART's `BodyNodeCollisionFilter::ignoresCollision` (81% of the filter's time), which resolves `Node::getBodyNodePtr` handles for both objects and pays two atomic refcount updates per candidate pair; the pair of refcount functions is 9% of the whole world. Only 15% of the filter time is the bitmask test itself. The April `stbi_*` texture decoding (6%) is gone.

5.6 gpu_lidar_sensor (one GPU lidar, 1.27 cores, 50,156 steps/s)

Thread	Share	Top function
simulation (gz-sim-main)	73.0%	DART solver

Thread	Share	Top function
sensors render thread	26.0%	`[libnvidia-eglcore]` 34% of the thread
`tx-0` (Zenoh)	0.6%	
SHM watchdog threads	0.02%	`zenoh_shm::watchdog::validator`

Inclusive: `World::step` 56.0% (`BoxedLcpConstraintSolver` 31.1%), `GpuLidarSensor::Update` 24.2% (83% of it `RenderingSensor::Render`, 17% `Lidar::PublishLidarScan`, where `Lidar::Clamp` alone is 1.6% of the process), NVIDIA driver 14.1%, `[[vdso]] clock_gettime` 4.9% (spinning in the driver), `gz::rendering::` 20.9%, `entt::` 6.3%, transport 2.7% (Zenoh 1.0%), ECM 1.4%.



gpu_lidar_sensor runtime. Simulation thread (left, DART) and the sensors render thread (right, `Ogre2GpuRays` and the NVIDIA driver).

Click to open interactive flamegraph

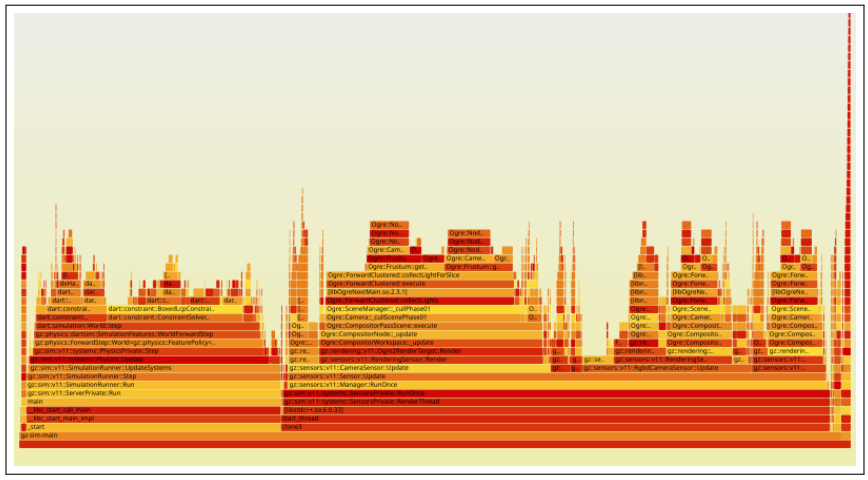
Findings: physics still dominates the process even though the lidar is the target. On the render thread the cost is the driver (render, readback, fence wait) rather than OgreNext. `Lidar::Clamp` and `PublishLidarScan` (copying the range buffer into the `LaserScan` message) are the only Gazebo owned leaves of note. `entt::` at 6.3% is the flat per step overhead of the `EachNew<>` calls in every system (see the cross reference section).

5.7 sensors_demo (six rendering sensors, 1.40 cores, 5,335 steps/s)

Thread	Share	Top function
sensors render thread	66.4%	`Ogre::Node::updateFromParentImpl` 22.7% of the thread
simulation (gz-sim-main)	32.1%	DART
`tx-0` (Zenoh)	1.1%	

Function	Self %	Category
`Ogre::Node::updateFromParentImpl`	15.1	OgreNext scene graph
`[libnvidia-eglcore]`	8.3	NVIDIA driver
`[libdart.so.6.13.2]`	7.0	DART
`Ogre::ForwardClustered::collectLightForSlice`	6.8	OgreNext light culling
`__memcpy_avx_unaligned_erms`	5.9	image copies
`[[vdso]]`	4.2	driver spin on `clock_gettime`
`Ogre::Frustum::updateFrustumPlanesImpl`	3.7	OgreNext
`BodyNode::getSkeleton`	3.6	DART
`gz::sensors::PointCloudUtil::FillMsg`	2.9	gz-sensors

Per sensor (inclusive of `Update`): camera 31.8%, RGBD camera 20.5%, thermal 8.9%, depth 2.3%, GPU lidar 1.5%. Physics is 29% of the process. `gz::transport::` is 3.5% and Zenoh proper 1.7%.



sensors_demo runtime. Render thread on the right (two thirds of the process), simulation thread on the left.

Click to open interactive flamegraph

Findings: this is the first run in which all six sensors are active (April subscribed to two non existent topics, see Methodology), so the render thread is bigger than before. Per frame, OgreNext walks the full scene graph (`updateFromParentImpl`, `_getDerivedOrientationUpdated`) and runs Forward Clustered light assignment for every camera, whether or not anything moved; that is the same pattern the July GUI study found in the GUI's render thread. The driver cost is GPU sync. On the Gazebo side the image copies (`memcpy` 5.9%, `BaseCamera::Copy` 7% of the sensor time) and `PointCloudUtil::FillMsg` (2.9%) are the visible pieces.

5.8 moving_robots_and_sensors (PR #3846 world, 1.49 cores, 3,015 steps/s)

This is the new benchmark world: two driven vehicles, a two joint arm following a 1000 point trajectory, a contact sensor with the touch plugin, a camera + GPU lidar pair, an RGBD camera, a thermal camera, and a box with altimeter, air pressure, air speed, IMU and magnetometer sensors, plus a force torque sensor.

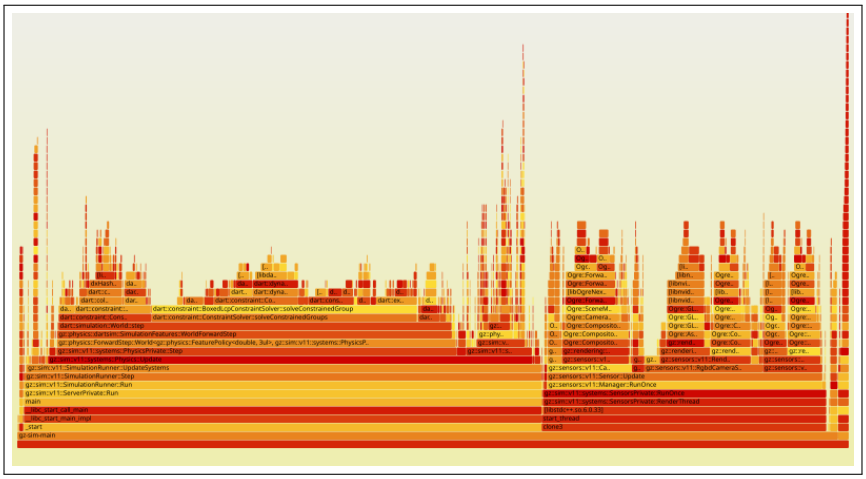
Thread	Share	Top function
simulation (gz-sim-main)	64.1%	DART
sensors render thread	34.3%	`[libnvidia-eglcore]` 19.8% of the thread
`tx-0` (Zenoh)	1.2%	
SHM watchdog threads	0.01%	

Function	Self %	Category
`[libdart.so.6.13.2]`	11.6	DART
`[libnvidia-eglcore]`	6.8	NVIDIA driver
`BodyNode::getSkeleton`	6.3	DART
`Ogre::Node::updateFromParentImpl`	4.9	OgreNext scene graph
`[[vdso]]`	4.9	driver spin on `clock_gettime`
`__memcpy_avx_unaligned_erms`	3.3	image copies
`BoxedLcpConstraintSolver::solveConstrainedGroup`	3.1	DART LCP
`BodyNode::getArticulatedInertia`	2.9	DART
`Ogre::ForwardClustered::collectLightForSlice`	2.2	OgreNext

Inclusive, simulation side: `World::step` 48.5% (`BoxedLcpConstraintSolver` 31.9%, collision 5.8%), `PhysicsPrivate::UpdateSim` 8.1%, `UpdatePhysics` 1.1%, the controllers (DiffDrive, AckermannSteering, JointTrajectoryController, TouchPlugin, JointStatePublisher) 1.9% together, non rendering sensor updates 0.95%,

ECM 2.6%, `entt::` 1.6%. Sensor side: RGBD camera 14.3%, camera 10.3%, thermal 7.2%, GPU lidar 1.5%, `PointCloudUtil::FillMsg` 2.9%. Transport: `gz::transport::` 2.3%, Zenoh 1.8%, protobuf 3.0%, `memcpy` 3.4%.

On the render thread alone: NVIDIA driver 35% plus 15% `clock_gettime` spinning inside it, `updateFromParentImpl` 14%, `memcpy` 9%, Forward Clustered light assignment 35% inclusive, `FillMsg` 4%, publishing 4%.



moving_robots_and_sensors runtime. Simulation thread (left) with DART and `UpdateSim`; sensors render thread (right) with the four rendering sensors.

Click to open interactive flamegraph

Findings: the mixed world runs at 3x real time and is bounded by the simulation thread (64% of CPU, a full core). Half of that thread is DART integrating the two vehicles and the arm (the LCP solver on wheel contacts), and the Gazebo owned remainder is the same `UpdateSim` kinematics query pattern seen in `sensors` (8%). The controllers themselves are cheap (under 2%). Sensors take a third of the process on their own thread and are GPU sync bound; they do not slow the simulation thread except through `Sensors::PostUpdate`, which is 0.7%. Transport, including SHM publication of 640x480 images and the RGBD point cloud, is about 2% of the process.

6 Per Thread Analysis

`gz_per_thread_flamegraph.sh` splits each capture by thread id. The interactive per thread flamegraphs are published next to the runtime ones.

World	Sim thread	Render thread	Zenoh `tx-0`	Other
3k_shapes_static	100.0%			
3k_shapes	100.0%			
sensors	100.0%			
jetty	99.9%	0.0%		
gpu_lidar_sensor	73.0%	26.0%	0.6%	0.2%
sensors_demo	32.1%	66.4%	1.1%	0.2%
moving_robots_and_sensors	64.1%	34.3%	1.2%	0.1%

Zenoh's runtime threads (`tx-0`, `rx-0/1`, `net-0`, `app-0`, `acc-0`) and the shared memory watchdog threads (`Watchdog_Valida`, `Watchdog_Confir`) appear in every capture; only `tx-0` accumulates measurable time, and only in the worlds that publish images and point clouds. The transport is a negligible consumer of CPU with this configuration.

7 Cross Reference: Gazebo Owned Hotspots

Item	Owner	Where	Cost	Notes
`WorldPoses` written and discarded every step	gz-physics dartsim / gz-sim physics	3k_shapes_static	~10%	`Write(WorldPoses&)` + `Write(ChangedWorldPose` both call `getWorldTransform` per link; gz-sim reads only `ChangedWorldPoses`
`EntityPtr` shared_ptr churn in `UpdateSim/UpdatePhysics`	gz-sim physics + gz-physics	sensors, moving_robots	13% / 8%	one refcount pair per link query per step
`BitmaskContactFilterData` and `BodyNodePtr` refcounts per candidate pair	gz-physics dartsim (+ DART)	jetty	16% inclusive	called from ODE near callback for every AABB overlap
Static geometry re-broadphased every step	DART ODE collision detector	3k_shapes_static, jetty	49% / 86%	not Gazebo code, but Gazebo chooses the detector and inserts static geoms
`EachNew<>` view construction per step	gz-sim ECM (EnTT)	gpu_lidar, sensors	5 to 6%	flat, per call, independent of entity count
Image copies and `PointCloudUtil::FillMsg`	gz-sensors	sensors_demo, moving_robots	6 to 9%	`BaseCamera::Copy` + `memcpy` + point cloud fill on the render thread
`Lidar::Clamp` / `PublishLidarScan`	gz-sensors	gpu_lidar	4%	per range element clamp and copy
Per frame full scene graph transform update	OgreNext via gz-rendering	sensors_demo, moving_robots	15 to 22% of render thread	same as the GUI study's Priority 1
SceneBroadcaster scene graph build	gz-sim	3k loading	21% of 1.4 s	no longer on the critical path

8 Cache Analysis

`gz_cache_stats.sh --pid <PID> 10` runs `perf stat` on the unprofiled server for 10 s (P core counters; the process ran on P cores throughout).

World	IPC	L1d miss	LLC load miss	cache miss (all refs)	Instructions / 10 s
3k_shapes_static	**0.71**	**8.8%**	15.7%	34.1%	36.6 G
3k_shapes	1.61	1.2%	54.3%	67.3%	80.2 G
sensors	2.71	0.1%	6.2%	20.1%	131.8 G
jetty	2.11	1.9%	0.5%	3.0%	103.6 G
gpu_lidar_sensor	2.59	0.3%	6.6%	9.2%	158.4 G
sensors_demo	2.65	0.7%	39.0%	30.7%	179.1 G
moving_robots_and_sensors	2.47	0.8%	37.0%	33.2%	177.6 G

Interpretation:

- **`3k_shapes_static` is memory bound** (IPC 0.71, one L1 miss every 11 loads). The static world executes only 37 G instructions in 10 s, less than half of the dynamic world, yet takes the same wall time per step. The per step scans over 3000 skeletons (ODE hash space rebuild, `updateSkeletonSource`, the two `getWorldTransform` passes) chase pointers through DART's heap allocated `BodyNode`/`Skeleton`/`Frame` objects with no locality. This is the strongest argument for Priority 1 and 4: the work is not expensive to compute, it is expensive to touch.
- **`3k_shapes` is LLC bound on the solver** (IPC 1.61, 54% LLC miss): DART's constraint groups and the LCP matrices for 3000 bodies exceed the last level cache.
- **jetty is compute bound** (IPC 2.11, 0.5% LLC miss): the hash space broadphase over 408 geometries fits in cache; the fix there is algorithmic (do less pair testing), not data layout.
- **The sensor worlds are healthy** on the simulation side (IPC 2.5 to 2.7), and their LLC misses (37 to 39%) come from the render thread streaming image buffers (`memcpy`, `FillMsg`, driver readback), which is expected.

9 Optimization Recommendations (Gazebo owned, priority ranked)

9.1 Priority 1: Remove the per step bookkeeping around `WorldForwardStep`

`SimulationFeatures::WorldForwardStep` (gz-physics dartsim) surrounds DART's `World::step` with bookkeeping that runs for every model and every link on every step. (Its full `WorldPoses` pass is not part of it: `WriteRequiredData` only writes un-queried entries and gz-sim's cached `stepOutput`, gz-sim #3685, is never queried for `WorldPoses`, so that pass runs once.) In the `3k_shapes_static` profile the per step items are:

Cost inside `WorldForwardStep`	% of process CPU
NaN check: `MetaSkeleton::getPositions()` for every model (a heap allocated vector per model, then a copy into `lastGoodPositions`)	16.1
`Write(ChangedWorldPoses &)`: `getWorldTransform` + compare for every link	~7
Two deep copies of the `ForwardStep::Output` in gz-sim (`Step` returned the cached member by value, `Update` copy assigned it)	3.7

DART does not integrate immobile skeletons, which is how dartsim represents static models, so none of this work can change anything for them. The recommended change, implemented and measured as part of this study: gz-sim returns the cached output by reference and hands it to `ChangedLinks` (an empty output while paused keeps the ECM fallback; merged upstream as gz-sim #3850 while this study was running), and dartsim (pull request on branch `caguero/dartsim_static_bookkeeping`) skips immobile skeletons in the NaN check (reading DOF positions into a preallocated buffer for the others), marks links of immobile skeletons as settled after one unchanged report and skips them until a pose command, a static toggle or a joint change invalidates them (one epoch counter), and skips the fluid added mass loop when no link uses it. Result on this machine, steady state, same worlds:

World	before	after	change
3k_shapes_static	239 steps/s	440 steps/s	1.85x
sensors	50.6k steps/s	51.6k steps/s	+2%
jetty	495 steps/s	506 steps/s	+2%
3k_shapes, moving_robots_and_sensors			within noise

In the static world's profile DART's `World::step` goes from 63% to 90% of the process. What remains (Priority 4) is the broadphase itself.

9.2 Priority 2: Remove `EntityPtr` refcount traffic from the per link kinematics queries

`UpdateSim` and `UpdatePhysics` look up `gz::physics::EntityPtr` handles for every link and joint on every step. Each lookup returns a copy, and the frame semantics queries take handles by value, so the sensors world pays two atomic operations per query on a control block shared by all handles. `EntityFeatureMap::Get` returns the `EntityPtr` by value; returning a reference into the map and passing `const EntityPtr&` through the query path would remove 5 to 6% from sensor heavy worlds.

9.3 Priority 3: Cheaper contact filtering in the dartsim collision callback

`BitmaskContactFilter::ignoresCollision` is invoked for every AABB overlap that ODE's hash space reports. It defers to DART's `BodyNodeCollisionFilter`, which resolves `BodyNodePtr` handles (atomic refcounts) for both objects, before applying the bitmask. Testing the bitmask first (it is a cheap map lookup and rejects most pairs in jetty), and adding a raw pointer check for the common cases (both objects static, or same skeleton with self collision disabled) before falling back to DART's filter, would cut most of jetty's 16%.

9.4 Priority 4: Keep static geometry out of the per step broadphase

Both static worlds spend half or more of their time re-inserting static geometry into ODE's hash space and re-testing static vs static pairs. DART's `OdeCollisionDetector` treats all geoms alike. Splitting static geometry into a separate, precomputed ODE space (`dSpaceCollide2` between the dynamic space and the static space) or using a DART collision group per static skeleton that is not updated after creation would remove the largest cost of `3k_shapes_static` and jetty. This is a gz-physics dartsim change even though the code that runs is DART's.

9.5 Priority 5: Reduce per step `EachNew<>` view construction

The EnTT ECM constructs a view or group for every `Each/EachNew` call. Systems call `EachNew` for many component combinations on every step to detect additions, and in fast stepping worlds those calls add up to 5 to 6% of CPU even when nothing is new. Caching views per call site, or short circuiting `EachNew` when the ECM has no new entities this step, would remove the cost entirely.

9.6 Priority 6: Avoid the extra image copies on the render thread

`BaseCamera::Copy` plus the `Image::set_data` copies and `PointCloudUtil::FillMsg` are 6 to 9% of the render thread. With the SHM transport the payload could be serialized straight into the shared memory buffer instead of staging through a `gz::msgs::Image` heap buffer and a serialized string. gz-transport currently exposes `PublishRaw` (which still copies) and no loan style API; adding one is the prerequisite, and this run shows the copies are now a visible fraction of the sensor pipeline.

9.7 Not Gazebo owned but worth tracking

- The OgreNext per frame scene graph update and Forward Clustered light assignment per camera (15 to 35% of the render thread). Render on demand for static regions of the scene, or a single light assignment shared by co-located cameras, is a gz-rendering change on top of OgreNext.
- The NVIDIA driver's sync spinning (`clock_gettime` under EGL) suggests the sensor pipeline waits for readback synchronously; asynchronous readback with a one frame delay would hide it.

10 Comparison with the April 2026 Run

Item	April 2026 (main, ZeroMQ)	September 2026 (PR #3447, Zenoh SHM)
3k_shapes loading	26 s	2.1 s
jetty loading	7.8 s	1.6 s
ECS view / demangling noise in 3k_shapes_static	54% of samples	gone; ECM `Each` 1.7%
`Barrier::Wait`, `pthread_cond_wait` in sensors	~4%	1.2% (`pthread_*`)
jetty `stbi_*` texture decoding	6%	0%
jetty `dxHashSpace::collide` self	21.9%	43.0% (share, not absolute)
Remotery `_rmt_BeginCPUSample`	1.5%	not built
Transport threads	not separated	0.6 to 1.2% (Zenoh + SHM)
sensors_demo active sensors	4 of 6	6 of 6

The comparison is qualitative: the April run sampled `cycles` on a hybrid CPU (half of the samples dropped) and ran with Remotery instrumentation on. Absolute steps per second were not recorded in April.

11 Validating Optimizations

For each priority above the validation is a differential flamegraph (`gz_diff_flamegraph.sh baseline.folded optimized.folded <name>`) of the relevant world plus the unprofiled RTF pass (`SKIP_PERF=1`). Suggested pairs:

Priority	World	Metric
1	3k_shapes_static	`Frame::getWorldTransform` + `WorldPose` allocation share; steps/s
2	sensors	`_Sp_counted_base` share under `UpdateSim`; steps/s
3, 4	jetty	`dxHashSpace::collide` and `ignoresCollision` inclusive; steps/s
5	gpu_lidar_sensor	`entt::` inclusive; steps/s
6	sensors_demo	render thread `memcpy` + `FillMsg`; RTF

12 References

1. B. Gregg, FlameGraph, https://github.com/brendangregg/FlameGraph
2. Gazebo profiling repository, https://github.com/caguero/gz-profiling
3. gz-sim PR #3447, ECM implementation with EnTT, https://github.com/gazebosim/gz-sim/pull/3447
4. gz-sim PR #3846, Benchmark for world with moving robots and sensors, https://github.com/gazebosim/gz-sim/pull/3846
5. gz-transport PR #868, Integrate Zenoh 1.8.0 part 2, https://github.com/gazebosim/gz-transport/pull/868
6. gz-transport PR #842, Enable Zenoh shared memory, https://github.com/gazebosim/gz-transport/pull/842
7. Gazebo GUI profiling report, July 2026, https://caguero.github.io/gz-profiling/2026-07-03/