Idle Server Profiling Analysis

Based on the uploaded Java Flight Recorder session collected after starting the server, logging in a player, and leaving the server online for roughly 14 hours.

Bottom line

The server is **not CPU-bound at idle**, but it **is allocation-bound at idle**. In practice that means the JVM spends very little time doing heavy compute, while still creating a large amount of short-lived garbage from repeated world scans and background pathfinding.

0.68%	~10.1 s	~3.42 ms	~50.9 MB/s
Average JVM CPU usage	Total GC pause time across ~13h 5m	Median GC pause	Approximate allocation rate

Executive summary

- This recording does **not** show a classic retained-heap memory leak. The post-GC floor appears fairly stable rather than climbing without recovery.
- The profile does show **heavy garbage churn** for an idle server. Most of that churn appears to come from repeated aggression scanning and world/chunk lookups.
- A background worker thread is also spending substantial time and allocation budget on **pathfinding and collision checks**, which is unusual for a server that is otherwise supposed to be idle.
- Today this waste is mostly hidden because CPU and pause times are still low. Under a higher player, NPC, or bot load, this pattern is likely to scale poorly.

What the profile points to

The hottest idle-time path in the recording centers on aggression processing. The strongest recurring stack was a chain through **NpcAggression.process**, **NpcAggression.scanPlayers**, **WorldLocator.findViewablePlayers**, **WorldLocator.find**, **ChunkManager.load**, and several **ConcurrentHashMap** lookup methods. That pattern strongly suggests the server is repeatedly walking nearby chunks and building view results even when very little useful state is changing.

Separate from that, a single ForkJoin worker showed very high allocation volume almost entirely in **WalkingNavigator.findPath**, **AStarPathfinder.find**, **CollisionManager.traversable**, and other collision/path support code. That is a second red flag: something is repeatedly asking for paths even though the server was only left online with a logged-in player.

Key evidence from the recording

Category	Observation	Why it matters
CPU	Average JVM CPU only about 0.68% total (0.42% user + 0.26% system).	The server is not currently limited by raw compute at idle.
GC	Total pause time roughly 10.1 seconds over ~13h 5m, with about a 3.42 ms median pause.	Garbage collection is active but not currently disruptive.
Heap	Young collections are frequent, but the after-GC heap range stays broadly stable.	This pattern looks more like churn than a retained-object leak.
Allocation	Approximately 2.4 TB allocated in TLABs overall, or roughly 50.9 MB/sec.	Idle work is creating far more temporary garbage than expected.
Hot thread	GameService produced the majority of allocation volume.	Main tick processing is doing repetitive, short-lived work.
Worker thread	One ForkJoin worker allocated roughly 964.5 GB mostly in pathfinding and collision code.	Async movement or repathing is probably being triggered too often.

Why this matters for Luna

The current behavior is "wasteful but survivable" at low load. That is often the most dangerous kind of performance issue because it hides in testing and only becomes painful later, once real gameplay features stack on top of it. With more active players, more NPCs, more bots, more path requests, or denser areas, the existing patterns are likely to amplify CPU use, increase map contention, and push GC activity up.

Most likely problem areas

1. Aggression scanning

The profile strongly suggests aggressive NPCs are scanning for nearby players too often, likely every game cycle or close to it. That turns idle time into repeated chunk traversal and map lookup work.

2. Hot-path chunk loading/lookups

Methods such as ChunkManager.load, Chunk.translate, equals/hashCode, and ConcurrentHashMap lookup/tree-node logic are prominent in the hot path. That usually means read paths are still paying object-allocation and hashing costs that should be avoided.

3. Repeated async pathfinding

The background worker indicates that repathing is happening even while nothing meaningful should be changing. That often points to chase/follow logic, path invalidation, or movement requests being reissued too aggressively.

4. Allocation-heavy path and collision code

The pathfinding stack includes Position construction/translation, collision matrix indexing, and node creation. Even if each piece is individually cheap, together they create a large amount of avoidable garbage.

5. Secondary connection-pool churn

There are some signs of connection-adder / connection-closer activity in the SQL pool. This does not look like the primary issue, but the pool may still be cycling connections more often than necessary.

Recommended fixes, in priority order

P1	Throttle or event-drive aggression scans instead of doing full nearby-player scans every tick.
Expected benefit	Largest likely idle-time reduction; directly attacks the hottest path.
P2	Split read-only chunk access from "load/create" access. Use a cheap get-if-loaded fast path for hot queries.
Expected benefit	Reduces object churn and map overhead in world and collision lookups.

P3	Debounce async path requests. Do not submit a new path if one is already in flight or the destination has not meaningfully changed.
Expected benefit	Cuts background worker churn and avoids duplicate pathfinding.

P4	Replace object-heavy chunk keys with packed primitive keys where practical.
Expected benefit	Improves lookup locality and reduces equals/hashCode overhead.

P5	Trim allocations inside path/collision code, especially Position construction, validation, and node creation.
Expected benefit	Further reduces garbage volume once the higher-level triggers are fixed.

P6	Review SQL pool settings such as maxLifetime, idleTimeout, and keepaliveTime.
Expected benefit	Useful cleanup, but secondary to the main world-scan issues.

Practical interpretation

- If you do nothing, the server will probably still appear "fine" under tiny loads, because the GC pauses are short and CPU is low.
- Once you add more bots, more aggressive NPCs, crowded chunks, or frequent movement, the current scan-and-allocate pattern is likely to scale poorly.
- This is exactly the kind of profile where fixing the top two causes can create an outsized improvement: less garbage, fewer map lookups, less worker churn, and better headroom everywhere else.

Suggested next code review targets

The highest-value files and methods to inspect next are: **NpcAggression.process**, **NpcAggression.scanPlayers**, **WorldLocator.findViewablePlayers**, **ChunkManager.load**, and **WalkingNavigator.findPath**. Those are the spots most likely to turn this high-level profiling result into concrete, line-by-line fixes.

Prepared from the uploaded idle-session Java Flight Recorder and distilled into a project note format for Luna.