

Simutrans Freelist Iteration

Performance Analysis

Skipping empty object-pool slots during sync_step

Author: victor_18993

Base revision: r12100 (git commit be2c53104)

Benchmark: 50,000 fast-forward steps, pak128.german heavy save

Document date: 2026-07-23

Contents

Contents	2
Executive summary	3
1 Profiling context	4
2 The original problem	4
3 Where the empty slots were.....	5
4 The technical change.....	5
5 Total performance result	6
6 Where the performance is gained	6
7 Contribution to the total gain	7
8 Why private_car does not improve	8
9 Correctness and determinism	9
10 Portability.....	9
11 Limitations.....	9
12 Conclusion	10
Resumen ejecutivo (español).....	11
Problema	11
Solución.....	11
Resultados.....	11
Dónde se gana y dónde no.....	11
Validación.....	11
Limitaciones	11

Executive summary

Simutrans keeps its transient game objects — city cars, pedestrians, smoke clouds, wildlife, road signs and a few others — in per-type memory pools (`freelist_iter_tpl`). Every simulation step, `sync_step` walks each pool. The original loop tested every slot of every memory chunk, including the free ones.

Across the freelist users measured in a congested `pak128.german` save, about 89% of the tested slots were empty. In the sparse, high-churn pools this means most of the per-step work was simply checking empty slots. The patch replaces the scan-every-slot loop with a next-set-bit scan that jumps straight from one live object to the next. Same objects, same ascending order, same behaviour — only the empty-slot testing is removed.

Over 50,000 fast-forward steps (6 runs per version), the mean total step time drops from 791 ms to 729 ms, and the two measured ranges do not overlap (baseline minimum 776 ms is above the patched maximum 750 ms). The clearest single win is `movingobj`, whose cost falls from about 62 to about 4 million cycles. The dominant subsystem, `private_car`, is essentially unchanged, because its pool is densely occupied and has no empty slots to skip. No simulation rule is modified.

Measured result: ~7.8% lower total simulation time in this workload.

1 Profiling context

The optimisation was not guessed from reading the code; it came from profiling. A large, congested network was built (512×512 map, 40 cities, 6 AI players, about 24 simulated years) and saved, then re-loaded and profiled deterministically with lightweight per-phase cycle timers (rdtsc) around each part of the step. The resulting picture of where simulation CPU goes is shown in Figure 1.

The single largest cost is `private_car` movement, roughly half of the step. That cost, however, is genuine per-vehicle work (driving, tile-access checks, congestion) — not a mass scan over empty slots — so it is not what this patch targets. The opportunity this patch exploits is the wasted iteration in the sparse pools further down the list.

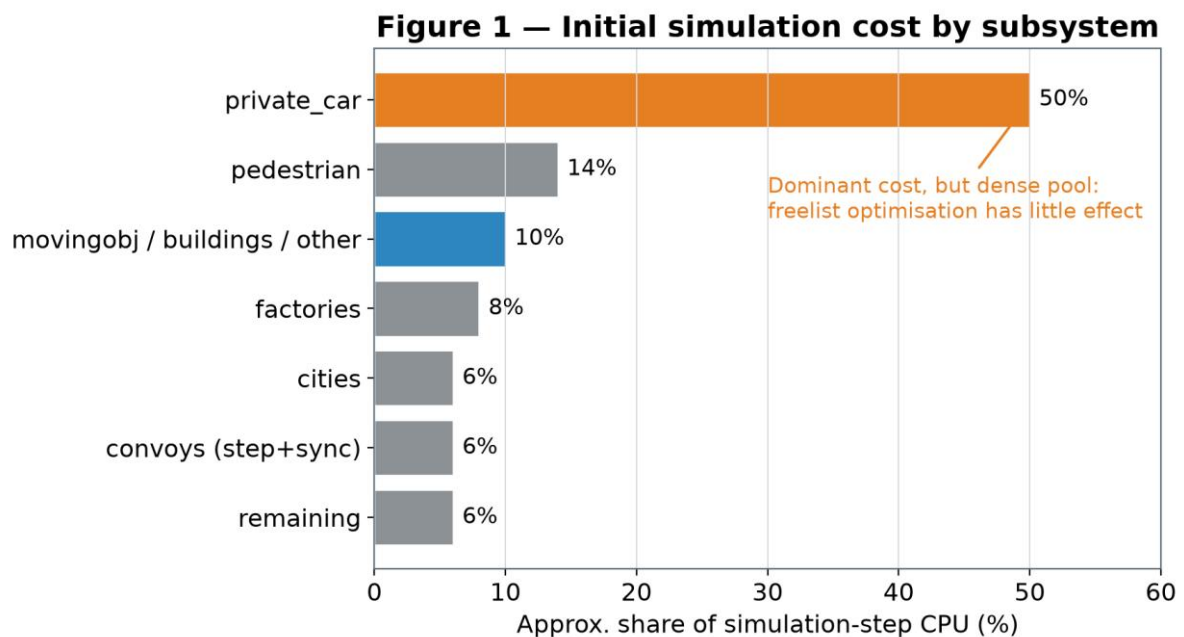


Figure 1. Initial simulation cost by subsystem (fast-forward, headless). `private_car` dominates but is dense work; the `freelist` optimisation targets the smaller, sparse pools instead. Shares are approximate.

2 The original problem

Each `freelist_iter_tpl<T>` is a static memory pool for one object type. It is a linked list of fixed-size chunks; every chunk carries an occupancy bitmap plus a block of object slots. A slot is marked live when an object is placed in it and cleared when the object is removed; a whole chunk is only released when the pool becomes completely empty.

Because objects such as smoke clouds or wildlife are created in bursts and then expire, chunks stay allocated long after most of their objects are gone. The original `sync_step` visited index 0, 1, 2, 3 ... and tested each one, doing real work only on the few that were still live. On a mostly-empty chunk, almost all of that testing is wasted.

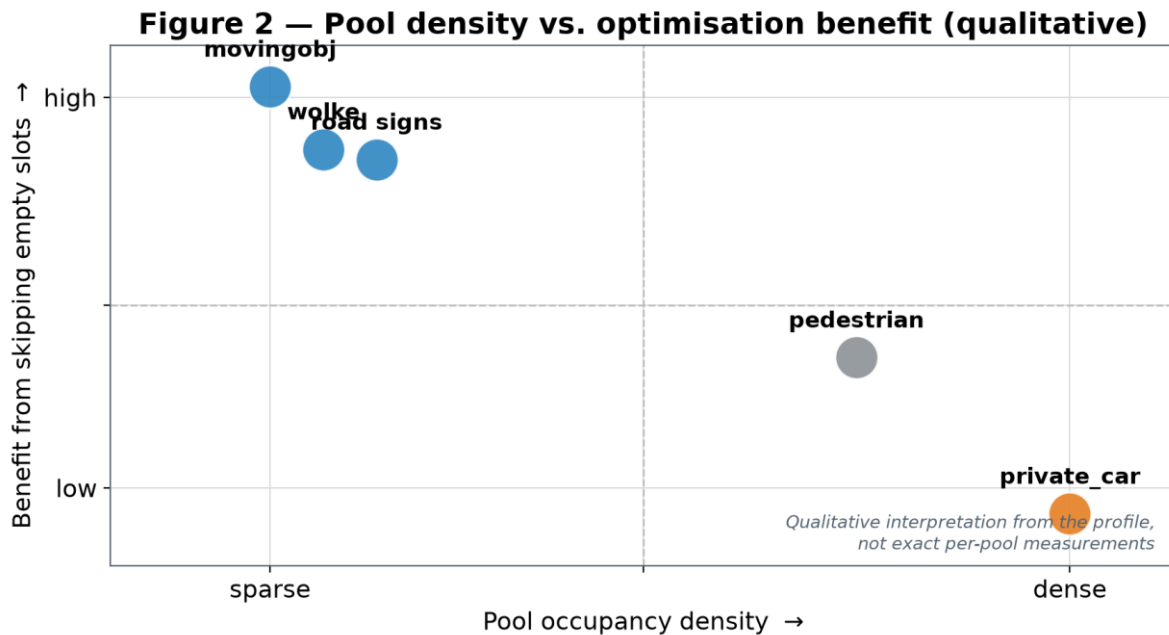


Figure 2. How much a pool benefits from skipping empty slots depends on how empty it is. Dense pools (*private_car*) gain little; sparse, high-churn pools (*movingobj*, *wolke*, *road signs*) gain the most. Qualitative interpretation from the profile.

3 Where the empty slots were

The 89% figure is an aggregate over all the freelist users that were measured. It does not mean every pool is 89% empty. The A/B test makes the distinction clear: the expensive *private_car* and the bulk of *pedestrian* sit in densely occupied chunks, whereas the emptiness — and therefore the wasted work — is concentrated in the sparse, high-churn pools.

Dense pools (little to skip): *private_car*, and the main body of *pedestrian*.

Sparse, high-churn pools (most of the waste): *movingobj* (wildlife), *wolke* (smoke), *road signs*, other small auxiliary objects, and part of *pedestrian*.

4 The technical change

The old loop stepped through every index and tested the occupancy bit. The new loop asks the bitmap directly for the first live slot and then for the next one, in ascending order, reading the live bitmap on each advance. If the current object removes itself mid-pass, the just-cleared bit is simply skipped, so the behaviour is identical to the old test-every-index walk. The early exit when the pool becomes empty is kept unchanged.

Before:

```
for every slot:
    if slot is active:
        sync object
```

After:

```

index = first active slot
while index exists:
    sync object
    index = next active slot

```

std::bitset offers no portable way to find the next set bit (the fast helpers are libstdc++-only), so it is replaced by a small plain-data bitmap of the same layout with a portable trailing-zero count: __builtin_ctzll on GCC/Clang, _BitScanForward64 on 64-bit MSVC, and a two-halves _BitScanForward on 32-bit MSVC. The count is never invoked with a zero word.

5 Total performance result

Figure 3 shows the headline measurement: total time for 50,000 fast-forward steps, six runs per version. The bars are the means; the whiskers show the full measured range. The patched maximum (750 ms) is below the baseline minimum (776 ms), so the two distributions do not overlap.

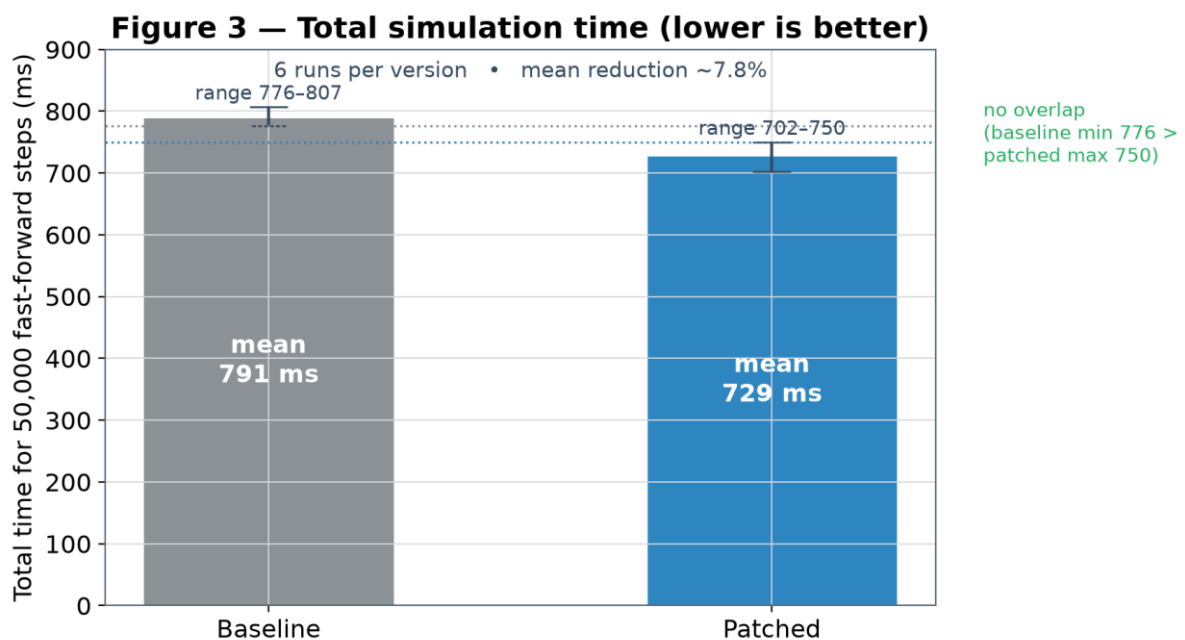


Figure 3. Total simulation time, 6 runs per version. Mean 791 → 729 ms (~7.8%). The measured ranges do not overlap. The y-axis starts at zero.

6 Where the performance is gained

Breaking the step down by subsystem (Figure 4A) shows movingobj almost vanishing while private_car barely moves. Because movingobj is tiny next to private_car on a shared axis, Figure 4B zooms in on the sparse-pool subsystems.

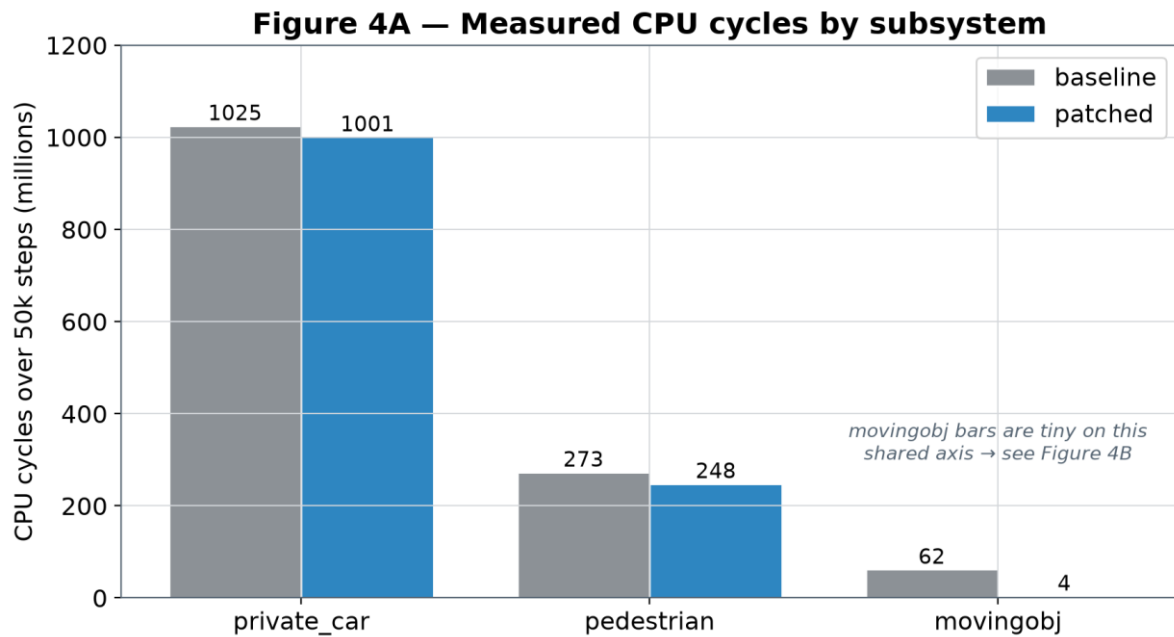


Figure 4. Measured CPU cycles by subsystem (millions, 50k steps). *private_car* is essentially unchanged; *movingobj* collapses to near zero.

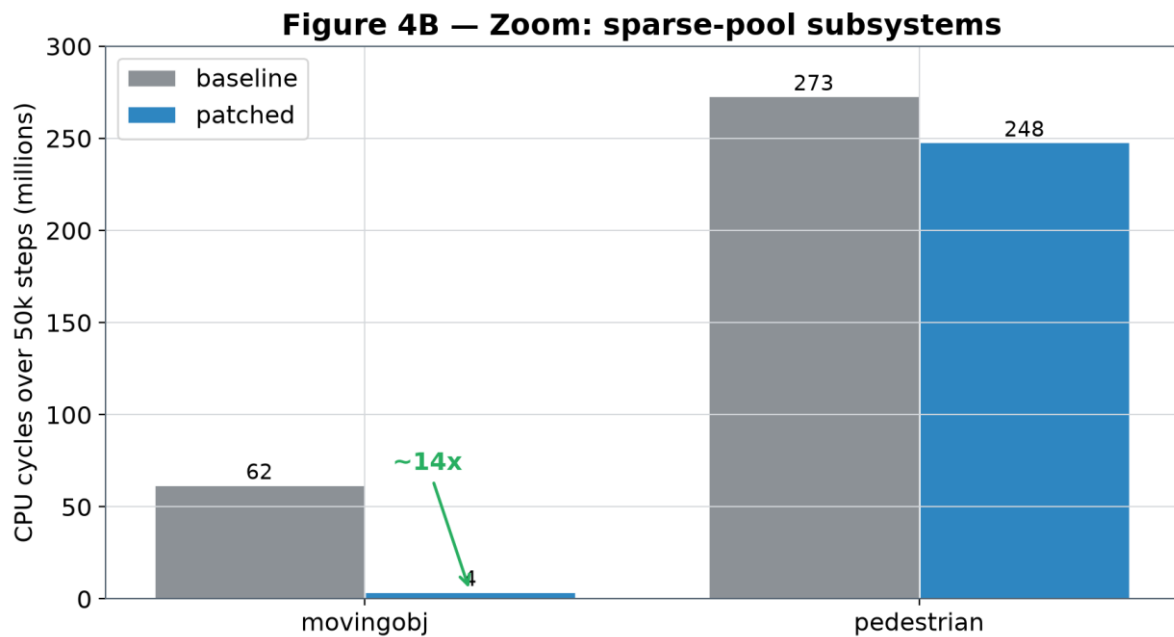


Figure 5. Zoom on the sparse-pool subsystems. *movingobj* drops about 14× (62 → 4); *pedestrian* also drops modestly (273 → 248).

7 Contribution to the total gain

The mean total saving is $791 - 729 = 62$ ms. *movingobj* gives the clearest directly-measured reduction; *pedestrian* adds a smaller measured reduction; and other shared-freelist users (smoke, road signs) that were not timed separately also contribute. *private_car* does not explain the improvement.

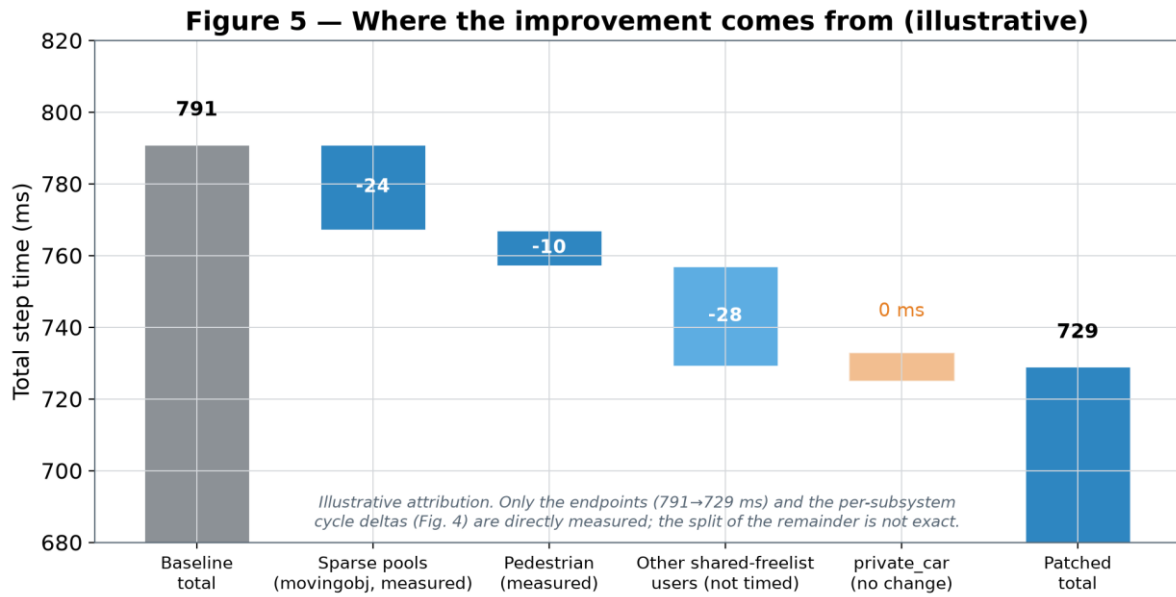


Figure 6. Illustrative attribution of the 62 ms saving. Only the endpoints (791 → 729 ms) and the per-subsystem cycle deltas (Figure 4) are directly measured; the split of the remainder is not exact. The y-axis starts at 680 ms, not zero.

8 Why private_car does not improve

It would be wrong to read this patch as a traffic-simulation speed-up. `private_car` is still about half of the step, and its pool is densely occupied: almost every slot the loop visits holds a live car, so there are few empty slots to skip. Skipping empty slots therefore cannot reduce the real per-car work — driving, tile-access and congestion checks, movement. Making that cheaper would require deeper, higher-risk changes and is out of scope here.

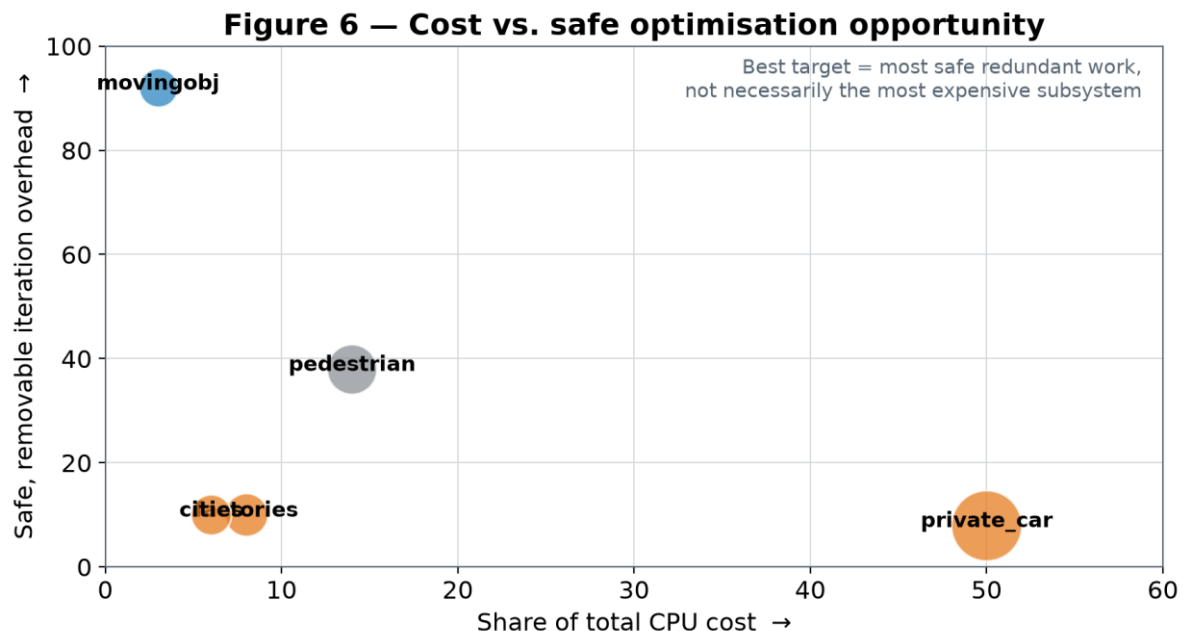


Figure 7. Cost versus safe, removable iteration overhead. The best optimisation target is not the most expensive subsystem but the one with the most safe, redundant work — here, the sparse pools rather than `private_car`. Qualitative.

9 Correctness and determinism

The new scan was validated two ways. First, a standalone test compares the index sequence produced by `find_first/find_next` against the plain scan-every-index sequence, for bitmap sizes $N \in \{1, 63, 64, 65, 127, 128, 161, 200, 256\}$ over thousands of random patterns, the explicit edge cases (first/last slot, the 63/64 cross-word boundary), and a mutation-during-iteration case (self-removal plus flips of later bits). All assertions pass (“ALL FREELIST BITMAP EQUIVALENCE TESTS PASSED”).

Second, the full game was run on the same save for 50,000 steps: route calculations, the end month and the observed convoy behaviour were the same before and after. In the tests performed the iteration order and the set of processed objects are unchanged, and the savegame format is not modified.

Check	Baseline	Patched	Result
Equivalence test (index sequence)	reference scan	<code>find_first/find_next</code>	identical — all pass
<code>route_calls</code> over 50k steps	846–847	846–847	identical
Final simulated month	23455	23455	identical
Total step time (mean)	791 ms	729 ms	~7.8% lower
Measured range overlap	776–807 ms	702–750 ms	no overlap
Savegame format	unchanged	unchanged	unchanged

Table 1. Validation summary. Behaviour-preserving in the tests performed; the time improvement is clean (non-overlapping ranges).

10 Portability

The bit scan is implemented with a small guarded helper; no `libstdc++`-internal APIs are used.

Platform / compiler	Bit-scan primitive
GCC (all targets)	<code>__builtin_ctzll</code>
Clang (<code>libstdc++</code> / <code>libc++</code>)	<code>__builtin_ctzll</code>
MSVC 64-bit (x64 / ARM64)	<code>_BitScanForward64</code>
MSVC 32-bit	<code>_BitScanForward</code> on the low then high 32-bit half
Other / fallback	portable guarded path (count never called with zero)

Table 2. Portable trailing-zero count by toolchain.

- No private `libstdc++` / STL-internal APIs.
- The trailing-zero count is never called with a zero word.
- No dependence on endianness (bit-index arithmetic only); shift amounts are always 0–63.
- No additional dynamic allocation; the bitmap uses `ceil(N/64)` 64-bit words.

11 Limitations

- One machine, one heavy save, one workload.

- Fast-forward / headless simulation time; in windowed play rendering can dominate the frame.
- The ~7.8–8% figure is specific to this scenario and should not be taken as universal.
- The patch does not optimise `private_car` directly, and does not modify `do_drive` or `can_enter_tile`.

12 Conclusion

- Profiling found objectively wasted work: iterating mostly-empty pool slots every step.
- Zero-impact hypotheses were ruled out by A/B measurement, not assumed.
- The final change is small and portable, and preserves object set and order in the tests performed.
- The gain is concentrated in the sparse pools; `private_car` remains the main hotspot and is untouched.
- The total benchmark improves cleanly, with non-overlapping ranges.

Small general change, large local reduction, clean ~8% workload improvement.

Resumen ejecutivo (español)

Problema

En cada step de simulación, Simutrans recorre sus pools de objetos transitorios (coches de ciudad, peatones, humo, fauna, señales) mediante `freelist_iter_tpl`. El bucle original comprobaba todos los slots de cada chunk, incluidos los libres. En el conjunto de pools medidos, cerca del 89% de los slots comprobados estaban vacíos, sobre todo en las pools dispersas y de mucha rotación.

Solución

El parche salta directamente de un objeto vivo al siguiente en el bitmap de ocupación, en orden ascendente y leyendo el bitmap vivo en cada avance. Mismos objetos, mismo orden, mismo comportamiento. Se sustituye `std::bitset` por un bitmap portable con conteo de ceros finales (`ctz` en GCC/Clang, `_BitScanForward` en MSVC).

Resultados

Sobre 50.000 steps en avance rápido (6 ejecuciones por versión), el tiempo medio total baja de 791 a 729 ms (~7,8%). Las distribuciones no se solapan: el mínimo del baseline (776 ms) está por encima del máximo parcheado (750 ms). `movingobj` baja unas 14 veces ($\approx 62 \rightarrow 4$ Mcyc).

Dónde se gana y dónde no

La ganancia se concentra en las pools dispersas (`movingobj`, humo, señales y parte de peatones). `private_car` — la mitad del coste — no mejora de forma significativa porque sus chunks están densamente ocupados: casi todos los slots contienen coches vivos y no hay vacíos que saltar. El parche no toca `do_drive` ni `can_enter_tile`.

Validación

Un test independiente comprueba que la nueva iteración produce la misma secuencia de índices que el escaneo plano en múltiples tamaños, casos límite y mutación durante la pasada. En el juego, `route_calls` (846–847), el mes final y el comportamiento de convoyes son idénticos. No cambia el formato de guardado.

Limitaciones

Un solo equipo, un solo save pesado, escenario fast-forward/headless. En modo ventana el render puede dominar. El ~8% es propio de este escenario, no universal.