RAMTERA TECHNICAL WHITE PAPER SERIES

WP-02

Finding the Real Bottleneck

Profiling-led optimisation of a physical verification engine

Two rounds of profiling took a full design rule check from 4,038 seconds to 120. Neither bottleneck was where we expected it.

4,038 s	256 s	120 s	33.6×
STARTING RUNTIME	AFTER ROUND ONE	AFTER ROUND TWO	TOTAL REDUCTION

Document	WP-02 — Finding the Real Bottleneck
Author	Sameer Pawanekar, Founder & Chief Executive Officer, Ramtera Inc.
Revision	v1.0
Date	August 2026
Classification	Public

ABSTRACT

Design rule checking is geometry-bound, and the instinct when it runs slowly is to reach for a better algorithm. On our engine that instinct would have been wrong twice. Two rounds of measurement-led optimisation reduced a full DRC run on a one-million-polygon design from 4,038 seconds to 120 — a 33.6× reduction — and in both rounds the dominant cost was somewhere no one had proposed looking. This paper describes the instrumentation, what it found, and the two structural fixes that followed.

01 The starting position

A full **density** design rule check on **daccore**, a mixed-signal design of approximately one million polygons, took 4,038 seconds on sixteen cores. Every measurement in this paper uses that same design and the same rule deck, so that the numbers compare cleanly.

Before any of the work described here, several plausible explanations were on the table: the polygon library, the threading model, memory allocation pressure, the rule deck itself. All were plausible. None turned out to be the answer. The engine had no shortage of theories about why it was slow; what it lacked was a measurement.

> **WHY MEASUREMENT COMES FIRST**
> The most expensive mistake available in performance work is optimising something that is not the bottleneck. It costs engineering time, it adds risk to correct code, and it produces no improvement — which then invites a second round of the same mistake somewhere else.

02 Instrumentation

A design rule engine executes a large number of small geometric operations, dispatched by rule. Sampling profilers report where time is spent in terms of functions, which is useful but does not answer the question that matters operationally: *which rule constructs are expensive?*

We therefore instrumented at the level of the evaluator rather than the function. Each geometric primitive accumulates two counters — total microseconds and invocation count — keyed by primitive. A timing wrapper around the pattern-execution entry point attributes every dispatch to its primitive, and a dump function emits the accumulated table at the end of the run.

One structural detail is worth naming, because it is the part that took the longest to get right. The evaluator cannot see the worker container's state, so the dump function has to live in the layer that owns the database instance rather than in the evaluator itself. Getting the ownership wrong produces either a compile failure or, worse, per-worker tables that silently do not aggregate.

> **WHAT THE INSTRUMENTATION PRODUCES**
> The output is a ranked table of primitives by total time, with invocation counts alongside. Total time identifies the target; the count tells you whether the fix is to make each call cheaper or to make fewer calls. Those are different fixes, and the ranking alone does not distinguish them.

03 Round one: the quadratic edge loop

The first profile pointed at a spacing-check family whose cost was dominated not by the geometric test itself but by the enumeration preceding it. The implementation compared every edge against every other edge in the candidate set — an all-pairs loop, quadratic in edge count.

On small cells this is invisible. On a design where a single layer contributes tens of thousands of edges, it is the entire runtime. The geometric predicate was efficient; it was simply being asked an enormous number of questions to which the answer was obviously no.

The fix

The remedy is standard and dull: index the candidate set spatially and query only within the constraint's reach, rather than enumerating all pairs and rejecting most of them. The predicate is unchanged. What changes is how many times it is called.

> **THE CORRECTNESS CONSTRAINT ON A PERFORMANCE FIX**
> The reach used for the spatial query must be derived from the rule's own constraint. Set it too tight and violations are missed — a correctness failure that a performance change should never be able to introduce. The regression test for this work is not a timing test: it is that the violation count is bit-identical before and after.

Alongside this, two backend choices were revisited. Some boolean operations had been routed through a general-purpose clipping library where the engine's own scanline operators were both applicable and faster. Where the geometry permitted it, those were switched back.

Round one change	Nature of the change
Spatial indexing of candidate sets in spacing checks	Eliminates an $O(K^2)$ pair enumeration; predicate unchanged
Boolean operator backend selection	Route operations to the scanline operators where the geometry permits, rather than to the general clipping path
Compiler configuration	Release builds at `-O3`, with profiling instrumentation confined to a separate build configuration

Table 1 — Changes in the first optimisation round.

Configuration	Runtime	Reduction from baseline
Baseline	4,038 s	—
After round one	**256 s**	**15.7×**

Table 2 — Round one result. daccore density check, ~1M polygons, 16 cores, identical rule deck.

04 Round two: profiling the faster engine

A fifteenfold improvement is the point at which most teams stop. Re-running the profiler on the faster engine is what produced the second half of this paper, and the result was more surprising than the first.

> **WHAT THE SECOND PROFILE FOUND**
> A single primitive — the boundary-edge orientation query used by edge-based rules — accounted for **435 seconds of 523 seconds** of measured evaluator time, across 2,432 invocations averaging 179 milliseconds each. Nothing else was close.

This had been completely masked in the first profile. When the quadratic loop dominated everything, a primitive costing a few hundred milliseconds per call was noise. Removing the larger cost did not create the second bottleneck; it revealed one that had been there all along.

Why the primitive was slow

The cost was not in the geometry. The implementation collected boundary segments and corners into two ordered sets, then discarded the corner sets entirely, then deduplicated the segments through a third ordered set. Three tree-based containers were being constructed and destroyed per invocation, one of them for a result that was never used.

This is an allocation problem wearing a geometry problem's clothing. No spatial index helps, because there is no search to accelerate. The work being done is real; it is simply the wrong work.

The fix

The rule construct in question asks for boundary edges of a given orientation. That question can be answered directly from an edge-extraction primitive that already existed, returning edges filtered by length and type, without materialising or deduplicating a corner set at all. The handler was rerouted to it.

One implementation note: edges returned as zero-width geometry vanish when inserted into a polygon set. They must be returned as thin bands rather than degenerate segments, or the result is empty and correct-looking.

Configuration	Runtime	Reduction from previous	Cumulative reduction
Baseline	4,038 s	—	—
After round one	256 s	15.7×	15.7×
After round two	**120 s**	**2.1×**	**33.6×**

Table 3 — Cumulative result. daccore density check, ~1M polygons, 16 cores, identical rule deck throughout. Violation counts identical across all three configurations.

05 What generalises

Profile again after every win

The second bottleneck was worth a further 2.1× and would never have been found by inspection, because it was invisible until the first was fixed. A large speedup does not mean the work is done; it means the profile has changed and needs re-reading.

Ranking by total time is not enough

Total time identifies the target. The invocation count tells you which kind of fix applies. Round one was a high-count problem and the fix was to make fewer calls. Round two was a high-cost-per-call problem and the fix was to make each call cheaper. Applying the round-one remedy to the round-two problem would have achieved nothing, and vice versa.

Not every bottleneck in a geometry engine is geometric

The single largest remaining cost in a physical verification engine turned out to be container construction, including for a result that was discarded. Domain intuition points at the geometry because the geometry is what the tool is for. The profiler has no such intuition, which is the reason to trust it.

Correctness is the acceptance criterion, not speed

Every change described here was accepted on the basis that the violation count was unchanged, and rejected otherwise. A performance change that alters results is not a performance change; it is a regression that happens to run quickly.

> **KEEP THE INSTRUMENTATION**
> The instrumentation is not scaffolding to be removed. It remains in the engine, behind a build configuration, because the profile changes every time the code does — and because the next bottleneck is already there, currently masked by whatever is slowest today.

Appendix A — Measurement environment

All measurements in this paper were produced on the single reference workstation specified below, using the same design and the same rule deck throughout.

Component	Specification
Processor	Intel Core i7-13700K (13th generation, Raptor Lake)
Cores / threads	16 physical (8 performance + 8 efficient) / 24 logical
Clock	800 MHz minimum, 5,400 MHz maximum
Cache	L1d 640 KiB, L1i 768 KiB, L2 24 MiB, L3 30 MiB shared
NUMA	Single node, all 24 logical CPUs
Memory	64 GB DDR5-4800 (2 × 32 GB)
Swap	1 GB
Storage	Kingston SNV2S500G NVMe SSD; all design data and scratch on NVMe
Operating system	Ubuntu 22.04.3 LTS (jammy), kernel 6.8.0-136-generic
Compiler	GCC 14.0.0 20231104 (experimental snapshot), `-O3`
Threads used	16
Design	daccore, approximately 1,000,000 polygons
Check type	**Density** design rule check
Rule deck	Identical across all measurements

Table A1 — Measurement environment.

CAVEAT — CORE TOPOLOGY

Hybrid core topology. The 8 performance + 8 efficient core split means thread-count scaling is not linear even for perfectly parallel work. All runtimes here are at sixteen threads and are directly comparable to each other; they should not be extrapolated to other thread counts.

CAVEAT — MEMORY

Memory headroom removes paging as a variable. 64 GB of RAM against 1 GB of swap means any workload exceeding physical memory is terminated rather than slowed. These measurements are compute-bound, never paging-bound.

Round-two profiler figures (435 s of 523 s across 2,432 invocations) are evaluator-attributed times from an instrumented build and are not directly comparable to the wall-clock figures in Tables 2 and 3, which are from release builds without instrumentation.

WHICH CHECK CLASS THESE FIGURES DESCRIBE

All figures in this paper are for the **density** design rule check on daccore, which is the longer-running of the two check classes on this design. The non-density check on the same design and the same core count completes in 52 seconds and is the subject of WP-11 on parallel scaling. The two should not be compared directly: they execute different rules over different derived layers.

About Ramtera

Ramtera Inc. develops **LVR**, a physical verification engine that executes foundry SVRF rule decks natively, covering design rule checking, layout versus schematic, electrical rule checking, layout XOR, density and fill, and parasitic extraction. LVR is written in modern C++ with a Qt6 interface, and has been validated against publicly available process design kits including SKY130 and IHP SG13G2.

Ramtera also develops **PRE**, a place-and-route engine sharing LVR's geometry kernel, and maintains a portfolio of granted patents, registered copyrights and trademarks across its verification and implementation technologies.

Ramtera Inc. is incorporated in the United States, with research and development conducted by Ramtera Design Automation India Private Limited in Bangalore, India.

Contact	
Web	https://ramtera.com
General enquiries	info@ramtera.com
Technical support	support@ramtera.com
Series	Ramtera Technical White Paper Series
This document	WP-02, revision v1.0, August 2026