RAMTERA TECHNICAL WHITE PAPER SERIES

WP-01

Compiling a 1.85-Million-Line Grammar

Build-time engineering for very large generated parsers

How a physical verification engine supporting eight process nodes in a single binary keeps its build under five minutes.

1.85M	~12,000	8	10×
GRAMMAR LINES	RULE CONSTRUCTS	PROCESS NODES	FASTER FULL BUILD

Document	WP-01 — Compiling a 1.85-Million-Line Grammar
Author	Sameer Pawanekar, Founder & Chief Executive Officer, Ramtera Inc.
Revision	v1.0
Date	August 2026
Classification	Public

ABSTRACT

Physical verification tools must parse foundry rule decks written in SVRF. Supporting many process nodes in a single binary produces a grammar far larger than anything a conventional compiler front end deals with — in our case a generated Bison grammar of approximately 1.85 million lines covering roughly 12,000 distinct rule constructs across eight process nodes. At that scale the parser stops being a parsing problem and becomes a build problem. This paper describes the failure mode, how to diagnose it, the structural remedy, and the measurements. Nothing here is specific to SVRF or to electronic design automation: it applies to any project in which a code generator emits a very large single function.

01 Why a rule parser gets this large

An SVRF rule deck is not a small configuration file. A modern foundry deck describes layer derivations, geometric constraints, connectivity, device recognition, density and fill requirements, multi-patterning colouring, and DFM checks.

A single advanced-node deck runs to tens of thousands of statements, and the statement grammar itself is broad: the same logical check may be expressed with a dozen legal operand orderings and optional modifier keywords, each of which the parser must accept.

Supporting eight nodes in one binary, the union of distinct constructs reaches roughly 12,000. The eight span four foundry generations, from mature 350 nm and 180 nm CMOS through advanced FD-SOI and FinFET processes, across multiple foundries. Individual nodes are not named in this paper. The resulting Bison grammar file is approximately 1.85 million lines.

The generational range matters to the argument, because the line count is otherwise easy to dismiss as padding. It is not. The breadth comes from genuinely different rule vocabularies across four foundry generations, not from repetition of the same constructs. A 350 nm automotive process and an advanced FinFET process share the basic geometric vocabulary and diverge almost everywhere else.

Two clarifications, because this number tends to be misread

> **WHAT 1.85 MILLION LINES DOES AND DOES NOT MEAN**
>
> **It is generated, not hand-written.** The grammar file is mechanical output. It is not maintained by hand, and its line count is not a proxy for human effort.
>
> **It is not shipped.** The parser is compiled into the tool binary and executes against whatever deck the customer supplies at runtime. Line count is a build-system concern, not a distribution or licensing one.

Both points matter for this paper. The size is a fact of the build pipeline, and the only thing that needs solving is how to compile it in a time that permits normal engineering.

02 The failure mode

Bison emits a C or C++ source file containing a single parsing function — conventionally `yyparse()`. Every semantic action written in the grammar becomes one arm of one very large `switch` statement inside that function.

For a small grammar this is not merely acceptable, it is efficient: the parser is one tight function with good locality and no call overhead on the hot path. For a grammar with roughly 12,000 productions carrying actions, it means a single function containing on the order of 12,000 basic blocks.

This is where modern optimising compilers behave badly. Several standard passes are **superlinear in the number of basic blocks within a single function**.

Compiler pass	Approximate cost behaviour	Effect at ~12,000 basic blocks
Dominator tree and dominance frontier construction	Near-quadratic in practice on dense control-flow graphs	Dominates early SSA construction
Global value numbering, partial redundancy elimination	Superlinear in blocks × values	Very large working set, heavy allocation

Compiler pass	Approximate cost behaviour	Effect at ~12,000 basic blocks
Register allocation (live-range splitting, interference graph)	Superlinear in live ranges	Expected to be the single largest contributor
Variable tracking for debug information (`-g`)	Superlinear in blocks × variables	Large, and pure overhead in a release build

Table 1 — Optimisation passes whose cost is superlinear in per-function basic-block count.

> **THE PASSES ARE NOT THE PROBLEM**
> None of these passes is defective. They are designed for functions of a few hundred basic blocks, and they are being handed one function two orders of magnitude larger.

There is a second, compounding effect. A translation unit is the unit of compilation *and* the unit of parallelism. A build system can compile a hundred files across sixteen cores; it cannot compile one file across sixteen cores. With all semantic actions in one translation unit, fifteen cores sat idle while one core worked for three quarters of an hour.

Symptom checklist

If a build exhibits all of the following, this is likely the failure mode:

1. One object file accounts for the overwhelming majority of build wall time.
2. That file's compile time is grossly disproportionate to its line count relative to the rest of the project.
3. Compiler memory usage on that file is measured in gigabytes.
4. Adding cores to the build does not help.
5. Dropping from `-O2` or `-O3` to `-O0` produces a dramatic improvement — far more than it does for normal code.

03 Diagnosis

Guessing is expensive at forty-four minutes per iteration. Measure first.

Per-pass timing

GCC's `-ftime-report` and Clang's `-ftime-trace` attribute wall time to individual optimisation passes. This is the single most valuable measurement in the whole exercise, because it distinguishes *“the file is big”* from *“one function in the file is big.”* The remedy is completely different in each case.

```
# attribute time to individual optimisation passes
g++ -O3 -ftime-report -c svrf.tab.cc 2> timereport-gcc14.txt
g++-11 -O3 -ftime-report -c svrf.tab.cc 2> timereport-gcc11.txt
```

Peak memory

If resident set size on one translation unit approaches or exceeds available memory, the machine may be paging, in which case the reported time is measuring the storage device rather than the compiler. On our build machine this was ruled out: 64 GB of RAM against 1 GB of swap means a compile exceeding physical memory is terminated by the out-of-memory killer rather than slowed by paging. The stalled build was therefore compute-bound throughout.

```
/usr/bin/time -v g++ -O3 -c svrf.tab.cc 2>&1 | grep 'Maximum resident'
```

Flag audit

Before optimising anything, confirm what the build is actually using. We found the parser being compiled with profiling instrumentation (`-pg`) and full debug information (`-g`) in what was nominally a working build. `-pg` inserts instrumentation at every function entry; `-g` on a function with 12,000 basic blocks generates an enormous amount of variable-tracking work. Neither belonged there.

04 The refactor: externalising semantic actions

The structural fix follows directly from the diagnosis. If the problem is one enormous function, the remedy is to stop generating one enormous function.

Before — action bodies inline

```
statement : SPACING layer layer NUMBER
          { /* forty lines of construction, validation,
               layer lookup, error handling, and
               evaluator registration, inline */ }
```

All such bodies land inside `yyparse()`.

After — action bodies externalised

```
statement : SPACING layer layer NUMBER
          { $$ = act_spacing_2layer($2, $3, $4); }
```

`yyparse()` collapses to a dispatcher: roughly 12,000 case arms, each one a single call instruction. Its basic-block count is unchanged, but each block is now trivial, and the superlinear passes have almost nothing to chew on — no long live ranges, no large value graph, no per-block variable tracking.

The action functions themselves are distributed across **sixteen translation units**, grouped by construct family. The grouping is not important to correctness. What matters is that there are enough units to saturate the build machine's cores, and that no single unit is disproportionately large.

Three implementation notes that cost us time

NOTE 1 — HEADER DISCIPLINE

Header discipline determines whether this works. Sixteen translation units that each include the same heavy header set will multiply front-end parsing work by sixteen and can erase the gain entirely. Action files should include a narrow interface header declaring the action function signatures and the shared value types, and nothing more. Prefer forward declarations over includes wherever possible.

NOTE 2 — THE VALUE TYPE IS NOW AN INTERFACE

The grammar's semantic value type becomes a real API boundary. Once actions live outside the generated file, the value type is an interface between the generator's output and hand-maintained code. Changing it recompiles everything. Keep it small and stable.

NOTE 3 — LINK TIME SURFACES

Link time becomes visible. It was previously hidden behind a forty-four-minute compile. At sixteen units it is a measurable fraction of the build, and a faster linker such as `mold` or `lld` is worth evaluating.

05 Results

MEASUREMENT DISCIPLINE
All measurements below were produced on the reference workstation specified in Appendix A. Each row differs from its neighbour in exactly one variable, so that the effect of the compiler flags and the effect of the code restructuring can be read separately.

Configuration	Compiler flags	Full build wall time	Cores utilised
Single translation unit, inline actions original configuration	`-pg -g`	**44+ min** frequently indistinguishable from a hang	1
16 translation units, externalised actions	`-pg -g`	**4 min**	16
16 translation units, externalised actions current release configuration	`-O3`	**4.5 min**	16

Table 2 — Full clean-build wall time. Reference workstation, Appendix A. `make -j20` in all three cases.

Reading the table

Rows one and two hold the compiler flags constant at `-pg -g` and vary only the code structure. The full build falls from over forty-four minutes to four. That is the measurement this paper rests on, and it attributes the improvement to the restructuring rather than to any change in how the compiler was invoked.

Rows two and three hold the structure constant and vary the flags. The difference is about thirty seconds — the release build at `-O3` is marginally slower than the instrumented build, which is what one would expect, since `-O3` performs more optimisation work on code that is now cheap to optimise.

WHAT THE TWO COMPARISONS ESTABLISH
The flags were never the story. In the restructured build they account for roughly thirty seconds out of four and a half minutes. In the original build they were sitting on top of a structural problem that cost forty minutes, and removing them would not have solved it.

Parallelism is bounded by unit count, not by the job limit

All three builds were invoked with `make -j20`, but the refactored build saturates at sixteen concurrent compiler processes, because there are sixteen action translation units to compile. Raising the job limit further changes nothing. The unit count, not the job limit, is the parallelism parameter that matters — which is why it was chosen to match the sixteen physical cores on the build machine.

The original build utilised a single core regardless of the job limit, because a translation unit is the unit of parallelism and there was only one that mattered. Fifteen cores sat idle for the duration.

Runtime impact

Converting inline action bodies into function calls introduces call overhead on a path that executes once per grammar production during deck parsing. Deck parsing is a small fraction of total tool runtime, which is dominated by geometric processing across millions to billions of polygons. The refactor moves work from the compiler to the linker and adds one call per reduction at parse time; neither is material at the scale at which the tool operates.

Scale of the artefacts

Quantity	Value
Bison grammar source	~1,850,000 lines
Distinct SVRF constructs across eight process nodes	~12,000
Process nodes supported in one binary	8
Action translation units after the refactor	16
Build job limit	`-j20`
Concurrent compiler processes achieved	16

Table 3 — Scale of the artefacts described in this paper.

Limitations of these measurements

The pre-refactor configuration was measured at `-pg -g` only; a single-translation-unit build at `-O3` was not separately timed. The flag comparison in rows two and three of Table 2 is therefore measured in the restructured build and not in the original one. Figures were produced with the compiler and toolchain given in Appendix A; a different compiler version will produce different absolute numbers, though the structural effect described in Section 2 is a property of how optimising compilers handle very large functions and is not specific to one toolchain.

06 Secondary levers

Once the structural problem was solved, ordinary build hygiene became worth applying. Ranked by the return we observed:

Lever	Effect	Notes
Remove `-pg` and `-g` from release builds	**Large**	Profiling instrumentation and debug variable tracking are both superlinear contributors here. Use a separate profiling build configuration.
`make -j` **matched to physical cores**	**Large**	Only becomes available *after* the split. Meaningless beforehand.
`ccache`	**Large** on repeat builds	Especially effective given that the generated grammar file is byte-identical across most rebuilds.
Narrow, stable interface headers	**Moderate**	Prevents the split from being undone by include bloat.
Precompiled header for the shared interface	**Moderate**	Worth it when one narrow header is included by all sixteen units.
`-fno-var-tracking-assignments` **on debug builds**	**Moderate**	Retains usable debug information at a fraction of the cost on very large functions.
Faster linker (`mold`, `lld`)	**Small but free**	Matters more as unit count grows.

Table 6 — Secondary build-time levers, ranked by observed return.

07 What did not help

Reporting the dead ends is more useful than reporting only the wins.

Lowering the optimisation level

Dropping to `-O1` did shorten the build, but it is not a fix — it trades a build problem for a runtime problem, and for a geometry-heavy tool the runtime cost of unoptimised code is severe. The correct move is to make the code optimisable in reasonable time, not to stop optimising it.

Splitting the grammar file itself

Bison's output is one function by construction. Splitting the input grammar across multiple `.ypp` files does not split the generated parser. It is the *actions* that must move, not the grammar text.

Adding hardware

More cores do nothing for a single-translation-unit bottleneck. More memory helps only if the machine was paging — worth checking, but not a solution.

08 Reproducing this on your own build

The pattern generalises to any generator-emitted code containing a large dispatch function: protocol parsers, interpreter main loops, state machines, generated serialisers.

1. Identify the single slowest object file in a clean build. Time each compiler invocation, not just the total.
2. Run `-ftime-report` (GCC) or `-ftime-trace` (Clang) on it. Confirm the time is in per-function passes rather than in front-end parsing.
3. Audit the flags actually reaching that file. Remove instrumentation and debug flags from release configurations.
4. If the time is in per-function passes, externalise the function bodies into multiple translation units, leaving trivial call-only dispatch behind.
5. Choose the unit count to saturate your build parallelism, and keep the shared interface header narrow.
6. Re-measure, including link time and incremental rebuild time, not only the clean-build number.

09 Conclusion

A 1.85-million-line grammar sounds like an argument against supporting many process nodes in a single binary. It is not. The size is mechanical output, and the cost it imposes is a build-engineering cost with a known structural remedy.

Moving semantic actions out of the generated parser function and into sixteen parallel-compilable translation units took our full build from over forty-four minutes to four and a half — roughly a tenfold reduction — at no measurable runtime cost. That change is what makes multi-node support in one binary practical to develop against day to day, rather than merely possible in principle.

> **SCOPE**
> This paper describes build-system engineering only. It does not describe LVR's rule-handler generation, grammar synthesis, or verification algorithms.

Appendix A — Measurement environment

All build-time measurements in this paper were produced on the single reference workstation specified below. Ramtera white papers that report measurements from other environments — in particular distributed full-chip runs — specify those environments separately and do not cite this one.

Reference workstation

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), 2 of 4 DIMM slots populated
Swap	1 GB
Primary storage	Kingston SNV2S500G NVMe SSD, 465.8 GB
Secondary storage	Seagate ST2000DM005-2U9102 HDD, 1.8 TB (not in the measurement path)
Working filesystem	`/dev/nvme0n1p2` mounted at `/`, 457 GB, 67 GB free
Operating system	Ubuntu 22.04.3 LTS (jammy), kernel 6.8.0-136-generic

Table A1 — Reference workstation specification.

Toolchain

Tool	Version	Note
Compiler	GCC 14.0.0 20231104 (experimental)	Snapshot build, **not** the Ubuntu 22.04 system compiler, which is GCC 11.x. See the caveat below.
Parser generator	GNU Bison 3.8.2	—
Linker	GNU ld 2.38 (GNU Binutils for Ubuntu)	Default BFD linker; `mold` and `lld` not in use.
Build system	GNU Make, `-j20`	Saturates at 16 concurrent processes, bounded by the sixteen action translation units.
Optimisation	`-O3`	No `-pg`, no `-g` in the release configuration.

Table A2 — Toolchain.

Caveats

> **CAVEAT 1 — COMPILER PROVENANCE**
>
> **The compiler is a pre-release snapshot.** GCC 14.0.0 20231104 is not the Ubuntu 22.04 system compiler, which is GCC 11.x. Compile-time behaviour on very large functions differs between compiler versions, so absolute figures reproduced on stock tooling will differ. The structural effect described in Section 2 is not specific to one compiler.

CAVEAT 2 — STORAGE

Two storage tiers exist; only one is in the measurement path. All build trees, design data and scratch space live on the NVMe device. A reader running the same workload from spinning media will see substantially worse results on random access.

CAVEAT 3 — HYBRID CORES

The hybrid core topology affects scaling curves. The 8 performance + 8 efficient core split means thread-count scaling is not linear even for perfectly parallel work: threads 1–8 land on performance cores, 9–16 on efficient cores, and 17–24 on SMT siblings. A knee at eight threads is a property of the machine, not of the software.

CAVEAT 4 — 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. Measurements from this machine are compute-bound or I/O-bound, never paging-bound.

Measurement protocol

1. **Clean state.** Clean build directory, or freshly copied input data, before each timed run.
2. **Repetition.** Minimum three runs; best-of-N reported with N stated. If run-to-run spread exceeds five per cent, the spread is reported rather than a single figure.
3. **Wall clock via** `/usr/bin/time -v`, capturing elapsed time, peak resident set size, and CPU percentage together.
4. **Thread count stated explicitly** for every parallel measurement, together with whether it maps to performance cores, efficient cores, or SMT siblings.
5. **Input identified** by name, scale and file size; anonymised descriptors plus size metrics are used where data is subject to non-disclosure obligations.
6. **No mixed-configuration comparisons.** Two rows in the same table differ in exactly one variable. A row that changes both compiler flags and code structure proves nothing about either.

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-01, revision v1.0, August 2026