Optimizing the Lean and Mathlib toolchain

Dan Romik*

September 9, 2026

Abstract

Lean and Mathlib have recently emerged as widely used tools for formalization of proofs in mathematics research, but are still plagued by performance issues making their use somewhat cumbersome and resource-intensive, as discussed recently by the Lean user community [1, 2]. We performed benchmarking and an analysis of the Lean+Mathlib toolchain code in search of inefficiencies that can be eliminated, and developed a collection of fixes that improve Lean's performance in several notable ways. Our fork `lean4.33.1-optimized` reduces the time to load Mathlib on a laptop from 10.2 to 2.3 seconds and its memory cost from 5.7 to 1.4 GB; on a small cloud server, a cold load falls from 104 to 35 seconds. The total time a beginner spends waiting for Lean during an editing session falls from 23.7 to 11.3 seconds, and a full build of a 1,387-file project completes in 23.0 minutes instead of 47.7. Every change is verified to leave Lean's output byte-for-byte unchanged, including the axioms each theorem depends on and the order in which search tactics offer their suggestions. We also fixed an existing reported Lean bug [3], and resolved the fork-safety question that has been open since April 2025 [4, 5].

1 Introduction

1.1 Background: the Lean+Mathlib toolchain

This paper reports the results of a project to optimize the Lean and Mathlib toolchains to improve their performance. We start with some background. The Lean programming language was launched in 2013. Its current, considerably more mature, iteration, Lean 4, was released in 2021 and has since found widespread adoption by practitioners of proof formalization in mathematics. Development of Lean and the associated mathematics formalization library Mathlib is now overseen by the Lean Focused Research Organization [6] with the participation of a thriving community of volunteer developers. Starting around 2025, at least three AI companies, Math, Inc., Harmonic and Axiom Math, have adopted Lean as standard infrastructure powering their autoformalization models (respectively `Gauss`, `Aristotle` and `AxiomProver`), which have had notable successes in advancing mathematical research and formalizing complicated bodies of mathematical knowledge [7, 8, 9, 10, 11, 12, 13, 14]. In August 2026, the Palomar Registry was set up by leading mathematicians and Lean advocates as a repository of record for Lean-formalized mathematics. [15]. Even more recently, Anthropic's AI models have autoformalized the Fermat-Wiles theorem in Lean [16], and OpenAI included an AI-generated Lean-formalized proof in its announced solution to the Navier-Stokes Millenium Prize problem [17]. Thus, Lean is rapidly becoming an indispensable tool for mathematical research, verification, and knowledge dissemination.

*Department of Mathematics, UC Davis and Rhombic Research. email: `romik@ucdavis.edu`

The use of Lean for mathematics now appears to account for the overwhelming majority of all Lean usage, to the point where "Lean" is now often understood to be synonymous with the combination "Lean+Mathlib". We adopt this shorthand for the purposes of this paper.

The Lean toolchain consists of the following components:

- **`lean`.** The `lean` executable, which parses a source file, *elaborates* it into a fully explicit proof term, and checks the result with a small trusted core called the *kernel*. Lean programs import libraries with an `import` directive, so one of the first tasks `lean` performs, preceding the elaboration and checking, is to load the imported libraries: it opens the compiled files of every imported module and builds an in-memory picture of their contents. That step is where nearly all of the work reported here takes place [18].
- **`lake`.** This is Lean's build tool and package manager, which resolves dependencies, determines what needs rebuilding, and runs one `lean` process per source file [18].
- **Mathlib.** The community-developed library of formalized mathematics: roughly 8,300 modules. Counting its own dependencies and Lean's built-in library, a plain `import Mathlib` loads 10,498 modules in all, whose compiled form is 52,490 files totaling 7.5 GB [19].
- **Compiled `Mathlib`.** The compiled library files themselves (`.olean` and its companion parts), normally downloaded pre-built rather than compiled locally. Each module is stored as five separate files holding four kinds of thing: a public part, a private part, an editor-support part, and the compiled code, which occupies two of the five.
- **`vscode-lean4`.** The Lean extension for Visual Studio Code, through which most interactive coding work is done. It does not invoke `lean` directly but starts `lake serve`, which in turn starts one `lean` worker process per open file [20].
- **`leanprover-community/repl`.** A `lean` process running in REPL (read-evaluate-print-loop) mode, colloquially known as "the community REPL". This invocation style for Lean is the engine most AI proof-search infrastructure builds on [12, 21, 22].
- **`elan`**. A version manager for Lean that installs toolchain bundles and switches between them. It is the one component here that never runs Lean code, and we mention it only because reproducing our measurements requires knowing which bundle was active [23].

A key characteristic of the Lean ecosystem is that actual elaboration of Lean programs — the computational process at the heart of the toolchain — is carried out through several distinct invocation paths and workflows, by both humans and computers engaged in different Lean-usage scenarios. A research project aimed at optimizing Lean and reducing its resource usage therefore has multiple vectors available to attack; indeed, such a multi-vector approach is the one we have pursued in this project.

1.2 Resource usage issues

Users of Lean have reported usability issues affecting their workflow, particularly relating to slow run times and slow responsiveness in the interactive Lean coding environment, on the community's Zulip discussion board. A beginner working through *Mathematics in Lean* on laptop with 8 GB of RAM reports that "every time VSCode restarts, it takes forever", and is told in reply that "8 gigs is not enough in 2026 to have a comfortable Lean experience" [1]; a Mathlib maintainer, under the heading "Lean is unusably slow", reports having to force-quit the editor repeatedly, with system monitoring showing the machine swapping [2]. Several posts ask which computer to buy in order to use Lean at

all [24, 25, 26], with the ensuing discussions including suggestions that working with Lean requires at least 16 GB of RAM (and some people suggesting that 32 GB provides for a more comfortable usage experience). Our motivation for starting this project was that we experienced such issues ourselves. This author is the creator of the MathBB online mathematics platform [27]. While adding support for AI-assisted Lean proof formalization and code execution on MathBB, we noticed that even short Lean programs took 8-10 seconds to run on the dedicated Lean execution server provisioned within MathBB. We had also previously noticed the slowness of writing and interactively validating Lean code in VS Code. Our working hypothesis when starting the project was that the Lean toolchain includes opportunities to meaningfully improve Lean's performance, including in some ways that require a relatively modest effort. As we explain next, this turned out to be the case.

1.3 Our optimization results: a summary

The main contribution of this project is the development of a set of eight distinct improvements to Lean that deliver significant gains in performance across several areas of measurement and several types of Lean usage. We structured the improvements in two parts: six improvements are delivered in the form of `lean4.33.1-optimized`, a fork of the stable Lean release `leanprover/lean4:v4.33.1` (commit `819816b2`); and two separate improvements that live outside the Lean source code and therefore do not logically belong as part of the forked project, and are shipped separately.

Table 1 summarizes the performance gains for different scenarios.

	`lean4:v4.33.1`		with improvements		% change	
Situation	time	memory	time	memory	time	memory
Loading the library						
`import Mathlib`, warm (laptop)	10.2 s	5.7 GB	2.3 s	1.4 GB	(−77%)	(−75%)
`import Mathlib` + `exact?` (laptop)	15.1 s	7.9 GB	3.8 s	1.6 GB	(−75%)	(−80%)
`import Mathlib`, warm (container)	2.44 s	5.9 GB	1.41 s	1.6 GB	(−42%)	(−73%)
`import Mathlib`, warm (cloud server)	5.68 s	5.26 GB	3.99 s	1.75 GB	(−30%)	(−67%)
`import Mathlib`, cold (cloud server)	104.1 s		35.4 s		(−66%)	
`import Mathlib`, warm (x86-64, 8 GB)	3.44 s	6.27 GB	2.28 s	1.97 GB	(−34%)	(−69%)
`import Mathlib` + `exact?` (x86-64)	9.70 s		3.60 s		(−63%)	
Building a project						
1,387-file project from scratch (laptop)	47.7 min	68 GB	23.0 min	34 GB	(−52%)	(−50%)
Interactive use						
Beginner's session, total wait on Lean	23.7 s		11.3 s		(−52%)	
First `exact?` call in the editor	4.7 s	+2 GB	1.4 s	+0.2 GB	(−70%)	(−90%)
Editor worker peak memory (RSS)		8.0 GB		3.7 GB		(−54%)
Checking service (one file per process)						
One check, warm (cloud server)	5.8 s	6.0 GB	1.5 s	1.8 GB	(−74%)	(−70%)
Elaboration (proof checking proper)						
Tactic elaboration CPU, textbook suite	22.9 s		6.2 s		(−73%)	

Table 1: Main results; figures in (green) are the reduction relative to `lean4:v4.33.1`, the latest stable release. "Warm" means the compiled library is already in the operating system's page cache; "cold" means it is not. Memory figures depend on the machine's memory headroom as well as on the change itself, and are quoted with the machine they were taken on. Additional details of each measurement and the hardware used are given in Sections 2 and 3.

Points to note

1. **The reference Lean.** We used `lean4:v4.33.1`, the latest stable release of Lean, as the logical starting point for our optimization work. Also currently available is the release candidate `v4.34.0-rc2`, which includes updates to `lean4:v4.33.1` [28, 29]. The performance issues our fork addresses are not changed between `lean4:v4.33.1` and `v4.34.0-rc2`, so our improvements apply equally to the release candidate, and can be easily integrated with it through a rebase operation once it is shipped by the Lean developers. Throughout the paper, "unmodified Lean" means `lean4:v4.33.1` exactly as the Lean developers release it, with none of our changes applied; it is the baseline every measurement here is taken against, and the thing our fork, `lean4.33.1-optimized`, is compared with.

2. **The main guiding principle.** Our guiding principle in developing these improvements was that they should change nothing about Lean's observed behavior, other than the change in resource usage. In other words, any user deploying `lean4.33.1-optimized` should see byte-identical results in terms of the reported output; not merely whether the run succeeded, but the exact text of every message, the list of axioms each theorem depends on, and the exact list and order of suggestions offered by search tactics such as `exact?` and `rw?`. We were able to satisfy this principle for all the improvements, except for improvement 5 where a possible (minor) difference in behaviors may appear, and for that improvement as well the principle could be satisfied with a bit more work; see the discussion at the end of Section 3.5.

   Thus, with one notable exception, `lean4.33.1-optimized` has the sole observable effect of making Lean run faster.

3. **Difficulty of attributing memory.** One qualification should be stated in connection with Table 1. Memory is harder to attribute than time: when several `lean` processes run at once they share much of what each of them reports using, so the figures above cannot simply be added up.

1.4 The six improvements in the `lean4.33.1-optimized` fork

The `lean4.33.1-optimized` fork comprises six sets of changes, each constituting a distinct improvement; we summarize the idea behind each one below, and give a more detailed description in Section 3.

- **Improvement 1: address reservation (macOS only).** Lean maps its 52,490 compiled library files into memory one at a time, and macOS searches for a free address for each. We claim the whole range once, up front, so the search happens once instead of 52,490 times. This alone removes about six seconds from a warm `import Mathlib` on a Mac. This improvement is specific to macOS: on Linux the operating system does not have the problem, and the change is not needed.

- **Improvement 2: shell-first layout.** Each compiled file stores, for every definition, a small record and the much larger content it points at. Lean reads all the records at start-up and almost none of the content — but the two are interleaved, so reading the records drags in gigabytes of proofs alongside them. We changed how the files are *written* so that the records sit together at the front, resulting in the files containing the same information in the same format, but with a different order. This improvement requires the Mathlib library to be rebuilt in order to work.

- **Improvement 3: no-touch reference counts.** Lean increments a reference count on every object it looks at, including objects living inside memory-mapped library files whose counts are never read. Writing to them forces the operating system to give the process its own private copy of the page. We skip the increment for objects in that region — the same waste the layout change (improvement 2) attacks, approached from the runtime side rather than from how the files are written.

- **Improvement 4: lazy part loading.** Each module is stored as five separate files: a public part, a private part, an editor-support part, and the compiled code in two more. Lean opens all five for every module you import. We open the public part on import and fetch the others only when something asks for them.
- **Improvement 5: prebuilt search index.** The first `exact?` or `apply?` in a session builds an index over every imported constant, which costs several seconds and a great deal of memory. We compute each definition's contribution once, when its module is compiled, and store it alongside the module, so the index is assembled from stored parts rather than built from scratch.
- **Improvement 6: tactic index image.** Tactics such as `simp`, and the machinery that finds instances, consult lookup tables that Lean rebuilds at every start-up from data that has not changed: 93 tables in all for `import Mathlib`. We write the finished tables to a file once and map that file into memory on later runs, so the tables are installed rather than reconstructed.

Each of the six improvements stands on its own and targets a particular inefficiency in Lean's existing design, with the six improvements together combining to achieve the overall performance gains listed in Table 1. The value of what individual improvements, as well as certain subsets of the improvements grouped together, bring to the table is also worth examining, and this also highlights certain interactions between the different improvements that need to be understood properly in order for the overall design of the `lean4.33.1-optimized` fork to make sense. A convenient way to illustrate these measurements is with an *ablation table*, shown below, which shows the effect on performance when a particular improvement, or a particular pair of improvements, is *removed* relative to the reference situation where all improvements are enabled.

		`import Mathlib`		+ one `exact?`	
#	Improvement(s) ablated	time	memory	time	memory
1	address reservation (macOS only)	+0.16 s	−0.01 GB	+0.11 s	0.00 GB
2	shell-first `.olean` layout	+0.08 s	+0.13 GB	−0.07 s	+1.06 GB
3	no-touch reference counts	+0.13 s	+0.01 GB	+0.05 s	+0.01 GB
4	lazy part loading	+0.81 s	+0.93 GB	+0.26 s	+1.10 GB
5	prebuilt search index	+0.06 s	−0.02 GB	+4.66 s	+2.94 GB
6	tactic index image	+0.46 s	+0.25 GB	+0.39 s	+0.27 GB
2 + 3	layout and reference counts combined	+0.35 s	+1.39 GB	+0.42 s	+1.35 GB
4 + 5	lazy loading and search index combined	+0.82 s	+0.93 GB	+4.11 s	+3.87 GB

Table 2: An ablation table showing the difference in performance when comparing between a run with all improvements enabled and one with a particular improvement or pair of improvements disabled but the others kept on. Figures cited are medians of three-run trials on the laptop with warm cache.

Points to note

1. **Relationship between improvements 2 and 3.** The reason we included in the ablation table the effects of removing both improvements 2 and 3 together is that they target the same inefficiency: by design, the shell-first layout improvement is meant to supersede the no-touch reference counts improvement and make the latter obsolete. We nonetheless included both in the `lean4.33.1-optimized` fork, because the layout change only takes effect on a Mathlib library rebuilt from source, and so its benefits would not be available to Lean users immediately on installing `lean4.33.1-optimized`: essentially every Mathlib user downloads the precompiled library rather than building it. By contrast, the no-touch reference counts change requires no rebuild and works

with the library already on disk. So for a user today, the shell-first layout change does nothing unless they rebuild, while the reference-count change delivers an immediate benefit. For a user with a rebuilt library, the position reverses and the reference-count change is effectively redundant.

In practical terms, what this means is that in a hypothetical future version of Lean that includes improvement 2 and that users will be downloading alongside an updated, rebuilt Mathlib, there is no real benefit to including improvement 3 as well and it can be discarded.

2. **Relationship between improvements 4 and 5.** These two improvements have a different type of interaction, wherein the prebuilt search index improvement actually helps mitigate a negative effect on performance *that is caused by introducing the lazy part loading improvement*. Specifically, the search index is built by walking every constant in the import closure, and lazy part loading has by then deferred exactly the parts that walk needs, so the first `exact?` of a session has to fetch them all back — a cost that did not exist before improvement 4 and that improvement 5 removes by assembling the index from stored parts instead. So, while both improvements have useful functions that contribute to the overall performance gains obtained by `lean4.33.1-optimized`, the point of including their combination as a separate entry in the ablation table is that looking only at the effects of removing each one individually may give a misleading impression as to where exactly its usefulness is coming from and how much of it there is: e.g., as can be read from the table, removing improvement 5 by itself costs 4.66 s on the `exact?` row while removing improvement 4 by itself costs 0.26 s there, which reads as improvement 5 carrying that workload single-handedly. Removing the two together costs 4.11 s — *less* than removing improvement 5 alone — because part of improvement 5's individual figure is to eliminate a cost that is incurred by the addition of improvement 4.

3. **Other interactions.** In addition to the pair relations discussed above, there are other overlaps in coverage between the different improvements. For example, improvement 1 is worth far more in isolation — 10.2 s → 3.8 s against `lean4:v4.33.1` — than what the table records it as being worth in the context of the other improvements. This is because improvement 4 removes most of the work it was saving by itself; see Section 3.1.

4. **Scope.** The improvements edit 20 of Lean's source files and add 2,243 lines to them, of which the fork accounts for 16 files and 2,119 lines. Table 3 below gives a breakdown of the sizes of the fork's six changes.

5. **Switch-off environment variables.** Our `lean4.33.1-optimized` fork allows any of the improvements 1, 3, 4, 5, and 6 to be disabled individually through an environment variable. For improvement 2, disabling it is a matter of running `lean` with an "old" Mathlib library that was built with `lean4:v4.33.1` instead of an "optimized" Mathlib built with `lean4.33.1-optimized`. The off-switching feature provided by the environment variables is included so as to facilitate testing and evaluation of our `lean4.33.1-optimized` fork by the community, but will likely become unnecessary in a production setting and should be removed once the Lean developers decide on which of the improvements to incorporate into the codebase.

1.5 The two additional performance improvements

The next two improvements reduce resource usage as the six above do, but neither is a change to Lean: one is a separate program that wraps it, and the other is a way of building Mathlib. Both therefore work with the official Lean release exactly as it is downloaded today. Sections 3.7 and 3.8 give the details.

Improvement	Files edited	Lines added/deleted	Main files affected
Address reservation	3	+310 / 0	`library/module.cpp`, `runtime/process.cpp`
Shell-first layout	5	+168 / −7	`runtime/compact.cpp`, `library/module.cpp`
No-touch reference counts	4	+221 / −15	`runtime/object.cpp`, `library/module.cpp`
Lazy part loading	5	+810 / −42	`Lean/Environment.lean`, `runtime/object.cpp`
Prebuilt search index	5	+297 / −33	`Meta/Tactic/LibrarySearch`, `Meta/LazyDiscrTree`
Tactic index image	3	+547 / −19	`Lean/Environment.lean`, `runtime/io.cpp`
`lean4.33.1-optimized`, merged	16	+2,119 / −82	

Table 3: The size of each change, on macOS. Every line the fork adds or deletes is attributed to the improvement whose commit introduced it, and the counts are of those lines; the last line is the fork patch itself. A line that two improvements both worked on is counted under each of them, which is why the individual figures sum to more than the merged total. The merged figure also includes the bounding of the on-disk index that lazy part loading maintains, and the redesign of the tactic index image described in Section 3.6, neither of which is a separate improvement. The Linux patch is the same 16 files, +2,150 / −82, the difference being the address reservation's replacement by a per-region range registration.

- **Improvement 7: the snapshot wrapper script.** Lean can already save its fully loaded state to a file and reload that state instead of importing from scratch, but the facility is experimental and performs no check that the saved state still corresponds to reality: we found that a rebuilt library file makes Lean's own loader crash outright, and that command-line options are silently ignored when a snapshot is used. Our tool, `leansnap`, supplies the missing check. It generates the snapshot once for a given toolchain, library version and import line; verifies before every reuse that all three still match and that no library file has changed underneath it; and otherwise falls back to running `lean` in the ordinary way, so that a stale snapshot costs time rather than correctness. On the small cloud server this takes a single proof check from 5.8 to 1.5 seconds warm, and from 104 to 34 seconds cold. It suits workflows that run one file per process against a fixed import line — a proof-checking service, a batch of exercises, repeated command-line runs — and not a project build, where every file imports something different.

- **Improvement 8: compiled Mathlib tactics.** Mathlib's tactics are themselves Lean programs, and in a normal build they are not compiled to native code: they run in Lean's bytecode interpreter, which our census of where elaboration time goes found accounts for 30% of it. They can be compiled using only what a Mathlib checkout already contains, because `lake` emits a C file for every module and Mathlib's artifact cache already ships those C files to every user; compiling and linking them one library at a time produces eight shared libraries, invalidates no compiled library file, and requires no change to any lakefile. Elaboration processor time on our fifteen textbook proof files falls from 22.9 to 6.2 seconds, and on files from real projects by 29–39%, the difference between the two being explained by how much of each was interpreted to begin with. It does not help with interactive editing, where the waiting is dominated by the opening import rather than by tactic execution.

  In the course of implementing the compiled Mathlib tactics improvement we ran into a crashing bug, which turned out to be a known bug previously reported in the Lean tracker [3]. We fixed it, and that fix is presented as improvement 9 below. Building the shared libraries is unaffected by the bug; the crash comes afterwards, in some situations, when a file is checked with those libraries loaded. Thus, the current improvement depends on improvement 9 in order to work.

1.6 Additional improvements not directly related to performance

While working on optimizing the Lean toolchain, we discovered several other issues affecting the toolchain and addressed them with their own improvements, which are not directly tied to performance issues but are of independent utility. Like the snapshot wrapper script and compiled Mathlib tactics improvements, these are packaged as separate patches that can be deployed independently of `lean4.33.1-optimized`. These improvements consist of: a fix to a known bug; a resolution and fix for the fork-safety question from [4, 5]; and a design improvement. We offer below a summary of this work.

- **Improvement 9: a fix to the meta-initializer crashing bug #14359.** This is a crashing bug, previously reported in July 2026 as Lean bug #14359 in the Lean tracker. [3] We traced it to the fact that native code can run before the data it reads is initialized. A compiled module has two initialization routines, and one fails to invoke the other for its own module; code reached through a particular import path then reads a value that was never set, and the process crashes with a segmentation fault, deterministically, and with no diagnostic.

  The bug is classified as high-priority in the bug-tracking system, and blocks a pull request [30]. Our fix, verified across the whole of Mathlib, is nine lines in Lean's code generator, with every output byte-identical. Anyone enabling `lake`'s `precompileModules` option may encounter this bug today.

- **Improvement 10: resolution of the fork-safety question (tracker issue #87), and a partial fix.** The `fork` system call duplicates a running process, and the copy shares the original's memory for free: nothing is physically copied until one of the two writes to it. Whether a `lean` process that has loaded Mathlib can be forked safely had been asked repeatedly in the community's issue tracker [4, 5]. To date, an answer has not been provided. The fork-safety question has obvious relevance to the problem of optimizing Lean's performance (and this is how we arrived at it), since forking `lean` processes that have already imported the library would be a natural way to save on import operations when running a large, multi-file build.

  Our resolution to the question: **Lean is not fork-safe, but it can be made safer.** We established that `lean4:v4.33.1` cannot be forked safely: a small C program of fifty-three lines, linked against the unmodified Lean without needing Mathlib, hangs the child process deterministically. The same demonstration isolates what triggers the failure, showing that the child process completes successfully if the parent process's task pool was never used, and also completes if the child keeps to a thread of its own. The program is included in the release package as `code/scripts/forktest.c`, and an accompanying convenience script `code/scripts/demo-fork-hang.sh` compiles and runs it in three different ways and reports the results.

  As a demonstration that this is not the end of the story and that the fork-safety situation can nonetheless be improved, we offer a fix: a 61-line change to Lean's runtime that makes forking work correctly, under specific assumptions on when it is used, and is verified over 40 consecutive trials producing byte-identical results, including when the process is duplicated in the middle of its own elaboration. The technical details are discussed in Section 3.10.

  A related question that also bears asking is whether forking is actually *useful*. We investigated that as well, and found that on Linux, duplicating a Mathlib-loaded process costs tens of milliseconds, and can bring a single proof check on a small server down to a quarter of a second — one of the most significant savings we were able to measure. On macOS however the same operation costs several seconds and is simply not worth performing. See Section 3.10 for further discussion and details.

- **Improvement 11: language-server configuration watching.** This is not a bug fix, but rather an improvement that makes the design of the Lean interactive coding environment more robust. We observed that the Lean language server does not notice changes to a project's build configuration; it watches only Lean source files, with the effect that editing a lakefile or updating a dependency manifest leaves every open file using the configuration it was opened with, while any file opened afterwards uses the new one. Two files open in the same session are therefore elaborated under different configurations, with an open file potentially reporting a value from a dependency that has since been rebuilt. The Visual Studio Code Lean extension partially compensates for this issue with watchers of its own, but not for dependency manifests or for configuration files outside the project root, and this benefit does not extend to users working with Lean in any other editor.

  We developed a fix covering all these scenarios; the fix is 54 lines and reuses the notification mechanism the server already has for a related purpose.

1.7 Deployment and reproducibility

The improvements, the patches that carry them, the harness that measured them and the documents that argue for their correctness are collected in a downloadable package (about 1.8 MB in total) that accompanies this paper [31]. We describe the package more in detail in Appendix A. The patches are separate from one another, so a reader may adopt the `lean4.33.1-optimized` fork, one or more of the other improvements, or any combination. A script, included as `code/scripts/check-patches.sh`, can be used to confirm in a few seconds that each applies to a clean `v4.33.1` tree and that they apply together.

In order to reproduce our measurements, start by building the `lean4.33.1-optimized` fork. Detailed instructions are given in the package document `docs/building.md`. To summarize the steps, you will need to clone Lean's own repository, check out `v4.33.1`, apply the patch, and build. The process takes about half an hour and 10 GB of disk space the first time.

With `lean4.33.1-optimized` built, run the included program `code/scripts/repro-import.py` to re-measure the import figures of Table 1. This takes about five minutes. Following that, running `code/scripts/check-equivalence.sh` will rerun the byte-identical-output check in about eight minutes. To reproduce the build, editor and elaboration rows of Table 1 requires separate procedures, which are described in `code/benchmarks/README.md`. The package document `docs/switches.md` names the environment variables that turn each of the improvements off. These switches were the mechanism used for the measurements in the ablation table of Section 1.4 (Table 2). The interactive measurements are one part of the suite that can be pointed at a reader's own project and working habits. This is explained in `code/benchmarks/interactive/README.md`.

To try the `lean4.33.1-optimized` fork in an editor, give the build a name that `elan`, Lean's toolchain manager, will recognize, and then select that name in the project. Naming it is one command, `elan toolchain link lean-fork` *`path`*, where *path* is the `build/release/stage2` directory the build produced; selecting it is either `elan override set lean-fork` in the project directory, or the same choice made from the editor's own menu of Lean versions, or a line in the project's `lean-toolchain` file. The extension then starts the fork's `lean` for that project, with no more configuration being needed; Section 2 of `docs/building.md` also describes how to keep a fork setup and an unmodified setup side by side.

There are two pitfalls to avoid, involving situations where you may be misled into thinking you are running our modified `lean` when you are actually running the unmodified version, or vice versa. The first is that `lake` runs the toolchain named in the project's `lean-toolchain` file rather than whichever `lean` the shell would find first through the `PATH` environment variable, so putting the fork on `PATH` and then opening a project pinned to the unmodified release measures the unmodified release. The remedy is the one given above: point the *project* at the fork, with `elan override set` or by editing its

`lean-toolchain`. This can be confirmed by running `lake env printenv LEAN_SYSROOT` in the project directory, which prints the toolchain directory `lake` will actually use. The second pitfall is that running `lean --version` to test out which version of Lean you are running prints out the same literal string regardless of whether you are executing `lean4:v4.33.1` or `lean4.33.1-optimized`. This may seem confusing (indeed it is confusing, and is not ideal practice), but it is intentional: the fork is deliberately built to report the unmodified release's version string and commit hash, because that is what allows it to load the community Mathlib artifact cache.

The package is licensed for reuse. The patches and the code are under the Apache 2.0 license, the same as Lean itself, so that anything here can be taken upstream without a licensing question; the documents and this paper are under CC BY 4.0.

1.8 Acknowledgment of AI usage

This work was assisted by Anthropic's Claude Fable and Claude Opus 5 AI models. The models were used for coding, analysis, for running the experiments, data collation and organization, for writing the technical documentation accompanying this paper, and for writing drafts of parts of this manuscript, which were then edited by the author. The author takes responsibility for the contents of the paper and their correctness, and for the quality of the paper's writing.

2 Benchmarking the toolchain's performance

In this section we give details about our methodology for measuring Lean's performance: the benchmark we developed, the corpus of Lean projects it uses, and the hardware on which it was tested.

2.1 The benchmark

To approach the work of optimizing the Lean toolchain, we started by building a benchmark suite, `lean-perfbench`, covering five distinct kinds of Lean usage; as noted in Section 1.1, the cost of loading the library falls very differently on each, so this provides a range of realistic usage scenarios that are relevant to test. To be clear, this is a benchmark of Lean's *performance*. (This is in contrast to several existing Lean benchmarks that aim to measure Lean's ability to *prove* a given corpus of theorems.) The Lean developers maintain a continuous benchmarking service of their own, Radar [32], which records build time and peak memory for Mathlib as commits land, and which we consulted during this work. It answers a different question: it watches one project for regressions over time, where `lean-perfbench` measures five kinds of end-user activity across a corpus of projects and four machines.

Our benchmark covers the following performance metrics:

Starting up. How long `import Mathlib` takes and what it costs in memory, measured in stages from importing nothing, through single components of the library, to all of it. This is the fixed cost every other activity pays.

Checking a file. Fifteen self-contained proof files written for this project at roughly undergraduate level, which use Mathlib as a student would. Separately, we re-check real Mathlib source files in place; these are far heavier and exercise different parts of the system.

Building a project. A complete project built from scratch, which is what anyone developing a library actually does. Memory is sampled across the entire family of processes the build spawns, not merely the one we started.

Editing. Ten scripted editing sessions replayed against the real editor server to simulate live coding in an editor. These range from a beginner making and correcting mistakes to an expert working through a research-level proof, in each case timing what the Lean programmer is left waiting for. This is the only measurement that reflects what Lean *feels* like, and the only one that detects a change which improves batch throughput while degrading interactive latency.

Disk. The storage space that the toolchain and a built project occupy.

Each run records wall-clock time; CPU time, split into the program's own work and the operating system's work on its behalf; peak memory across the whole process family; the number of pages fetched from disk; and the versions of every component involved. Results are written as one self-describing file per run, so that any number can be traced back to the exact toolchain, library and machine that produced it.

Alongside the timing suite we maintain an *equivalence* suite, described in Section 1.3, which is what licenses the claim that these changes are invisible to users.

2.2 The corpus

The `lean-perfbench` benchmark draws on six real formalization projects, chosen to be large, actively developed, and differing from one another in size and other notable parameters. Each occupies 7–8 GB on disk once its compiled library cache is downloaded, although the projects themselves have source code sizes in the low megabytes. The list of component projects is shown in Table 4 below.

Project	Description	Source files	Size of source
`formal-conjectures`	open conjectures, from Google DeepMind	1,387	4.5 MB · 121k lines
Equational Theories	a very large, largely machine-generated project	1,301	9.9 MB · 164k lines
Physlib	physics formalized in Lean	721	8.0 MB · 195k lines
FLT	Fermat's Last Theorem, in progress	271	2.8 MB · 59k lines
Prime Number Theorem+	the prime number theorem and related results	236	9.7 MB · 173k lines
Carleson	Carleson's theorem on Fourier series	121	2.4 MB · 50k lines

Table 4: The corpus [33, 34, 35, 36, 37, 38]. Counts are each project's own files, not the libraries they depend on.

The projects in the corpus were chosen for two further reasons. Between them the projects pin six different Lean versions: v4.29.1, v4.32.2, v4.33.0, v4.33.1, and the two release candidates v4.34.0-rc1 and v4.34.0-rc2. That range is helpful to demonstrate that the results of the census of Section 2are not an artifact of one release: each project was measured under the Lean version it pins, unmodified. And the projects are structurally different from one another: some are deep, with files building in long chains, others flat, with a thousand files importing the same thing.

While the corpus projects were useful to get an understanding of Lean's current performance, the improvements themselves were tested mostly against Mathlib: importing it, editing files in it, and re-checking its own source files. We also tested them against the fifteen proof files described above. Among the corpus projects, only `formal-conjectures` was used for testing the improvements. One reason for this is that our improvements are built from `lean4:v4.33.1` and cannot run a project pinned to another release. So while the census covers all six projects, every comparison between unmodified Lean and `lean4.33.1-optimized` on a corpus project is made on `formal-conjectures`, the one corpus member tied to the v4.33.1 Lean version.

2.3 Experimental setup

Our performance measurements were taken on four machines with different hardware parameters, providing a range of results that give a broad sense of the reach of our optimization improvements.

A laptop. Apple M2 Pro, 12 cores, 32 GB, macOS 14.5, 16 KB pages. The primary machine, where the fork was developed and where most numbers were obtained.

A Linux container. Debian bookworm on arm64, 12 virtual cores, 8 GB, 4 KB pages, hosted on the same laptop. Linux behaves very differently from macOS on precisely the mechanism we spent most effort on, so this is where portability is tested; it is also where the library was rebuilt from source to exercise those changes that alter how compiled files are written.

A small cloud server. An AWS `t4g.large`: 2 Graviton2 cores (ARM Neoverse-N1), 8 GB, an 80 GB network-attached SSD, Amazon Linux 2023. This machine falls on the underpowered side, and its constraints expose costs that a laptop conceals: a Graviton2 core is 2.3–2.4× slower than an M2 Pro core on this code, and a cold import there takes 104 seconds against 24 in the container.

An x86-64 server. An AWS `c7i.4xlarge`: 16 vCPU (Intel Sapphire Rapids), 32 GB, Amazon Linux 2023. Rented for a single session to establish that the portable changes behave the same on the architecture most Lean users and Mathlib's own build servers run on. They do; see below.

The container and the cloud server are both Linux environments but are not substitutes for one another. The container is a virtual machine hosted on the laptop, so any measurement of *timing under memory pressure* is unreliable there: it can clear its own page cache but not the host's cache of its disk image. Cold *page-fetch counts* taken in the container are therefore sound while cold *times* are not, and the latter are reported only from the cloud server.

Every measurement records the machine's one-minute load average before and after the run, and reported figures are medians of repeated runs with their spread stated.

The Linux server measurements do not reveal a material difference in timing performance between ARM and x86-64 architectures: cold page-fetch counts agree to within 1%, and the equivalence suite is byte-identical on both. One area of difference did emerge, and it is not the architecture that matters but the machine's memory headroom. Peak resident memory depends on how much spare memory the operating system has to work with: given plenty, Linux speculatively maps neighboring pages on every fault, which inflates a small working set proportionally more than a large one. Measured on a 32 GB machine, the layout change of Section 3.2 appears to save 24% of peak memory; measured on the same machine constrained to 8 GB, it saves 48%, reproducing the ARM figure exactly.

3 Technical notes

This section adds technical detail to the eleven improvements introduced in Section 1.4 and the two subsections that follow it, one note per improvement. The notes are independent of each other; each describes the mechanism the improvement acts on, what was changed, and what the change costs. References of the form `src/library/module.cpp:535` are file names and line numbers in the `lean4:v4.33.1` source tree, and locate the code the surrounding sentence describes.

3.1 Improvement 1: address reservation (macOS only)

This improvement hinges on low-level details relating to how Lean collaborates with the OS to load modules into memory. In the specific case of macOS, we identified an inefficiency that causes this process to run much more slowly than it needs to. We explain the issue and how our fix works.

When Lean compiles a module it writes the module's data to an `.olean` file. The file is an image of the data as it sat in memory, produced by copying the module's objects into one contiguous block (which Lean calls a *compacted region*) and writing that block out unchanged. The pointers between objects are stored as the plain memory addresses they had at the moment of writing.

Lean uses the `mmap` the system call to load a file and make its contents appear directly in a process's memory. The caller names an address; on success the file's first byte is readable at that address, with the file's pages fetched from the operating system's cache as they are touched and nothing copied in advance. Lean's call site is `src/library/module.cpp:535`, inside the loader `lean_compacted_region_read`.

Which address a file wants is decided when the file is written, at `module.cpp:288-294`: it is a hash of the module's name, reduced modulo `0x7f0000000000` so that the result stays within the 127 TB of address space that user code may occupy. Addresses are therefore spread pseudo-randomly over that range, and this, crucially, is what results in the cost of the load becoming quadratic in the number of modules loaded, as we explain below.

The address given to `mmap` is a request rather than a demand: passed without the `MAP_FIXED` flag, it is a *hint*, and the operating system is free to place the mapping elsewhere if the range is unavailable. Lean passes the address recorded in the file's header as a hint. If the mapping lands where it asked, every pointer inside the region is already correct and the loader can hand the root object to the elaborator without inspecting the graph; this is the fast path taken at `src/runtime/compact.cpp:643`. If it lands elsewhere, Lean notices the mismatch (`module.cpp:537`), discards the mapping, reads the file into ordinary heap memory instead (`module.cpp:559`), and then walks the entire region rewriting every pointer it contains (`compact.cpp:659` onwards).

An `import Mathlib` directive maps 52,490 files this way: five parts for each of 10,498 modules. All but about 165 land at the address they asked for. The exceptions are not collisions between Mathlib's own files, whose ranges do not overlap at all: every one of them has a hashed address between 64 and 448 GB, which on arm64 macOS is a range reserved for the GPU and forbidden to user mappings. Those files are quietly placed elsewhere and take the rewriting path just described. This happens in unmodified Lean as well as in our `lean4.33.1-optimized` fork.

The cost that the address reservation improvement removes is on the operating system's side of the transaction, in the work of finding the requested range and confirming it is free. macOS keeps, for each process, an address-ordered linked list of the *gaps* between its existing mappings, and to satisfy a request at a given address it walks that list from the bottom of the address space until it reaches the gap the address falls in (`vm_map_store_find_space_forward`, `osfmk/vm/vm_map_store.c:454-462` in the published Darwin kernel source [39]; the list is maintained for every ordinary process, `osfmk/vm/vm_map.c:1548-1562`). Every file mapped in the middle of a gap splits it in two and so lengthens the list by one, and because the requested addresses are spread across the space rather than clustered, the k-th mapping walks past a constant fraction of the $k-1$ gaps already there. The import as a whole therefore costs $\Theta(k^2)$. This is an instance of the famous "Shlemiel the painter's algorithm", nicknamed so by Joel Spolsky in reference to the setting of an old yiddish joke [40]: each mapping restarts its search at the bottom of the address space, as the road painter in the story walks back to the paint can he never moves, and each mapping made lengthens the list the next one has to walk.

The time these walks consume is system time: time the process spends inside the operating system rather than in Lean's own code, and none of it spent on input or output. With the file cache warm, unmodified Lean spends 7.57 of its 10.17 seconds there. The address reservation described below brings that to 1.66 seconds and the wall time to 3.83.

Mapping the files in descending order of requested address would end each walk immediately, and over the real files it does: 0.78 seconds. That order is out of reach. Lean discovers which modules to import by reading each module's list of imports out of the module itself, so files are mapped in the order the import graph is uncovered, and the order cannot be known in advance. Obtaining it would mean

either consulting `lake`'s `-setup`, which a plain `lean` invocation has no access to, or reading 2.19 GB of file headers before mapping anything. Table 5 shown below gives the alternatives we measured, in a small C program that replays the loader's exact sequence of system calls over the real files. Its two ordering rows behave as the list walk predicts: descending order is the fastest strategy of the six, and ascending order, in which every mapping walks the whole list, the slowest.

mapping strategy	system time	
hinted `mmap`, Lean's import order (unmodified)	6.1–7.2 s	the baseline
hinted, ascending requested address	11.5–13.4 s	the worst case
hinted, descending requested address	0.78 s	unreachable: imports are read as mapping proceeds
`MAP_FIXED` into free space, Lean's order	4.1 s	the free range still has to be found
reserve above 1 TB, then `MAP_FIXED`	0.89–0.90 s	what we did; plus 0.1 s to reserve
the same, in descending order	0.87–0.89 s	order has stopped mattering

Table 5: The mapping loop in isolation, on a quiet machine, two runs of each strategy. The residue of about a second is the four file-system calls per file and the `mmap`s themselves, which no strategy avoids.

Our improvement removes the search instead of reordering it. It adds to `src/library/module.cpp` a class, `olean_address_reserver`, which claims address space up front and then hands out pieces of it to the loader. Here is how this is achieved: before mapping its first region, Lean asks the operating system which parts of its own address space above 1 TB are free, and claims each such free range $[s, e)$ with a single mapping backed by no file. Here is a code snippet from our patched `module.cpp`:

```
// lean4.33.1-optimized, src/library/module.cpp:148,
// in olean_address_reserver::reserve
void * p = mmap(reinterpret_cast<void *>(s), e - s, PROT_NONE,
                MAP_ANON | MAP_PRIVATE | MAP_NORESERVE, -1, 0);
```

This costs a tenth of a second and claims about 126 TB. `PROT_NONE` makes the claimed range unreadable and unwritable, so nothing can use it by accident, and `MAP_NORESERVE` tells the operating system that no physical memory need ever be set aside for it. The `MAP_FIXED` flag is deliberately absent: between asking which ranges are free and claiming one, another thread may have mapped something there, and `MAP_FIXED` would overwrite it without warning. If the address returned is not the one asked for, Lean releases it and asks again.

Each region file is then mapped over the top of the address reservation. This is the same class's `map_file`, which the loader now tries before the ordinary hinted call of `module.cpp:535`:

```
// lean4.33.1-optimized, src/library/module.cpp:213,
// in olean_address_reserver::map_file
void * p = mmap(base_addr, size, PROT_READ | PROT_WRITE,
                MAP_PRIVATE | MAP_FIXED, fd, 0);
```

Here `MAP_FIXED` is what is wanted, and it is safe, because the range being overwritten is known to be our own reservation. The operating system is no longer searching for free space but subdividing a range it has already granted, which it settles by a direct lookup rather than a walk. The reservation is a single entry that shrinks rather than a growing list of separate ones, so the cost per file no longer depends on how many files came before it or in what order.

The bookkeeping the class needs is one map of reserved-but-still-unused ranges, guarded by a mutex. A file whose requested range is not inside the address reservation — because it lies below 1 TB, or in the GPU range, or because the address reservation was disabled or failed — falls through to the original

hinted call at `module.cpp:535`, unchanged. Freeing a region restores the `PROT_NONE` claim over its range and returns the range to the map.

Everything that happens after a mapping succeeds is untouched, including the fast-path test at `compact.cpp:643`, so a region avoids the pointer-rewriting walk under exactly the same condition as before. The file format, the addresses recorded in it and the import order are all unchanged, and no Lean source file is edited; the change is C++ only, which is why the stage-0 bootstrap is unaffected by it.

The address reservation raises the process's virtual size to about 126 TB. This is visible in a process listing and has no other consequence: `PROT_NONE` pages are never touched, so no page tables are built for them and they occupy no physical memory. Peak resident memory is unchanged to within 10 MB. `mimalloc`, the allocator Lean uses, requests its own large allocations between 2 and 30 TB, which now falls inside the address reservation; those allocations are placed elsewhere and `mimalloc` accepts a different address, which is its documented behaviour when a request cannot be met.

One regression came with the change, and we devised a fix to mitigate it, which is included as part of improvement 1. A 126 TB reservation is free to hold but not free to duplicate, and the `fork` system call, which creates a new process by duplicating the current one, has to copy the description of the entire address space. Each `fork` from a reserving process cost about 0.4 seconds. Lean starts its child processes at `src/runtime/process.cpp:453`, in the function `spawn`, with a `fork` followed by an `execvp` at `process.cpp:510`. `lake` starts sixteen `git` processes on every invocation, so `lake env true` went from 1.30 seconds on unmodified Lean to 11.2. The solution is to start child processes with `posix_spawn`, which creates the new process directly rather than by duplicating the parent, and takes about 0.002 seconds per child. On macOS the fork now takes that route instead, in a block that occupies `process.cpp:474-573` in our patched source and calls `posix_spawnp` at line 543. With `fork`, the child process did its own preparation before calling `execvp`: redirecting the three standard streams, closing the parent's ends of the pipes, and changing directory. `posix_spawnp` leaves no such window, since there is no child to run code in until the new program is already running, so those same operations are prepared in advance and handed to the call as a list. The same invocation now takes 0.74 seconds — faster than unmodified Lean, because `fork` was already expensive for a process holding all of Mathlib.

The address reservation improvement changes nothing for the 165 files whose requested addresses fall in the GPU range, since those addresses cannot be claimed in the first place; as in unmodified Lean, they are mapped elsewhere and relocated. Repairing that would mean not assigning addresses in the GPU range when a module is written, one line at `module.cpp:288-294`. We did not, because the change would reach a user only through a rebuilt library, and the files concerned are 0.3% of the total.

It's worth considering the value of the address reservation improvement not just as a standalone fix but in the context of the overall set of improvements that `lean4.33.1-optimized` comprises. Applied on its own Lean, improvement 1 is worth 6.34 seconds. However, removed from the finished fork, it is worth a more modest 0.16 seconds (Table 2). The reason is that the cost improvement 1 removes grows with the square of the number of mappings, and as we will see in Section 3.4, improvement 4 reduces that number from 52,490 to 10,498, and therefore shrinks the very term this improvement attacks, by a factor of about twenty-five. Thus, the order in which the two improvements are counted decides which of them is credited with the savings. Regardless, we believe they both have value and are worth adopting.

To conclude this subsection, let us emphasize once more that the address reservation modification has the distinction in our set of improvements of being relevant only on macOS. As it happens, Linux does not have the problem that improvement 1 solves: it keeps a process's mappings in a balanced tree annotated with the largest free range in each subtree, so finding space takes time logarithmic in the number of mappings, and no amount of scattering makes it quadratic. The same 52,490 files map in 0.64 seconds there against 9.5 seconds on macOS. Our changed code is compiled only on macOS, and Linux and Windows builds are byte-for-byte what they were.

3.2 Improvement 2: shell-first `.olean` layout

Start-up reads about 100 MB of information out of the compiled library and makes 2.8 GB of it resident. That ratio is what creates the opportunity for this improvement: the data that Lean needs at start-up is small, but it is stored interleaved with data that is large and not needed at all.

For each constant a module defines, its `.olean` holds a *shell* — the `ConstantInfo` record and the two small structures beneath it, 80 to 110 bytes in all, holding the constant's name, its hints, and pointers to its type and its value — and, separately, the type and the value themselves, which are expression trees and routinely run to kilobytes. Importing a module builds a map from name to `ConstantInfo`, which requires reaching every shell in the file. It does not require the trees; those are followed later, if some tactic or elaboration step unfolds that constant, which for most of Mathlib's 771,000 constants never happens in a given session.

The compactor that writes an `.olean` performs a depth-first, post-order walk of the object graph (`object_compactor::to_offset`, `src/runtime/compact.cpp:202`). It can emit an object only once all of its children have addresses, so it writes the children first and the parent immediately behind them. For a constant that means the type tree, then the value tree, then the shell that points at both. Repeated for every constant, the shells end up distributed evenly through the file rather than gathered anywhere. Measured on files written by unmodified Lean, the shells of `Mathlib/Analysis/Calculus/Deriv/Basic.olean` occupy 22 of the file's 27 pages, and those of `Mathlib/Logic/Basic.olean.private` 39 of 62.

Memory arrives from disk one page at a time, and a page on the laptop we developed `lean4.33.1-optimized` on is 16 KB. Touching one 90-byte shell therefore makes an entire 16 KB page resident, nearly all of it a proof term that nothing will read. Done 771,000 times, the 100 MB of shells drags in 2.8 GB of expression trees behind it.

The solution that we developed for this inefficiency changed neither what an `.olean` contains nor how it is read, but the order in which the compactor emits objects: each file now begins with its constant names, then all of its shells in one block, then the arrays and extension entries the start-up path reads, and only then the expression trees.

Post-order exists for a reason: a parent's content is not final until its children have addresses, so the shells cannot simply be moved to the front. They are *reserved* there instead. The whole of the new arrangement is visible in the three calls the compactor now makes, at `compact.cpp:534-536` in our patched source:

```
// lean4.33.1-optimized, src/runtime/compact.cpp:534-536,
// in object_compactor::operator()
process_front();
object_offset off = to_offset(o);
compact_pending();
```

The middle line is the unmodified traversal, unchanged. The other two are the shell reservation and the patching.

`process_front` (`compact.cpp:478`) runs before the traversal, over a list of objects the caller has nominated. For each shell it calls `reserve_shallow` (`compact.cpp:470`), which copies the object to the current end of the buffer with its header and scalar fields filled in and its pointer fields left blank, registers that copy in the table mapping heap objects to region addresses so that every other reference to the same object resolves to the reserved copy, and records the pair on a pending list.

The traversal then runs as before, emitting the trees into the body of the file. Afterwards `compact_pending` (`compact.cpp:506`) walks the pending list and patches each reserved shell's pointer fields with the addresses its children have by then been given. That list stores buffer offsets rather than pointers,

because the buffer is grown by reallocation as the file is built and any pointer into it would be stale by the time it was used.

Which objects are nominated is decided outside the compactor. A new entry point, `lean_save_module_data_part` (`src/library/module.cpp:658`), reads the module's constant table and hands over the names array and, for each constant, its three shell objects; they reach the compactor through `add_front` (`compact.h:109`, called at `module.cpp:505`). The compactor itself knows nothing about constants, and a `ModuleData` it does not recognize is silently left in the unmodified order.

One case is handled separately, because getting it wrong costs disk storage space. A module's parts share one compactor, so a shell whose expression trees were already emitted into an earlier part has nothing to defer, and reserving it anyway would stop the compactor recognizing it as a duplicate of an object already written. Such shells are compacted normally, which costs nothing and preserves the structural sharing.

The format version does not change, the reader is not touched, and files written the old way continue to load. That the object graph is unchanged rather than merely equivalent was checked with layout-independent fingerprints, and the writer's switch set to its default value reproduces the old byte order exactly. In Lean's own `Prelude` the shells go from 68% of the file to 6.7%; in `Lean/Expr.olean`, from 24 pages to 4.

Added to improvements 1 and 3, the shell-first layout improvement takes peak memory for an `import Mathlib` directive from 3.44 to 2.76 GB and page reclaims from 286,356 to 244,516, at no cost in time. A `vmmap` of an idle process after the import attributes the difference exactly: resident `.olean` bytes fall from 1.28 to 0.89 GB and resident `.olean.private` bytes from 0.84 to 0.55 GB, with every other category unchanged.

The saving is not confined to the import. The dense packing is a property of the file, so a constant looked up in the middle of a proof also finds its shell on a page filled with other shells, rather than faulting in a page that is mostly somebody's proof term. Three lemma-heavy textbook files stay 0.68 GB below the previous configuration for their whole run, not only at start-up.

The shell-first layout improvement comes with two costs, both small. First, the compiled files grow by 0.04% — 2.9 MB across the 8.09 GB corpus — because a shell whose trees were emitted in an earlier part is no longer always recognized as a duplicate. Second, the library has to be rebuilt: the shell-first layout is written into the files, so a user gets nothing from this improvement until somebody compiles Mathlib with it. That is the reason, already discussed in Section 1.4, why improvement 3, which targets the same inefficiency but does not require a rebuilt Mathlib in order to work, is useful to ship alongside this one in the `lean4.33.1-optimized` fork.

3.3 Improvement 3: no-touch reference counts

This improvement attacks the same memory as the previous one, from a different direction, and it comes down to a single machine instruction that Lean did not need to execute.

`finalizeImport` (`src/Lean/Environment.lean:2307`) builds the constant map by looping over each module's parallel arrays of names and `ConstantInfo`s. A name is only hashed and compared, so the compiler passes it borrowed. A `ConstantInfo` is stored into the map, so the compiled code must own a reference to it, and taking ownership increments the object's reference count — which begins by reading that count out of the object's header (`lean_inc_ref`, `src/include/lean/lean.h:659`).

For an object in a memory-mapped region that read is pointless. The compactor gives such objects a reference count of zero, which the runtime treats as persistent: the increment is a no-op and the object is never freed. The comment on `lean_is_persistent` (`lean.h:596`) says as much — "we never update the reference counter of objects stored in compact regions". Nothing is written back, and what remains is a single load instruction: the one in `lean_is_st` (`lean.h:591`), which fetches the count to decide

which branch of the increment to take and for these objects always finds zero. That load is the whole cost. Every one of the 771,000 constants is touched once, to no effect, and each touch makes another 16 KB page resident.

The natural fix is to test whether an object is persistent and skip the increment if it is. Unmodified Lean has that test, `lean_is_persistent` at `src/include/lean/lean.h:596`, and it works by reading the reference count out of the object's header — the same read that faults the page in. Using it would cost the page fault it was meant to prevent, and the loop would touch all 771,000 shells exactly as before.

The fork decides the same thing from the pointer alone, without dereferencing it. The loader registers the address range of every mapped region in a table (`lean_register_persistent_address_range`, implemented at `src/runtime/object.cpp:646`), and a pointer that falls inside one of those ranges necessarily denotes an object of a memory-mapped region, which is persistent. Testing it is then an interval lookup in that table — `lean_is_in_persistent_address_range`, `object.cpp:671` — which reads none of the library's memory.

The array read the constant-map loop performs is replaced by one that consults the address-range table first: `lean_array_fget_no_touch` (`object.cpp:691`) skips the increment for an element inside a registered range and takes the ordinary path otherwise. Both constant-map loops in `finalizeImport` use it, and the routine that later marks the map persistent gets the same guard (`lean_mark_persistent`, `object.cpp:716`), since it too was reading every header it passed and so reaching the same pages a second time.

The Lean-side change is smaller than it sounds but not trivial, because the loop had to be restructured so that the compiled code holds exactly one owned reference to each `ConstantInfo` rather than relying on the array read to produce it.

On macOS they are free: the address reservation of improvement 1 has already claimed them, and the address-range table is filled in as a side effect of placing each region. On Linux there is no reservation to inherit, deliberately, since covering the address space with `PROT_NONE` mappings there would inflate the process's mapping count against the per-process limit. Instead each region that lands at its saved address registers its own range as it is mapped and unregisters it before being unmapped, the address-range table being a lock-free snapshot indexed by bucket so that lookups stay cheap at 52,000 entries. Where no range is registered — on Windows, with the address reservation switched off, or for a region that had to be relocated — every path degenerates to the unmodified code.

Added to improvement 1, the no-touch reference counts improvement takes peak memory for `import Mathlib` from 5.69 to 3.44 GB and page reclaims from 423,383 to 286,356, with user time unchanged to within measurement noise: no work is being done faster, work is simply not being done. A `vmmap` of the idle process afterwards shows resident `.olean.private` bytes falling from 2.59 to 0.84 GB and resident `.olean` bytes from 1.78 to 1.28 GB, every other category identical, which is the signature to expect if the only thing that changed is which pages of the library were touched.

As discussed in Section 1.4, the marginal value of this improvement in the finished fork is small (Table 2), because improvement 2 removes most of the same cost and does so for every later lookup rather than only at import. The case for keeping both is that improvement 2 requires a rebuilt library and this one does not.

An important note on possible duplication. Lean's own developers have a draft improvement (pull request #14362 [41]) that targets the same waste being addressed by the shell-first layout and no-touch reference counts improvements, building the constant map lazily so that most shells are never visited. It was unreviewed and unmerged when we last checked. If it is accepted, the no-touch reference counts should be retired in its favor: they are the least interesting of the three ways to solve the problem, and the only reason to prefer them is that they work today on a library nobody has to rebuild, in contrast to both our improvement 2 and the 14362 draft.

3.4 Improvement 4: lazy part loading

Improvements 2 and 3 above reduce what it costs to reach the library's pages: the shell-first layout packs them so that fewer are needed, and the no-touch reference counts stop the import reaching them at all. By contrast, the lazy part loading improvement reduces how much of the library is opened in the first place.

A compiled Lean module is not one file but five. The `.olean` holds the public view: names, types, and the bodies other modules may see. The `.olean.private` holds proof bodies, the bodies of unexposed definitions, and private declarations. The `.olean.server` holds information the editor uses and nothing else does. Two more, `.ir.sig` and `.ir`, hold the compiled code Lean's interpreter runs when a tactic executes. The grouping is written down in the `ImportedModule` structure, `src/Lean/Environment.lean:2007-2012`. For `import Mathlib` that is 52,490 files and 7.5 GB, and unmodified Lean maps all of them: the loader's own comment at `Environment.lean:2233` reads "opportunistically load all available parts".

This is again a source of inefficiency and an opportunity for savings. In practice, almost none of the files that are loaded are actually consulted. A proof body matters only if something unfolds it, compiled tactic code only if that tactic runs, and the editor part not at all in a batch run. Lean has a newer module system that addresses part of this. A file that begins with a `module` declaration has its dependencies imported at the public level, and the parts holding what is not publicly visible are then not loaded at all: checking such a file maps 36,400 files rather than 52,490 and ends about a gigabyte lower. The older style of file — one that opens directly with `import Mathlib` and carries no `module` declaration — still loads everything, and that is what nearly everyone writes today, including every file in our corpus.

With the improvement we created in place, the import now maps only the `.olean` part of each module. The other parts are registered rather than read, and fetched when something asks for them. A private part is loaded when something asks for what the public part leaves out. In the public file a theorem is recorded in the way an axiom would be: its name, universe parameters and statement are kept, and its proof is dropped; a definition whose body is not exported, and an `opaque` declaration, are recorded the same way. That is how 453,291 of the Mathlib closure's 664,547 exported constants are stored. A lookup needing no more than the statement is served from the public part; one needing the proof or the body loads the module's private part, as does a lookup of one of the 142,218 names that exist only there, or an extension reading that module's entries. A compiled-code part is loaded the first time the interpreter asks for that module's code. Files mapped fall from 52,490 to 10,498 and bytes mapped from 7.50 to 2.19 GB.

A file using the older style without a `module` declaration does need a little information from the deferred parts eagerly, before anything asks: the code generator's auxiliary names, for which the module-index map must be complete; declarations marked to run at initialization; the names of private-only constants; and the private-level entries of extensions that are not lazy, such as a `private theorem` carrying `@[simp]`. That is a small amount of data, and it lives in the files whose opening we are trying to defer. Thus, in devising the improvement, a mechanism was needed to recover this data.

Our improvement recovers that data from a side file, read from a cache directory and keyed by the module list together with the sizes and modification times of the deferred files. The file is created on the first import of that closure, and read from the cache afterwards. A stale key is considered a miss, so a library rebuilt underneath the cache is recomputed rather than misread. There is one side file per distinct import closure, not one per source file compiled, and its size follows the number of constants in the closure: e.g., 59 MB for `import Mathlib.Topology.Basic`, 73 MB for `import Mathlib.Analysis.Calculus.Deriv.Basic`, 81 MB for `import Mathlib.Tactic`, 103 MB for `import Mathlib`. Most of each file is names: some 762,000 auxiliary names and 142,000 private ones for the full

closure.

The side files accumulate in one directory, making it necessary to put measures in place to ensure this cache does not grow to fill unbounded amounts of disk space. The measures we added are standard for caches of this type: nothing new is written when free disk space is below a floor, 10 GB by default, though what is already there is still read; the cache directory is held under a size cap, 5 GB by default, by discarding whole entries oldest-first after each write; and an admission threshold decides how many times a closure must be seen before its file is written. At the default admission threshold of one, the first import writes, and when set to two, a from-source build writes nothing at all. We recommend to the Lean developers evaluating our proposal to consider which values for these parameters make sense as defaults in the event that this improvement is to be incorporated into the Lean codebase.

The mechanism of the lazy part loading improvement is designed to help with Lean work that repeats one import line multiple times, as can happen in an interactive editor session, a proof-checking service, a batch of exercises, or repeated command-line runs against `import Mathlib`. One closure means one file, written once and read thereafter, and setting aside an extra 103 MB of disk space is a small price to pay compared to the 7.5 GB of compiled Mathlib library whose loading this optimization allows deferring.

On the other hand, the lazy part loading improvement does not yield meaningful benefits for a typical project build, where almost every file imports something different, so closures never recur and every write is wasted. That is not a hypothetical limit: the build figures of Table 1 were measured with lazy part loading switched off, and are the work of improvements 1 to 3 alone.

The lazy part loading improvement provides the largest performance boost to an `import Mathlib` measurement of any of the `lean4.33.1-optimized` improvements (Table 2). Added to improvements 1 to 3, lazy part loading reduces the import timing from 3.57 to 2.79 seconds and peak memory from 2.76 to 1.64 GB, with page reclaims falling by 68,379 and system time nearly halving, from 1.49 to 0.86 seconds.

However, when we developed lazy part loading we observed a different way in which it makes performance *worse*. An `exact?` command examines the type of every imported constant, and under lazy part loading that walk pulled 9,859 of the 10,498 private parts back in with a dozen threads serializing on the loader. The first `exact?` went from 9.0 to 20.2 seconds, and from 24 to 160 seconds of processor time, markedly worse than unmodified Lean. This led us to develop improvement 5, described in the next subsection, which manages to claw back all of the degraded performance associated with this regression, and yields additional performance benefits.

A second bug came to light in the same experiments, and we fixed it too. When Lean stores a value into a reference that threads share, it marks everything reachable from that value as multi-threaded (`lean_mark_mt`, `src/runtime/object.cpp:798`), walking the object graph and reading each object's header as it goes. A value reaching into the compiled library therefore faulted in the pages the no-touch reference counts had just avoided. Our fix gives that walk the same interval test those reference counts use: an object inside a memory-mapped region is persistent and can never be single-threaded, so it is skipped without its header being read (`object.cpp:802` and `813`). Unmodified Lean reads the same headers in the same walk, at no cost, because by that point the import has already made those pages resident.

A possible improvement to the improvement. While the lazy part loading improvement is finished and provides real benefits, there is an even better and conceptually more correct way to implement the same optimization: if the data Lean needs eagerly from a module were stored in that module's own public file, the cache and much of the complexity of the present design would be unnecessary. Such a "true" lazy part loading implementation would require a change in file formats, and for that reason we did not pursue it in this project, but it is an idea the Lean developers can consider.

Lean's own developers have a draft improvement here too (pull request #14145 [42]), which defers

the loading of compiled code in much the way lazy part loading defers it. It was a draft, unreviewed and labeled as breaking Mathlib, when we last checked. It is also narrower: we measured it as helping only files that carry a `module` header, and costing about 25% more memory for a file without one, which is the style nearly everyone writes today.

3.5 Improvement 5: prebuilt search index

Improvement 5 has two components. The first one fixes the performance degradation caused by improvement 4 in connection with processing `exact?` tactic calls, as described in the previous subsection. We start by explaining how that regression comes about and how we fixed it, then describe the second part of improvement 5.

The `exact?` tactic asks Lean to find a lemma that closes the current goal. In unmodified Lean we measured the first such call in a process as costing 4.9 seconds and 2.2 GB. It is the slowest thing a beginner meets in the editor, and it is paid again every time a file is reopened.

Before it can search, `exact?` builds a discrimination tree over every imported constant: all 771,000 of them. For each one it decides whether the constant is a candidate at all and computes the key it would be filed under: the head symbol of its conclusion and the first few arguments beneath it. Computing that key means instantiating the constant's type and reducing it, which is elaboration work, done once per constant, on the first call and not before. The per-constant step is `addImport`, `src/Lean/Meta/Tactic/LibrarySearch.lean:140`.

The source of the regression caused by improvement 4 has to do with a particular aspect of `exact?`'s design: one of the filters applied to each constant discards *recursors*, the induction principles Lean generates for each inductive type, since those can never satisfy what `exact?` is looking for. That filter was written in a way that needed the constant's full definition rather than its statement. For a module whose private part had been deferred, obtaining the full definition meant loading that part, and the first theorem examined in each module was enough to require it. A single `exact?` therefore loaded 9,859 of the 10,498 private parts the import had just declined to open.

It turned out that this problem has an easy fix, which required a total of sixteen lines of code. The filter is `isRecCore` at `src/Lean/MonadEnv.lean:36`, and in unmodified Lean its one line of body resolves the constant in full. The public part of a module already records whether a constant is a recursor, because a recursor is stored there in full rather than reduced to its statement as a theorem is, so the filter can read the kind from data the import has already mapped. Our patch adds a lookup that does exactly that, `findKindNoLoad?` (`src/Lean/Environment.lean:1548`), and has `isRecCore` call it (`MonadEnv.lean:40`); both line numbers refer to the `lean4.33.1-optimized` source. With the fix in place, private parts loaded during `import Mathlib` plus one `exact?` fall from 9,859 to 120. This fix returns the situation to roughly where unmodified Lean already stood in terms of performance when handling `exact?` calls.

The second component of our improvement goes beyond repairing a regression, and produces actual performance gains. It moves the computation of the keys out of the process that runs `exact?` and into the compilation of the library: when a module is compiled, Lean now computes, for each constant that module defines, the keys the constant would be filed under, and writes them into the module's `.olean` file along with everything else the module exports. Storing them makes those files 3.3% larger, and they then travel with the compiled library exactly as its other contents do. We refer to those stored keys as the *prebuilt search index*, which is what this improvement is named for. A library compiled by unmodified Lean carries no such keys, and the improvement then falls back: either to a cache file of assembled entries, about 51 MB for the `import Mathlib` closure and written by the first `exact?` of a session, or, when no directory is named for that file, to the same walk unmodified Lean performs. As with the shell-first layout improvement (improvement 2), the full benefit reaches a user only once

somebody compiles the library with the fork.

Computing the keys in advance is possible because they do not depend on anything that varies from one session to the next. A constant's root key and the first three keys beneath it are fixed by its type together with facts settled in the module where it is declared: reducibility and deprecation can only be set there, and whether the constant is a class, which of its binders are instances, and whether a literal is a numeral are properties of the type itself.

The first `exact?` of a session then assembles its discrimination tree from the stored keys, with no elaboration at all. The tree it builds is also smaller, and stays smaller for as long as the process lives. Unmodified Lean stores in every entry an instantiated copy of that constant's type, against the possibility that a search reaches it; an entry in our version stores the constant's name instead, and the type is recovered only when a search visits that candidate.

In unmodified Lean the first `exact?` after `import Mathlib` costs 4.9 seconds, 2.2 GB of additional resident memory and 29 seconds of processor time. In the finished fork it costs 1.5 seconds, 0.2 GB and 12 seconds. It is the largest single improvement in the fork on that workload, by an order of magnitude over anything else (Table 2).

One may wonder if timing the first `exact?` call measures the correct thing; is it possible that the fix simply pushes the same resource usage down to the second call or beyond, in which case an improvement to the first call's timing would only be an illusory savings? In fact, with our design the cost has *not* been moved to the second call; rather, it has been eliminated from the session, at the cost of requiring an upfront library recompile and a modest increase to compiled module file size. The tree is built once per process and reused by every later call, and the new path does less work to reach it: unmodified Lean computes an entry for all 771,000 constants up front, where ours reads stored keys and elaborates only what a search visits. What remains resident afterwards, to the benefit of everything the process does next, is 0.2 GB rather than 2.2.

An important note on changes in behavior. In the introduction we laid out our guiding principle that our improvements should not change Lean's actual user-facing output in any observable way. Improvement 5 is the one place where that principle is not fully met: there are files on which an `exact?` call under `lean4.33.1-optimized` offers different lemma suggestions from the same call run against `lean4:v4.33.1`. The difference is confined to which lemmas are offered. Any suggestion the fork makes is elaborated and checked by the kernel exactly as before, so no incorrect proof can be accepted; but the suggestions are not guaranteed to be the same ones, and a lemma that `lean4:v4.33.1` offers can in principle be missed.

The difference arises because unmodified Lean computes the keys during the first `exact?` of a process, in whatever context that call finds itself, whereas in `lean4.33.1-optimized` they are computed when the defining module is compiled and cannot be affected by the file making the call. The route we were able to construct runs through reducibility, and it is a narrow one: `lean4:v4.33.1` refuses `attribute [local reducible]` on an imported constant outright, with the message that "`[reducible]` affects the term indexing datastructures used by `simp` and type class resolution", so reaching the difference at all requires the user to set the `allowUnsafeReducibility` configuration option. Unmodified Lean's tree is also built under the calling context's configuration and options, which we did not examine, so we do not claim reducibility is the only route.

With additional work that we did not undertake, the prebuilt search index improvement can be extended further to conform with the guiding principle of no behavioral change. It would be enough to compare the reducibility state of the calling environment with the one the keys were built under, use the stored keys when they agree, and fall back to the walk unmodified Lean performs when they do not. The test would be cheap, making use of the local reducibility entries of the file being checked, and it will degrade to Lean's current behavior in exactly the cases where the two could differ, which would make

improvement 5 faster or equal in every case and byte-identical in all of them. `lean4.33.1-optimized` as it currently stands does not do this.

Further improvements to the prebuilt search index are possible. The mechanism of this improvement can be extended to also cover the `rw?` tactic. `exact?` and `apply?` are served from the stored keys, but `rw?` builds its index by a different route, reducing under binders and filtering differently, and still performs the full walk. Extending the mechanism to cover it as well would be perhaps forty lines of additional code, which we did not write.

3.6 Improvement 6: tactic index image

Tactics do not search the library one lemma at a time. `simp` consults a discrimination tree of rewrite rules, instance resolution consults a tree of instances, and a dozen other mechanisms consult tables of their own. Those tables are not stored. Lean rebuilds each of them at every start-up by replaying the contributions of every imported module, from data that has not changed since the last time it did so; the replay is the `addImportedFn` call at `src/Lean/Environment.lean:1988`.

Running every persistent extension's import hook accounts for 839 milliseconds on the fork, close to a third of the 2.79 seconds an `import Mathlib` takes there without the improvement described below. The largest contributors are `simp`'s tree at 99,669 imported entries, the instance tree at 43,527, the compiler's specialization cache at 37,708, the match-equation table at 26,842, `grind` at 13,348, and Mathlib's own `to_additive` and `to_dual` tables at 25,422 and 11,377. There are 93 such tables in all for `import Mathlib`.

Our improvement targets this source of inefficiency with an added mechanism to generate a file that contains the information needed by the tactics. This file is associated with a given import set (with `import Mathlib` being one of the main use cases). We call this file the *tactic index image*. Here is how it works: to generate the image, `lean4.33.1-optimized` probes every persistent extension by asking the object compactor — the same machinery that writes an `.olean` — to serialize that extension's assembled imported state. Those the compactor accepts are written into a single compacted region. The essential detail is what that region uses as its dependencies: the `.olean` regions of the closure itself. Anything already present in a library file is referenced at its address in that file rather than copied, so the image holds the structure of the tables and not the terms they point at. For `import Mathlib` it is 212 MB.

The tables an image holds combine the contributions of every module in the closure, so an image belongs to an import set rather than to any one module and cannot be stored inside the modules themselves. An image is never generated automatically. To generate one, a user runs `lean` once, with the writing switch set, on a file carrying the import line the image is to serve and the resulting file is placed beside the library's compiled files. For the case of `import Mathlib`, this takes about 12 seconds.

Reading an index image is done automatically. A run whose imports match the set some available image was built for maps that image and installs each stored state as the extension's imported state, in place of running the extension's import hook; a run with no matching image rebuilds the tables as Lean always did. Mapping the file rather than copying its contents into memory saves memory as well as time: the tables are never copied into the process, and a page of the image is fetched only if some tactic consults that part of that table. The measured effect is fewer page faults: 160,553 against 176,179 without a tactic index image.

Switching the tactic index image off in the finished fork costs 0.46 seconds and 0.25 GB on `import Mathlib`, taking it from 2.33 seconds and 1.39 GB to 2.79 and 1.64. Most of that is user time, 1.24 seconds against 1.63, which is what to expect when the change removes computation rather than page faults.

As with improvement 2, the tactic index image improvement reaches users only if somebody distributes the artifact: for Mathlib, that would mean distributing an additional 212 MB file alongside the 7.5 GB of compiled files its users already download, an increase of 2.8%, built once per release. This scenario is the main one we had in mind when designing the improvement, since an editor session, a proof-checking service and most newcomers will repeatedly run files with an `import Mathlib` directly, which will all benefit from a prepackaged tactic index image. By contrast, the improvement will not help with building a library from source: for example, Mathlib's 8,705 modules each import a different set of modules, and each would need an image of its own.

A logistical limitation on shipping the improvement. A tactic index image is of use only on the machine that built it, and that is a limitation rather than a design choice. Lean finds an image by a key computed from the full path of every compiled file in the import closure, while `lake` unpacks a downloaded library into the user's own directory rather than the one a distributor built in. An image shipped alongside a library would therefore be looked for under a key built from paths that do not match, which would result in the image found to be missing and the tables rebuilt exactly as before.

Making a distributed tactic index image usable therefore means keying on the module-relative part of each path rather than the whole of it, which would be an edit of about three lines to `lean4.33.1-optimized` at `src/Lean/Environment.lean:635`. Such an edit would weaken the key as a check against a stale image, but not the guard that matters: the reader refuses an image whose dependency regions are not the ones it was saved against. We have not made that change, but it would be easy to make.

3.7 Improvement 7: the snapshot wrapper script

This improvement and the next are not part of `lean4.33.1-optimized` and neither requires a patched Lean. This one is a way of running the official Lean that avoids repeating work, usable today by anyone.

The idea is to take advantage of the fact that Lean can save its fully loaded state to a file and reload that state instead of importing from scratch: the options `-incr-header-save` and `-incr-load`, handled in `src/Lean/Elab/Frontend.lean`, lines 283–284. This creates an opportunity wherever many files are checked against the same import line: the import is paid for once, and every later file pays only the cost of reading the saved state.

The Lean snapshot-saving feature has some limitations one needs to work around. It does not check that a saved state still corresponds to reality, and this can lead to a crash when running with a rebuilt library file. Moreover, command line options are silently ignored when a snapshot is used.

Our solution was to create `leansnap`, a Python script of around 350 lines, that supplies the missing checks. It generates a snapshot once for a given toolchain, library version, import line and set of `-D` command-line options, and files it under a key derived from all four; before every reuse it verifies that they still match and that no library file has changed underneath it; and it falls back to running `lean` in the ordinary way if anything fails to match. The options are part of the key rather than something checked at reuse because there is nothing left to check by then: an option is baked into a snapshot when it is saved and disregarded when it is loaded. A run with different options therefore finds nothing filed under its key and either saves a snapshot of its own or runs `lean` plainly, so an option given on the command line can never be silently dropped.

per check	ARM server		x86-64 server	
	plain	snapshot	plain	snapshot
warm	5.8 s · 6.0 GB	1.5 s · 1.8 GB	2.87 s	0.71 s
cold	104 s	34 s	109 s	25 s
ten in succession	6.0 s each	1.5–2.0 s each	3.04 s each	0.82 s each

Outputs were byte-identical to a plain run in every comparison. For `import Mathlib` the snapshot is 310 MB, and identical in size to the byte on both architectures (although the snapshots themselves are not identical, containing machine-specific pointer addresses). Two smaller files sit beside it — the list of library files the snapshot depends on, and the manifest of their sizes and timestamps that `leansnap` checks — bringing the total to 322 MB. The plain cold page-fetch counts agree to within 0.3%, so the mechanism is the same on both and only the processor speed differs.

The snapshot wrapper script pays off wherever many files are checked one after another against the same import line, each in its own process; examples may include a service that checks submitted proofs, a set of exercises to be graded, a benchmark suite of independent files, or the same file re-checked from the command line during development. On the small cloud server a warm check falls from 5.8 to 1.5 seconds and from 6.0 to 1.8 GB, and over ten checks in a row the average is 1.5–2.0 seconds each against 6.0. Cold, a single check falls from 104 to 34 seconds, and a series of ten from 161 to 53.

The gain is largest where the machine is small. Inside a 2 GB memory limit every snapshot-backed check completed at full warm speed, the container peaking at 612 MiB, while a plain check thrashed at 15–22% processor utilization and never finished. On that server the wrapper also beats `lean4.33.1-optimized` on this workload — 1.5–1.8 seconds warm against 3.99 — and the two cannot be stacked, since a snapshot records the environment after a complete import whereas lazy part loading leaves most of the library unloaded; with both enabled, every check fails.

The script does not help with situations such as a library build, where each module imports a different set: Mathlib's own build presents 8,705 of them, one per module. Some large projects do contain substantial potential overlap in import closures — e.g., of the 1,387 files of `formal-conjectures`, 1,149 carry the same single import line — but in that scenario a build is driven by `lake`, which invokes `lean` directly and our script's current design does not interface with. The script is also not designed to help in the interactive editor environment.

3.8 Improvement 8: compiled Mathlib tactics

We now describe an additional improvement which, similarly to the snapshot wrapper script improvement, in a technical sense does not change the Lean codebase itself and is therefore not packaged as part of the `lean4.33.1-optimized`. However, as we explain later in this section, that distinction is related to a logistical challenge we encountered in deciding how to package the improvement, and with additional work, a more finished version of the improvement could be incorporated into Lean as well, similarly to improvements 1–6.

The idea behind this improvement starts with an observation about the low-level details of how Mathlib's tactics are executed during elaboration. Mathlib's tactics are themselves Lean programs, and they are compiled when the library is built, along with everything else in it. Building a module produces, besides the files holding its definitions, two further forms of its executable code. One is an intermediate representation of Lean's own, stored in the module's `.ir` and `.ir.sig` files. The other is a translation of that same representation into C, produced by `Lean.Compiler.LCNF.emitC` (`src/Lean/Compiler/LCNF/EmitC.lean:1172`) and written out as one `.c` file per module. Mathlib's artifact cache ships both to every user.

The intermediate representation is the one that runs. Lean's interpreter reads it one instruction at a time and carries out the operation each instruction denotes; this happens inside the loop at `src/library/ir_interpreter.cpp:640-648`. The C file is not used afterwards in any way. Every call of every Mathlib tactic is therefore interpreted, and interpreted again every time it is called. Our census of where elaboration time goes, which is part of the material accompanying this paper, found that this accounts for 30% of all elaboration processor time. Interpretation produces identical results to executing the machine-native, compiled version of the same tactic, but runs much slower; this is the

savings opportunity we identified.

It turns out that machine code for those tactics can actually be used without changing either Lean or Mathlib, because the interpreter already prefers it wherever it exists: before running a function it looks for a native symbol of the same name among the shared libraries loaded into the process (`lookup_symbol` at `ir_interpreter.cpp:827`, which asks the dynamic linker through `dlsym` at `ir_interpreter.cpp:361` and remembers the answer, hit or miss), calls that symbol if it finds one, and interprets the intermediate representation otherwise. Compiling the C files that are already on disk, and loading the result, therefore replaces interpretation with machine code as an equivalent, drop-in replacement, and does not change the elaboration results. `lake` has an option, `precompileModules`, that does exactly this compilation. For a reason that we explain later in this section, this option cannot be used with Mathlib, so our improvement achieves the same result but through a different mechanism.

Our route uses only what a Mathlib checkout already contains. First, we compiled and linked Mathlib's C files, per library, which produces eight shared libraries totaling 153 MB: one for Mathlib and one for each of its seven dependencies. The build takes about fourteen minutes of wall time on the laptop, does not invalidate any compiled library files, and requires no change to any lakefile. Second, to take advantage of the compiled libraries, we needed to find a way to let Lean know about them. This can be achieved by adding eight "`--load-dynlib=...`" command-line arguments when running the `lean` executable, one for each of the compiled files. To make the process easier, and for other reasons explained below, we created a wrapper script, `lean-native-wrapper.sh`, that generates and runs the command. The command-line arguments are added in dependency order (e.g., with `Batteries` preceding Mathlib), which is essential for the run to succeed; an incorrect order results in the run terminating with an unresolved-symbol error. Within `lean`, each of those arguments leads to a call to `Lean.loadDynlib` (`src/Lean/LoadDynlib.lean:67`). We installed the wrapper script as the `lean` of a toolchain directory, so that `lake` and the editor invoke it in place of the real executable.

Using these libraries also required us to first fix a problem in Lean itself, which turned out to be a known bug: the meta-initializer crashing bug (#14359 in the tracker system) [3], whose fix is the subject of our improvement 9, described in Section 3.9 below. While developing the compiled Mathlib tactics improvement, we discovered that with the compiled libraries loaded, a file written with Lean's newer `module` header crashes deterministically with a segmentation fault: of the files we checked, one failed during the import and another failed much later inside a linter. On investigation we traced this to the same bug #14359 reported in the Lean tracker, which we were able to fix (see Section 3.9 for the details); with that fix in place, both failures are gone. Files in the traditional style, opening with a plain `import Mathlib` and no `module` header, do not cause this problem.

The `lean-native-wrapper.sh` wrapper script is more than a mere convenience: while invoking `lean` by hand with the eight arguments works for the scenario where one wants to run `lean` directly, a build is driven by `lake`, which invokes `lean` for every module. One could use the option to pass extra arguments to Lean through a lakefile's `moreLeanArgs` configuration option, but `lake` would then record those arguments in the traces it uses to decide what needs rebuilding, so naming the libraries there would invalidate the compiled files this improvement depends on leaving untouched. The solution we converged to was the wrapper script, which has the ability to stand in for the `lean` executable, receive the arguments passed from `lake`, add the arguments for the compiled libraries and then invoke the "true" Lean. This bridges the gap and makes for a functional design.

Staying outside `lake`'s traces has a price, and it is the main caveat a user of the compiled Mathlib tactics improvement should know. Because `lake` is not told about the shared libraries, there is no mechanism that can alert the user when they fall out of step with the library they were built from. A user who updates Mathlib and fetches the new artifact cache, but does not rebuild the shared libraries, is left with libraries built from the old source. Lean runs the compiled form of a declaration whenever it can find one, so any declaration whose name did not change will silently run its old implementation.

Declarations that were renamed or removed are not at risk, since the lookup fails for them and Lean interprets them as it would have anyway. `lake`'s own `precompileModules` is not susceptible to this problem, because it records the libraries in the traces that decide what to rebuild: the very mechanism we gave up in order to leave the cache intact. The caveat could be removed by adding a check that each library was built from the same sources as the compiled files it accompanies; we did not build one.

As we indicated at the start of the section, a more complete version of this improvement could live inside Lean. It would be enough for Lean, on importing a module, to look in the build directory of that module's package for the package's native library and load it if it is there, since the symbol lookup that follows is automatic already. Such a design would inherit the caveat described above, since Lean would be finding the libraries rather than being handed them, and would need the same added check we mentioned. Neither the check nor the integration into Lean seem difficult, but we did not pursue this version of the improvement in the current project.

In terms of the effect on performance, the compiled Mathlib tactics improvement yields handsome rewards. The interpreter's share of elaboration collapses by 88–94% in every group of files we measured. The precise benefits depend on how much of a given workload was interpreted to begin with, as illustrated in the table below.

elaboration processor time	interpreted before	unmodified	compiled
Fifteen textbook proof files	78%	22.9 s	6.2 s
`formal-conjectures` files	40%	69.3 s	42.6 s
Mathlib's own files	33%	219 s	156 s

The spread across the three is entirely explained by that first column. Per tactic call, those tactics which are implemented in Lean get between two and twenty times faster: e.g., `aesop` 98 to 5 ms, `linarith` 112 to 47 ms, `fun_prop` 77 to 32 ms, `norm_num` 12 to 5.5 ms. On the other hand, tactics that are implemented in C++, among them `simp`, `omega`, `decide` and `grind`, do not register a performance change, which is a useful check that the mechanism is doing what it claims. Results are identical by construction, as we confirmed on 22 cases and 1,008 lines of output, every one byte-identical.

Six scripted editing sessions per arm found median edit latency unchanged at 212 ms, which is mostly attributable to the server's 200 ms debounce interval, with `exact?`, goal display and completion latency unchanged and peak worker memory unchanged to within 10 MB. An interactive session's cost is the opening import and the debounce, not tactic execution, which is what our editor measurements had already found by a different route. The gain for this improvement is for batch work, with some example use cases including: building a project, checking a corpus, and a proof-search service.

Relation of the compiled Mathlib tactics improvement to `lake`'s `precompileModules`. We promised earlier in this section to explain why `precompileModules` cannot be used with Mathlib. The issue is that Mathlib's artifact cache is platform-independent by contract: one set of compiled files serves every user on every operating system. `lake`'s `precompileModules` makes a package's build platform-specific, and so breaks that contract. For that reason, `Aesop`'s build file disables it, with a directive in `.lake/packages/aesop/lakefile.toml:4` that reads "`precompileModules = false`" and is followed immediately by a comment: "`# We would like to turn this on, but it breaks the Mathlib cache.`"

The limitation has been noticed by users: one report on the community's Zulip board describes Mathlib's cache shipping the Linux shared libraries but not the macOS ones, so that turning the option on forces a full re-link exceeding the platform's limit on command-line length [43], and another describes a user who wanted to know what compiling their own tactic would buy, tried the option, hit a build cycle, and never got an answer [44]. Had anyone got past this obstacle and shipped compiled tactic

code by `lake`'s route, they would have met the meta-initializer crashing bug described above, since `precompileModules` loads native libraries in the same way.

Distribution of the compiled Mathlib tactics. Compiling the C files takes fourteen minutes, which is more than one would want to ask of every Lean user, so the libraries would ideally be distributed the way the rest of the build is. That can be done without disturbing the existing cache. What makes that cache platform-independent is what it holds: compiled library files and C source, which are the same on every machine. Shared libraries are not, so they need a cache of their own — one archive per library per platform, identified by operating system and processor as well as by the toolchain version and library revision the existing cache already uses, and fetched by a command of its own.

The existing cache would then be untouched, holding the same bytes for every user as it does today, and the platform-specific artifacts would live somewhere it never has to know about. A user whose platform nobody had built for would be told so and left exactly where they are now, free to compile the libraries themselves or to do without them. This is a change to how Mathlib is distributed rather than to Lean or to Mathlib itself, which is why we describe it rather than build it.

3.9 Improvement 9: the meta-initializer crashing bug

As described in the previous section, compiling Mathlib's tactics to native code, which was the subject of improvement 8, exposes a bug in Lean's C code generator. This is the already-reported meta-initializer crashing bug (#14359 in the tracker system), dating from July 2026 [3]. The bug is classified as high-priority, remains open in the tracker, and blocks [30], an otherwise merge-ready pull request improving Lean's own `precompileModules` feature, whose author notes it should not be merged until the bug is fixed. We explain here what the bug is and how we fixed it.

Compiling a Lean module to C turns the module's constants into C variables, which an initialization function fills in when the library loads. Here is one of those variables, in the C compiled from Mathlib's docstring linter, together with the only line in the file that assigns it:

```
/* unmodified v4.33.1, Mathlib/Tactic/Linter/DocString.c; lp_... abbreviates
   the compiler's long generated prefix */
LEAN_EXPORT lean_object* lp_..._deindentString___boxed__const__1;
...
lp_..._deindentString___boxed__const__1
    = _init_lp_..._deindentString___boxed__const__1();
```

That assignment never runs when the file being checked opens with a `module` header, and it does harm only when Mathlib's compiled code has been loaded as native code, the way improvement 8 loads it — run through Lean's interpreter instead, the same definitions never touch these C variables. For such a file Lean skips the initialization function that fills in constants, for any module whose ordinary code it does not expect to execute, and runs only the module's second one, the function for its compile-time code — named `meta_initialize_` in the generated C, which is where this bug gets its name (`src/Lean/Compiler/InitAttr.lean:161`). The code generator writes that second function so that it initializes the module's imports but never the module itself (`src/Lean/Compiler/LCNF/EmitC.lean:1009`). So the variable keeps the null value it starts with, and the linter, whose own code reads it, dereferences null, causing a segmentation fault crash. The filed report is about a compiler-generated specialization; the linter constant above is an ordinary definition, which shows the bug reaches any of a module's run-time declarations that its compile-time code uses, and so widens what a correct fix must cover.

Our fix to the bug was to change the code generator so that a module's compile-time initialization function also calls the module's run-time one. Calling it a second time is harmless, because a run-time initialization function returns at once if it has already run.

In Lean's code generator the change is nine lines (of which five are comment). We verified it through extensive empirical testing as well as analysis of the code. With the fix in place, the two observed failures — one during import, one much later during elaboration — do not occur in 50 runs across ten configurations, while an unpatched build fails on both. Our equivalence suite improved from 21 of 22 cases to 22 of 22, the previously failing case being the file that crashed. Of the 8,312 C files Mathlib's build emits, 8,303 differ from those emitted by unmodified Lean only by the inserted call.

Broader implications of the bug fix, and a possible improvement. A module's run-time initialization function does two jobs: it fills in the module's constants, which is what was needed to fix the crashing issue; and it runs the module's `initialize` blocks, whose effects reach outside the module, and may include, e.g., registering an option, or adding an entry to a table. The unmodified Lean's decision to skip that function, which our bug fix overrides, is about the second job and not the first: it is withholding those effects from a module whose ordinary code it does not expect to execute. Our fix makes sure the function is called, so the effects now happen where Lean had previously determined them to be unnecessary. Lean's own legacy entry point has always done exactly that, and nothing in our equivalence suite behaves differently with the fix than it does under the interpreter, so we believe the fix to be functional and correct. However, there is a possibility for a more precise, refactored version of the fix that would split the function in two, and call only the half that fills in constants. We did not pursue this version, but the Lean developers may want to consider whether it is worth implementing.

3.10 Improvement 10: fork safety, and a partial fix

The `fork` system call duplicates a running process, and the copy shares the original's memory, with any copying of memory blocks being deferred until one of the two processes writes to it. This creates an opportunity to repeatedly fork a `lean` process that has finished loading Mathlib, saving the repeated resource usage associated with loading the library when processing a multi-file project. The Lean community raised the question of whether Lean's design is actually safe to fork, and the question is currently unanswered. Two issues in the `repl` repository ask it: one has been open since April 2025, with a single speculative comment and no maintainer reply [4], and another in which it is raised three times across ten comments, again without an answer [5].

We resolved this question of whether Lean is safe to fork. The bottom-line answer, which is easy to demonstrate, is that it is *not* safe, as this can result in the child process hanging, deterministically. However, as we explain below, some measures can be taken to improve the situation.

The cause for the lack of safety is Lean's pool of worker threads (`task_manager`, `src/runtime/object.cpp:758`), which is kept together with a count of how many workers are idle. `fork` copies the count but not the threads, so the child believes workers are waiting when none exist; it dispatches work, signals them, and waits for a reply that cannot come. The result is a silent hang; worse behavior even than a crash, since the user is kept waiting with no indication of what is happening. The release package carries the demonstration as `code/scripts/demo-fork-hang.sh`: it builds a fifty-three-line C program against an unmodified toolchain and runs it three ways, hanging in the first and completing in the other two, which is what locates the cause in the pool rather than in `fork` itself.

In the `repl` discussion thread [5], a claim was made that Lean is single-threaded, and the same belief is apparently baked into the design of a published system: the Kimina Lean Server runs one Lean process per core on the stated grounds that "a single Lean REPL process is single-threaded, and its CPU usage does not exceed one core" [45, Sec. 3]. This is false. We measured 8 threads after import on a 12-core

machine, and 4 at `lean`'s lowest thread setting. There is no setting below that one: `lean -j0` does not run single-threaded, it stops with an internal panic. At least 2 threads exist before any Lean code runs at all.

Some work can nonetheless be done in a forked child of unmodified Lean. Work marked as requiring its own dedicated thread bypasses the broken pool and runs correctly. Ordinary proof checking does not use that path, but the editor server does, by policy. So it is also the case that unmodified Lean can be forked successfully in some specific situations of interest.

Motivated by our optimization-focused work, we put some effort into constructing a patch that would improve the fork-safety situation. This partial fix is the subject of the current improvement. Sixty-one lines in one runtime file add a function, `lean_reinit_task_manager_after_fork`, which sits beside the task manager's own initializer (`object.cpp:1125`) and which the child calls once, before it runs any Lean code. The function lets go of the worker handles the child inherited, without trying to join threads that no longer exist; sets the counts back to zero; rebuilds the mutex and the condition variables in place, in case the fork landed while another thread held one of them; and starts a worker if the parent left queued work behind.

With the fix, a forked child of a Mathlib-loaded process elaborated Lean correctly and byte-identically 40 times out of 40, every one of those forks taken while four of the parent's threads were in the middle of elaborating. That is a better result than we expected, and it is worth being careful about what it shows: the window in which a fork does damage is narrow, not absent, and the conditions set out below are what keep a child clear of it.

A forked child can also go on to extend the environment it inherited, taking one of 3,181 modules to 3,305 and producing byte-identical results, provided it calls `enableInitializersExecution` before importing. We measured that on an unmodified toolchain, with an existing runtime call standing in for ours, so it is a property of `fork` and of Lean's import machinery rather than anything our patch provides.

The patch removed one clear obstacle to fork-safety, but even with it in place, there remains a set of conditions both the parent and child process need to satisfy in order for forking to succeed. First, the forking program must be a custom host binary rather than the `lean` shell: a program that links Lean's shared library and calls into it directly, instead of running `lean` as a command. Only such a program can choose the moment at which to fork and carry out the child's repairs afterwards; the `lean` executable provides no way of doing either. Second, at the time of forking, the parent must be quiescent; when we tested forking in situations where a thread was mutating a shared reference at the instant of the fork, the child process hung every time, and in another test with a thread mutating a Lean object graph during forking, the child aborted every time, while forking a quiescent parent was successful. Third, in the child process, `lean_reinit_task_manager_after_fork` must be called before any Lean code runs, the random number generator reseeded, and fresh file descriptors taken if it performs input or output, since parent and child otherwise share one file offset. Fourth, for as long as it lives, the child process must stay away from Lean's asynchronous input and output layer, which aborts the process as soon as anything reaches it. Fifth, the child must finish by calling the low-level exit function `_Exit` (which Lean exposes as `IO.Process.forceExit`), rather than by returning from `main`; when we tested the latter, the child did not exit but hung indefinitely at no processor usage.

Under the conditions described above, the patched version is demonstrated safe through many correct runs and an analysis of the code and possible execution paths which may trigger a failure, though this falls short of proof of safety. Of the failure modes we identified by reading the runtime, three resisted every attempt we made to trigger them, one we could find no way to test with the equipment we had, and six were never exercised at all; we cannot rule any of them out at present. All the failures we did produce are hangs and aborts rather than Lean generating wrong answers: the one test that could have yielded silently incorrect output instead fails loudly, twelve times out of twelve.

The potential usefulness of forking. Separately from the question of fork-safety, we made some measurements to understand to what extent forking is a *useful* strategy for optimization purposes, and found a significant difference between different platforms. On Linux, forking incurs a time cost of 20–47 milliseconds. When we tested a forking-based elaboration procedure in the context of a library import that takes 5.9 seconds without forking, this brought a single `lean` check on the cloud server down to 0.25 seconds: a significant improvement.

On macOS, the result was very different: forking a process holding all of Mathlib takes 3.2–7.5 seconds, making this effectively pointless as an optimization technique.

A note on possible duplication of efforts with the Lean community. A pull request in the Lean repository (pull request #13123 [46]) retires idle worker threads after five seconds, which would have the effect of emptying the pool a child process would otherwise inherit during forking. We backported it and confirmed that waiting until idle and then forking works with no patch at all, five times out of five, against a control that hung. Thus, the pull request code appears to render our fix unnecessary. However, that change was merged into the main Lean repository on 12 June 2026 and reverted three days later; a re-landing commit exists, but sits on a branch that was never merged, so as of now the code is not part of Lean's official release: we verified against the source that the code is absent from releases v4.32.0, v4.33.0, v4.33.1 and `v4.34.0-rc2`, and from the current development branch. In considering whether to adopt our fork-safety patch, the Lean developers should take the #13123 submission into account.

Questions remaining. The fork-safety question has been resolved as it stands for the current Lean, and our patch pushes things further in the direction of improved safety. Fork-safety remains an unfinished issue: it remains open whether our patched runtime is safe in general as opposed to safe in every case we ran, and how it behaves under memory allocators other than the one Lean ships with.

3.11 Improvement 11: language-server configuration watching

This is the most minor in our set of improvements. It addresses a design issue we noticed that can cause problems in one particular usage scenario when working in the interactive editor. The issue is that the Lean language server has no mechanism for noticing that a project's build configuration has changed. It registers watchers for Lean source files and for the index files that accompany them, and only for them: the registration at `src/Lean/Server/Watchdog.lean:1624` lists two glob patterns, `**/*.lean` and `**/*.ilean`. Neither pattern covers `lakefile.toml`, `lake-manifest.json`, or the compiled library files, so a change to any of those never reaches the server at all. A `lakefile.lean` also produces no effect, but for a different reason: while it matches the glob pattern, this match causes the server to look for the files that import it, and since no files import a lakefile, the server again does nothing.

The files that go unwatched are the ones that decide which linters run, whether `autoImplicit` is on, whether `sorry` produces a warning, and which version of each dependency is used. A change to any of them reaches a file opened afterwards, because that file's configuration is computed when it opens; it never reaches a file that is already open, whose configuration is recomputed only if its own import block is edited.

In the scenario where incorrect behavior would be observed, a user changes their configuration mid-way through an editing session, with some files open, then resumes editing, and potentially opens new files. The newly opened files would be processed by the language server with the more recent configuration settings, whereas the files open before the change would be processed using the older, now-stale settings. This happens silently, without the user being made aware.

As an example, on a real project we tested with, with two files open and the `sorry` warning flipped from off to on, the file we restarted reported six `declaration uses 'sorry'` warnings and the file we

left alone reported none: same project, same moment, no diagnostic to say why. The same asymmetry appears when a dependency is rebuilt underneath the session: an open file goes on reporting the old value indefinitely, and keeps reporting it through edits to its body, updating only when its own import block is touched.

Two of the cases we measured cause a worse failure than reporting stale output. Adding a dependency to a lakefile without updating the manifest, or removing an entry from the manifest, makes every newly opened file report a single configuration error in place of its real diagnostics. For the second of these, removing an entry from the manifest, we recorded that the files that were already open carried on as though nothing had happened.

The one staleness signal the system has is a sticky diagnostic suggesting a rebuild. It is triggered by changes to Lean sources but not by changes to build outputs. As an example, the mundane occurrence of running a build in a terminal while the editor is open does not produce that diagnostic.

Our improvement adds a mechanism to the language server that prevents this problem from occurring. It works by reusing the mechanism already present, adding three additional patterns in the watcher registration, and a branch that sends the existing staleness notification to every open worker. When a Lean source changes, the server sends that notification only to the workers whose imports include the changed file; a change to the build configuration has no such set of dependents to consult, since it bears on every file in the project, so the new branch sends it to all of them. The patch is 54 lines added and 8 removed, across `Watchdog.lean`, `src/Lean/Server/FileWorker.lean` and `src/Lean/Data/Lsp/Internal.lean`.

We deliberately did not include the file that names the toolchain on the list of files to watch, because a file restart cannot pick up a toolchain change and the advice the diagnostic gives would then be false. That case requires the client to restart the server, which is a client-side action.

Limitations of the improvement. With the configuration-watching improvement in place, there are some scenarios that lead to a false positive detection (the user sees a warning of the configuration changing where one was not needed), and one in which the warning is correct but the remedy it recommends does not go far enough. For false positives, the server reacts to the notification that a file changed rather than to the content of the change, so touching a lakefile without editing it produces the diagnostic even though the configuration has not actually changed. By contrast, the Visual Studio Code extension compares contents before it prompts, and the patched watchdog could be enhanced to do likewise, but at the cost of carrying more state than the surrounding code does.

The watch patterns we add apply across the whole workspace, so an additional scenario potentially introducing false positives involves a `lake update`: re-fetching packages rewrites their lakefiles, which marks every open file stale.

Finally, there is one case in which the warning is raised correctly but the advice accompanying it is not sufficient. If the edit adds a library with a new source directory, the server's own search path is still the one it computed at start-up, so restarting the file — which is what the diagnostic asks the user to do — leaves go-to-definition resolving against the old list. Only a restart of the server itself picks the new directory up, and that, as with a change of toolchain, is something only the client can do.

Partial overlap with Visual Studio Code Lean extension functionality. The Visual Studio Code Lean extension watches some of the same files on its own account, and offers to restart the server when they are modified, so users of that editor already get some of the benefits of our improvement. Ours does not achieve anything the extension's prompt does not — both tell the user that the configuration has changed, and in both cases it is a restart that applies it — but it covers a good deal more ground. It watches lakefiles anywhere in the workspace rather than only the one at the project root; it watches `lake-manifest.json`, which the extension does not, so that a `lake update` presently gives its users no

signal at all; it asks for a restart of the files that are affected rather than of the whole server; and, being on the server side, it serves every client of `lake serve` rather than one editor.

The existence of the VS Code watching mechanism is somewhat validating of our approach, suggesting that other developers have already noticed the problem our improvement is aimed at addressing. However, we are not aware of anyone attempting to address this issue on the server side, and found no public report in any of the online trackers.

4 Summary

The work presented in this paper is of a deeply technical nature, and will require careful evaluation and analysis by the Lean community in order to determine whether and to what extent the improvements proposed here are technically sound and fit into the existing Lean development roadmap.

We recognize that the technical notes explaining our proposed improvements probably do not make for easy reading. We hope that the patient reader will be rewarded with interesting insights into the inner workings of Lean and the enormous complexity of what goes on under the hood when Mathlib modules are compiled and files are elaborated.

We have been careful to scope all our claims as precisely and honestly as possible, including acknowledging any limitations or shortcomings of the proposed improvements that we are aware of, and to provide detailed documentation to facilitate analysis of our proposals. Regardless of whether the specific optimizations proposed here will ultimately be adopted, we believe that our initial premise that the Lean toolchain provides opportunities for significant improvement remains valid. We have additional ideas for areas of improvement to consider, which we hope to explore in future projects, and invite interested researchers, particularly ones who are looking for collaborations, to contact us.

A Contents of the release package

Everything this paper describes is in the package that accompanies it, which is about 1.8 MB of patches, scripts and documents, and is laid out as follows.

```
README.md          what the package is, and where to start
AGENTS.md          the same, written as an index for AI assistants
LICENSE            Apache-2.0, covering patches/ and code/
LICENSE-DOCS       CC-BY-4.0, covering docs/ and the paper

paper/             this paper: paper.pdf and its LaTeX source

patches/           the five patches, one per separately adoptable change
code/
    tools/         leansnap.py; lean-native-wrapper.sh
    scripts/       building, checking and reproduction
    benchmarks/    the measurement harness, and where each figure came from
docs/
    improvements/ one document per improvement, 1 to 11
    building.md   switches.md   warnings.md
    evidence.md   evidence-audit.md   testing.md
    development-labels.md   what the labels in the older documents mean
```

The five patches are the fork for macOS, the fork for Linux, and improvements 9, 10 and 11, which ship separately from it. Each applies to a clean `v4.33.1` tree on its own, and all but one apply on top of

the fork as well; the exception is improvement 10, the only change outside the fork that edits a file the fork also edits. `code/scripts/check-patches.sh` verifies all of that in a few seconds without building anything.

The package carries no Lean source and no compiled binary, so using the fork means building it. Lean's own repository is cloned separately, checked out at `v4.33.1`, patched and then built, which takes about half an hour and 10 GB of disk on a first build; `docs/building.md` gives the steps and the two `cmake` flags the build requires.

The documents under `docs/improvements/` are written for a reader deciding whether a change is sound rather than merely whether it is fast. Each states the problem concretely enough to be checked, names the files and functions its patch touches, gives the correctness argument, reports what was verified and what that verification does not cover, and lists what remains provisional. They are longer and more exacting than the technical notes of Section 3, and where a document and its patch disagree, the patch is authoritative.

The package carries the benchmark suite but not our run logs, and that is deliberate: a reader should measure their own machine rather than trust ours, and absolute numbers taken here would not reproduce there in any case. What it does carry is the provenance of the figures — which runs they came from, and, for the ablation of Table 2, the switch settings that define each row — in `code/benchmarks/results/README.md`.

B The files each improvement touches

The tables below list every file that any of our modifications edits or adds, and record the correspondence between files and improvements in each direction. Table 6 is indexed by file and Table 7 by improvement; they contain the same information. The rows for the Lean source tree are derived from the patches themselves rather than from our notes, and their paths are as they appear in the Lean repository at `v4.33.1`; the two files that belong to no patch are given at their path in the release package.

Two improvements edit no line of Lean, and that is the point of them: the snapshot wrapper script (improvement 7) is a separate program that runs unmodified Lean, and compiled Mathlib tactics (improvement 8) is a way of building Mathlib. Each ships one program of its own, and those are the last two rows of each table.

File	Changed by improvements
The `lean4.33.1-optimized` fork	
`src/CMakeLists.txt`	1
`src/Lean/Compiler/IR/CompilerM.lean`	4
`src/Lean/Elab/Frontend.lean`	5
`src/Lean/Environment.lean`	2, 3, 4, 5, 6
`src/Lean/Meta/LazyDiscrTree.lean`	5
`src/Lean/Meta/Tactic/LibrarySearch.lean`	5
`src/Lean/MonadEnv.lean`	5
`src/Lean/Parser/Extension.lean`	4
`src/include/lean/lean.h`	3
`src/library/module.cpp`	1, 2, 3
`src/runtime/compact.cpp`	2
`src/runtime/compact.h`	2
`src/runtime/io.cpp`	6
`src/runtime/object.cpp`	2, 3, 4
`src/runtime/process.cpp`	1
`stage0/src/runtime/object.cpp`	bootstrap copy; see note
Shipped as separate patches	
`src/Lean/Compiler/LCNF/EmitC.lean`	9
`src/Lean/Data/Lsp/Internal.lean`	11
`src/Lean/Server/FileWorker.lean`	11
`src/Lean/Server/Watchdog.lean`	11
Also edited by improvement 10 (fork safety)	
`src/runtime/object.cpp`	10
Shipped with the package; no part of the Lean tree	
`code/tools/leansnap.py`	7
`code/tools/lean-native-wrapper.sh`	8

Table 6: Every file our modifications change or add, and the improvements that change it. Twenty distinct Lean source files: sixteen in the fork, four more in the separately shipped patches. The last two rows are not Lean source at all; improvements 7 and 8 leave the Lean tree untouched and ship a program each instead. Improvement 10 is the only one outside the fork that edits a file the fork also edits, which is why it appears twice. **Note on the bootstrap copy**: `stage0/src/runtime/object.cpp` is the copy of the runtime used to build the compiler that builds the compiler, and must carry the same changes as `src/runtime/object.cpp` for the tree to bootstrap. The patch carries it as a binary delta, so it contributes no lines to the counts of Table 3.

Improvement	Files affected

1. Address reservation	`src/CMakeLists.txt`
	`src/library/module.cpp`
	`src/runtime/process.cpp`

2. Shell-first `.olean` layout	`src/Lean/Environment.lean`
	`src/library/module.cpp`
	`src/runtime/compact.cpp`
	`src/runtime/compact.h`
	`src/runtime/object.cpp`

3. No-touch reference counts	`src/Lean/Environment.lean`
	`src/include/lean/lean.h`
	`src/library/module.cpp`
	`src/runtime/object.cpp`

4. Lazy part loading	`src/Lean/Compiler/IR/CompilerM.lean`
	`src/Lean/Environment.lean`
	`src/Lean/Parser/Extension.lean`
	`src/runtime/object.cpp`

5. Prebuilt search index	`src/Lean/Elab/Frontend.lean`
	`src/Lean/Environment.lean`
	`src/Lean/Meta/LazyDiscrTree.lean`
	`src/Lean/Meta/Tactic/LibrarySearch.lean`
	`src/Lean/MonadEnv.lean`

6. Tactic index image	`src/Lean/Environment.lean` `src/runtime/io.cpp`
7. Snapshot wrapper script	`code/tools/leansnap.py` — no Lean source; a separate program that runs unmodified Lean
8. Compiled Mathlib tactics	`code/tools/lean-native-wrapper.sh` — no Lean source; a recipe for building Mathlib
9. Meta-initializer crashing bug	`src/Lean/Compiler/LCNF/EmitC.lean`
10. Fork safety	`src/runtime/object.cpp`

11. Language-server configuration watching	`src/Lean/Data/Lsp/Internal.lean`
	`src/Lean/Server/FileWorker.lean`
	`src/Lean/Server/Watchdog.lean`

Table 7: The same correspondence read the other way: for each improvement, every Lean source file it changes. The bootstrap copy `stage0/src/runtime/object.cpp` is omitted from the individual rows, since the fork carries it once on behalf of every improvement that edits `src/runtime/object.cpp`. Improvements 7 and 8 change no Lean source at all, which is why they work with the official release exactly as downloaded.

References

[1] "VSCode slow startup", Lean Zulip, channel `#general`, 2 August 2026. `https://leanprover.zulipchat.com/#narrow/channel/113488-general/topic/VSCode.20slow.20startup`

[2] "Lean is unusably slow", Lean Zulip, channel `#lean4`, 27 May 2025. `https://leanprover.zulipchat.com/#narrow/channel/270676-lean4/topic/Lean.20is.20unusably.20slow`

[3] "Meta initializer uses runtime-initialized data with specialized instance", `leanprover/lean4` issue #14359, filed 10 July 2026; labelled `bug`, `P-high`; open at the time of writing. `https://github.com/leanprover/lean4/issues/14359`

[4] "Fork safety", `leanprover-community/repl` issue #87 (open since April 2025, unanswered). `https://github.com/leanprover-community/repl/issues/87`

[5] "Server mode", `leanprover-community/repl` issue #84. `https://github.com/leanprover-community/repl/issues/84`

[6] The Lean Focused Research Organization. `https://lean-lang.org/fro/about/`

[7] E. Chen et al., "Fel's Conjecture on Syzygies of Numerical Semigroups", arXiv:2602.03716, February 2026. The formal statement and proof were generated by `AxiomProver` from a natural-language statement of the conjecture and verified in Lean; formalizations at `https://github.com/AxiomMath/fel-polynomial`. `https://arxiv.org/abs/2602.03716`

[8] Math, Inc., "Introducing Gauss, an agent for autoformalization". `https://www.math.inc/gauss`

[9] Math, Inc., "Strong PNT": the Lean formalization of the strong Prime Number Theorem, completed with `Gauss` in response to a challenge from Terence Tao and Alex Kontorovich. `https://math-inc.github.io/strongpnt/`; `https://github.com/math-inc/strongpnt`

[10] "Prof. Viazovska's proofs of sphere packing formalized with AI", EPFL news, 11 March 2026. `https://actu.epfl.ch/news/prof-viazovska-s-proofs-of-sphere-packing-formaliz/`

[11] "Progress in Formalizing Sphere Packing in Dimension 8", arXiv:2604.23468; project repository `math-inc/Sphere-Packing-Lean`. `https://arxiv.org/abs/2604.23468`; `https://github.com/math-inc/Sphere-Packing-Lean`

[12] Harmonic, "Aristotle: IMO-level Automated Theorem Proving", arXiv:2510.01346. `https://arxiv.org/abs/2510.01346`.

[13] Harmonic, "Aristotle Achieves Gold Medal-Level Performance at the International Mathematical Olympiad, iOS App Beta Launch'. `https://www.harmonic.fun/news/imo-ios`

[14] `AxiomProver`, Axiom Math. Described, with the Lean formalizations it produced, in [7]; the group's formalizations are published at `https://github.com/AxiomMath`.

[15] Palomar — a registry of Lean-verified results. `https://palomar-registry.org`

[16] Formalizing Fermat's Last Theorem. Anthropic, September 4, 2026. `https://www.anthropic.com/research/formalizing-fermats-last-theorem`

[17] On the Navier–Stokes Millennium Prize Problem. OpenAI, September 8, 2026. `https://openai.com/index/navier-stokes-solution/`

[18] Lean 4 (source repository), the Lean FRO and contributors. https://github.com/leanprover/lean4. Version v4.33.1, commit 819816b2, used throughout this paper: https://github.com/leanprover/lean4/tree/v4.33.1

[19] mathlib4, the Lean mathematical library. https://github.com/leanprover-community/mathlib4

[20] vscode-lean4, the Visual Studio Code extension for Lean 4. https://github.com/leanprover/vscode-lean4

[21] repl, the Lean 4 read-eval-print loop. https://github.com/leanprover-community/repl

[22] Harmonic. Running Lean at Scale, Sep. 11, 2025. https://www.harmonic.fun/news/lean-at-scale/

[23] elan, the Lean toolchain installer. https://github.com/leanprover/elan

[24] "Hardware suggestions?", Lean Zulip archive, #general. https://leanprover-community.github.io/archive/stream/113488-general/topic/Hardware.20suggestions.3F.html

[25] "Laptops suitable for Lean", Lean Zulip archive, #general. https://leanprover-community.github.io/archive/stream/113488-general/topic/Laptops.20suitable.20for.20Lean.html

[26] "Computer for Lean", Lean Zulip archive, #general. https://leanprover-community.github.io/archive/stream/113488-general/topic/Computer.20for.20Lean.html

[27] MathBB. https://mathbb.app.

[28] Lean v4.34.0-rc2 release candidate. https://github.com/leanprover/lean4/releases/tag/v4.34.0-rc2.

[29] Lean v4.34.0-rc2 release notes. https://lean-lang.org/doc/reference/latest/releases/v4.34.0/.

[30] "Load missing dynlibs individually with precompileModules and initialize lazily", leanprover/lean4 pull request #14326. https://github.com/leanprover/lean4/pull/14326

[31] The release package accompanying this paper: the patches, the tools, the benchmark suite and the technical documents. https://github.com/danromik/lean-optimizations

[32] Radar, the Lean continuous benchmarking service. https://radar.lean-lang.org; https://github.com/leanprover/radar

[33] formal-conjectures, Google DeepMind. https://github.com/google-deepmind/formal-conjectures

[34] Equational Theories Project, led by T. Tao. https://github.com/teorth/equational_theories

[35] Physlib: the Lean physics library, formerly PhysLean. https://github.com/leanprover-community/physlib

[36] Fermat's Last Theorem in Lean, Imperial College London. https://github.com/ImperialCollegeLondon/FLT

[37] PrimeNumberTheoremAnd. https://github.com/AlexKontorovich/PrimeNumberTheoremAnd

[38] Carleson's theorem in Lean. https://github.com/fpvandoorn/carleson

[39] XNU, the Darwin kernel (source distribution), Apple Inc. Release xnu-10063.121.3, corresponding to macOS 14.5, the version used for the measurements in this paper. https://github.com/apple-oss-distributions/xnu/tree/xnu-10063.121.3

[40] Joel Spolsky, "Back to Basics", *Joel on Software*, 11 December 2001. https://www.joelonsoftware.com/2001/12/11/back-to-basics/

[41] "Store per-module constant prefix trees in oleans", leanprover/lean4 pull request #14362. https://github.com/leanprover/lean4/pull/14362

[42] "Lazy IR loading", leanprover/lean4 pull request #14145. https://github.com/leanprover/lean4/pull/14145

[43] K. Sullivan et al., "The clang command line with all ~15,000 Mathlib .c.o.export", Lean Zulip, channel #lean4, 17 February 2026. https://leanprover.zulipchat.com/#narrow/channel/270676-lean4/topic/The.20clang.20command.20line.20with.20all.20~15.2C000.20Mathlib.20.2Ec.2Eo.2Eexport

[44] T. Skrivan et al., "Compiling mathlib tactic", Lean Zulip, channel #lean4, 18 May 2026. https://leanprover.zulipchat.com/#narrow/channel/270676-lean4/topic/Compiling.20mathlib.20tactic

[45] Marco Dos Santos, Hugues de Saxcé, Haiming Wang, Mantas Baksys, Mert Unsal, Junqi Liu, Zhengying Liu and Jia Li, "Kimina Lean Server: A High-Performance Lean Server for Large-Scale Verification", *The 5th Workshop on Mathematical Reasoning and AI*, NeurIPS 2025. arXiv:2504.21230. https://arxiv.org/abs/2504.21230; https://github.com/project-numina/kimina-lean-server

[46] "Expire idle task pool threads after 5 seconds", leanprover/lean4 pull request #13123 (merged 12 June 2026, reverted 15 June 2026). https://github.com/leanprover/lean4/pull/13123