Real-Time LuaTeX: Recompiling Large Documents in 1ms

Clemens Lode

Abstract

This paper shows that pixel-perfect LaTeX output and real-time editing are not mutually exclusive. Line breaking is paragraph-local; page breaking is global but not perceptually immediate and therefore need not block the editing loop.

By extracting display lists directly from LuaTeX's node structures (bypassing PDF generation), a single paragraph can be recompiled and its display list delivered in ∼1 ms, using vanilla TeX Live with no engine modifications and full `microtype` [6] support.

We demonstrate that this approach is sufficient to build software that can recompile large documents in real time: `texlode`, a browser-based book authoring tool with output identical to standard LuaLaTeX. Benchmarks show that per-paragraph recompilation achieves $O(1)$ latency, constant regardless of document size, whereas Typst's [3] incremental compilation scales linearly ($O(n)$).

The tradeoff is temporary inconsistency: pages the user is not viewing may lag until a background compile converges, the same pattern word processors have used for decades.

1 The speed problem

LuaLaTeX compilation scales poorly with document size. A 100-page book may take 10–30 seconds to compile, making interactive editing (even with caching) impossible. Cloud compilation (for example, Overleaf) helps with toolchain complexity but not with the fundamental bottleneck: long documents with complex preambles, OpenType fonts, and full `microtype` settings are computationally costly.

Word processors solved this decades ago. Word, InDesign, and Google Docs recompile only the edited paragraph on each keystroke; the rest of the document stays put until a background process reconverges global layout. LaTeX has never worked this way, not because TeX's algorithms are too slow, but because its architecture treats compilation as a monolithic, file-in-to-file-out operation.

This paper grew out of a decade of providing LaTeX publishing services, over thirty client books, and the author's own *How to Publish Your Book with LaTeX* [5]. The book documents the complete self-publishing workflow using LuaLaTeX, and in doing so, documents the pain: client books with complex fonts routinely hit Overleaf's compile timeout. It recommends using draft mode during editing, then switching to a local installation for final output, forcing authors to choose between typographic quality and interactive feedback.

The question this paper addresses is not how to make LuaTeX faster, but whether full compilation is even necessary during editing.

2 Prior approaches

The LaTeX community and its neighbors have tried several strategies to make typesetting feel interactive.

TeXpresso [1] achieves near-real-time XeTeX rendering through process snapshotting. Every ∼500 ms of processing, the driver forks the engine, creating a copy-on-write checkpoint. When an edit arrives, it finds the most recent checkpoint before the change and re-runs from there. This incrementality with full reruns from optimal starting points requires a modified XeTeX engine and `fork()`, limiting it to Linux and macOS.

SwiftLaTeX and BusyTeX [7, 8] compile TeX Live to WebAssembly, enabling client-side LaTeX in the browser. But compilation still processes the full document each time.

Typst rebuilt typesetting from scratch around incremental compilation. Every element depends only on explicit inputs, enabling automatic memoization at the engine level. The result is fast (single-digit milliseconds for small documents), and the engine is clean, modern, and well-engineered. But it is inherently incompatible with LaTeX, and its incremental recompilation still scales linearly with document size.

These approaches share an assumption: that the unit of compilation is the document. They differ in how to make that unit cheaper: better checkpointing, smarter memoization, faster engines. None asks whether full compilation is necessary in the first place.

3 The measurement

How long does LuaTeX actually need to typeset a paragraph? We can find out by wrapping paragraph text between two `\directlua` timing calls, as in Figure 1; running the file with `lualatex` prints the time to the terminal.

On a mid-range laptop (Intel Core i7-1355U, TeX Live 2025), this consistently reports under 1 ms. Because a single compile is often faster than the resolution of `os.gettimeofday()`, a more systematic benchmark amortizes the timer — each sample is the mean of 100 consecutive in-session compiles — and

```
% paragraph-benchmark.tex
\documentclass{book}
\usepackage{microtype}
\begin{document}
\directlua{t0 = os.gettimeofday()}%
The quick brown fox jumps over the lazy dog.
\par
\directlua{
  local ms = (os.gettimeofday() - t0) * 1000
  texio.write_nl("Paragraph: "
    .. math.floor(ms * 100 + 0.5) / 100 .. " ms")
}
\end{document}
```

Figure 1: A self-contained single-paragraph timing example; run with `lualatex` and read the terminal output.

Table 1: Per-paragraph line-breaking time by paragraph type in vanilla LuaLaTeX (median, P5, and P95 over 30 amortized samples).

Paragraph length	Median	P5	P95
Short (1 line)	0.17 ms	0.03 ms	0.96 ms
Medium (4–5 lines)	0.70 ms	0.65 ms	0.89 ms
Long (10+ lines)	1.99 ms	1.87 ms	2.41 ms
Inline math	0.20 ms	0.17 ms	0.27 ms
Display math	0.11 ms	0.10 ms	0.13 ms

reports median, P5, and P95 over 30 such samples after a 5-sample warmup:

The dominant cost is line breaking, not content complexity: a one-line paragraph with inline math (0.20 ms) is faster than a plain five-line paragraph (0.70 ms). Display math is particularly fast because it produces a single-line box with no break-point search.

4 Why one paragraph is enough

A single paragraph is a meaningful unit of work because of a structural property of TeX, evident in the Knuth–Plass line-breaking algorithm [4]:

Line breaking is local. Given the same `\hsize` and content, TeX produces identical line breaks regardless of what appears before or after the paragraph. Changing one paragraph does not affect how any other paragraph breaks into lines.

Page breaking is global. A paragraph's height changes can cascade through the rest of the document: subsequent paragraphs may shift to different pages, floats may move, page numbers may change.

This separation is exactly what word processors exploit. During editing, the user is looking at one paragraph on one page. The only operation that changes what they see at that moment is line breaking, and line breaking is local. Page breaking matters, but it does not need to happen on the same keystroke. A background process of the word processor can reconverge global layout while the user continues typing.

Table 2: Median per-paragraph compile time in each window. No degradation has been observed; the variation between windows is noise, the variation between columns reflects paragraph length.

Compile window	Short	Multi-line
1–50	0.12 ms	1.89 ms
226–275	0.09 ms	1.84 ms
451–500	0.09 ms	1.93 ms

The challenge for LaTeX is that TeX was never designed to work this way. Word processors have built-in paragraph renderers that can be called as functions. TeX is a batch processor: it reads files, processes the entire document, and writes output. There is no API for "compile this one paragraph and return the result".

Our engineering contribution, `texlode`, adapts LuaTeX to this pattern: keeping the engine alive, feeding it individual paragraphs, and extracting results from internal structures that were not meant to be an output format.

To verify that compile time is truly independent of document state, we ran 500 consecutive paragraph compiles in a single LuaTeX engine session:

The locality claim has boundaries. Footnotes couple a paragraph to page breaking; counter-dependent content (`\ref`, `\thepage`) requires global state; `\parshape` set by surrounding code is unavailable in isolation. These cases fall to the background convergence path, the same full compile that resolves page numbers and float placement. The fast path handles body text, which is where the user spends most editing time and where latency matters most.

5 From measurement to real-time editor

Compiling only the edited paragraph and converging global layout in the background is how every modern word processor works. The engineering challenge is adapting LuaTeX's batch-mode architecture to support it.

Keeping the engine alive. LuaTeX startup takes ∼1 second for a book-class preamble with OpenType fonts. We maintain a persistent LuaTeX process that accepts paragraph content, compiles it, and returns the result without restarting.

Capturing output without PDF. Generating a PDF for a single paragraph would add significant overhead. Instead, we extract the display list directly from LuaTeX's node structures after line breaking: a stream of positioned glyph commands with coordinates in scaled points. The

Table 3: Time breakdown for a single per-paragraph round trip.

Pipeline stage	Short	Medium
LuaTeX (line break + traversal)	0.65 ms	4.99 ms
IPC + serialization	0.12 ms	1.03 ms
Deserialization + expansion	0.02 ms	0.09 ms
Total round-trip	0.79 ms	6.11 ms

traversal must correctly replicate TeX's internal position calculations, including all `microtype` adjustments, to produce output identical to standard LuaTeX compilation. TeX was never designed to expose its internal rendering state; extracting a pixel-identical display list from LuaTeX's post-line-break node structures required reverse-engineering internal conventions with no prior precedent.

Rendering in the browser. The glyph stream is sent to the browser and drawn to HTML5 Canvas via `opentype.js`. Each glyph command maps to a Canvas path at the coordinates specified by the engine. Rules (filled rectangles) and color operations complete the primitives.

Background convergence. A full document compile runs periodically, producing a complete page-position cache. We overlay fast-path paragraphs at cached positions, so the user sees live line breaking with correct global layout. Page numbers, cross-references, and float placement converge through this path, typically within seconds of an edit.

A pipeline breakdown shows where time is spent: Over 80 % of the time is LuaTeX's line-breaking algorithm, the useful work. The engineering overhead stays under 20 %.

`texlode`, the browser-based book editor built on this architecture, is scheduled for public release in October 2026. It provides collaborative editing via conflict-free replicated data types (CRDTs), manuscript import from Word, proceedings management, cover design tools, and print-ready PDF output identical to standard LuaLaTeX. The accompanying conference talk demonstrates the editor in action. For details: `texlode.com`.

6 Comparison to Typst

Typst's incremental compilation is fast, but its architecture reconverges global layout on every edit. To measure the cost, we edit the same medium paragraph at the middle of documents of increasing size, compiling with Typst 0.14.2 in watch mode (its standard interactive editing workflow) and recompiling only that paragraph in LuaLaTeX:

Table 4: Both engines edit the same medium paragraph. The LuaLaTeX column is constant: only the edited paragraph is recompiled, so its cost equals the single-paragraph time from §3, independent of document size. The Typst column grows linearly because Typst reconverges the full document layout.

Document size	Typst 0.14.2	LuaLaTeX
10 pages	12.6 ms	0.70 ms
100 pages	76 ms	0.70 ms
300 pages	206 ms	0.70 ms

Typst reconverges global layout by design: it guarantees globally consistent output on every run. But a visual text editor like `texlode` does not need that guarantee on every keystroke, it only needs the paragraph the user is looking at to be correct now, and the rest of the document to converge soon. Typst's 206 ms for a 300-page book corresponds to roughly 5 fps, usable but visibly laggy for smooth drag interactions.

Human-computer interaction research identifies 100 ms as the threshold below which responses feel instantaneous [2]; one display frame at 60 Hz (16 ms) is the budget for continuous visual feedback. At 0.1–2 ms per paragraph, LuaLaTeX sits inside the frame budget: layout stops being an artifact you regenerate and becomes a continuous quantity you can manipulate. Text reflows around a figure in real time as you reposition it, table contents are readjusted in real time as you change column widths, line breaks settle keystroke by keystroke, and adjustments to leading or font size produce smooth, immediate feedback, all with full `microtype` support in every update, not approximated or deferred.

Benchmark scripts for the vanilla-LuaLaTeX paragraph timing and the Typst comparison are available at `github.com/texlode/luatex-benchmark`.

7 Conclusion

Per-paragraph reflow during editing is standard practice; word processors have worked this way for decades What was missing was the implementation for LuaTeX. Adapting LuaTeX to the interactive pattern required engineering: persistent engines, display list extraction from node structures never designed as an output format, browser rendering, background layout convergence. But the line-breaking algorithm that Knuth designed is fast enough to run on every keystroke. We just had to stop asking it to compile entire documents. Authors of book-length LaTeX documents no longer have to choose between typographic quality and interactive feedback.

References

[1] F. Bour. TeXpresso: Live rendering and error reporting for LaTeX. *TUGboat* 44(2):185–192, 2023. `tug.org/TUGboat/tb44-2/tb138bour-texpresso.pdf`

[2] S.K. Card, T.P. Moran, A. Newell. *The Psychology of Human-Computer Interaction.* Lawrence Erlbaum Associates, 1983.

[3] M. Haug. Fast typesetting with incremental compilation. Master's thesis, Technische Universität Berlin, 2022. `user.tu-berlin.de/mhaug/fast-typesetting-incremental-compilation`
`pdf`

[4] D.E. Knuth, M.F. Plass. Breaking paragraphs into lines. *Software — Practice and Experience*, 11(11):1119–1184, 1981.

[5] C. Lode. *How to Publish Your Book with LaTeX.* LODE Publishing, 2026. `www.lodepublishing.com/book/latex-book-publishing-in-2026/`

[6] R. Schlicht. *The `microtype` package: Subliminal refinements towards typographical perfection*, 2024. `ctan.org/pkg/microtype`

[7] SwiftLaTeX. SwiftLaTeX, the visual LaTeX editor. `github.com/SwiftLaTeX/SwiftLaTeX`

[8] TeXlyre. BusyTeX: TeX live compiled to WebAssembly, 2026. `github.com/TeXlyre/texlyre-busytex`

◇ Clemens Lode
Düsseldorf, Germany
`clemens (at) lodepublishing dot com`
`https://www.lodepublishing.com/`