

TDOM: LaTeX Authoring Without the Compile Wait Verified Incremental Typesetting over a Live Document Tree

Kosei Kawaguchi Takumi Kokawa

Fermion Inc.

`contact@fermion.company`

August 18, 2026

Abstract

Editing a $\text{\LaTeX}$ document still requires recompiling the entire file before any change can be inspected, and the delay grows with the document. This paper presents TDOM (TeX Document Object Model), a typesetting runtime that carries over to unmodified LuaTeX the transition the Web once made from page reloads to incremental re-rendering over the DOM. Because the $\text{\TeX}$ macro language can alter its own lexical rules during execution, no syntax tree of the source can be defined statically; TDOM therefore keeps the execution itself rather than a parse of it. The document is split into blocks, and a LuaTeX process that has typeset everything up to each block boundary is kept resident, halted, by `fork(2)`. An edit resumes the process sitting just before it and feeds it only the changed block; counters, macro definitions and font state are already held by that process. The range affected is settled by a bounded check, while page breaking, floats, footnotes, cross-references and the table of contents converge incrementally rather than awaiting a full recompilation. Agreement between the display and genuine LuaLaTeX output is checked mechanically by four independent procedures, including layout comparison at 0.1 bp precision against PDF content streams and an equivalence test between incrementally maintained and freshly built engine states under randomized edits; constructs whose agreement cannot be established fall back to display taken from the real output, at line, block or document granularity. In our evaluation, steady-state keystroke response stays at or below 18.2ms (p50) and below 100ms (p95) on documents of 4 to 30 pages, largely independent of document size.

1 Introduction

Writing with $\text{\LaTeX}$ repeatedly involves recompiling the entire document after every edit, opening the resulting PDF, and finding the affected passage. Compile time grows with document size; for a package-heavy book of roughly 100 pages, a single compile has been reported to take 10–30 seconds[1]. Each wait is short in isolation, but placing it between every modification and inspection makes the writing loop itself heavy. Authors must either postpone checking their work or interrupt their train of thought whenever they do check it. The more a field depends on the typesetting quality for which $\text{\LaTeX}$ has no substitute, the more persistent this gap between editing and display becomes.

The history of the Web provides a precedent for removing the same kind of gap. Web pages in the 1990s were reloaded wholesale from a server after each action. With the standardization of the Document Object Model (DOM)[12], a document instead became a tree of objects retained inside the browser: when a script modifies the tree, the layout engine recomputes and repaints only the affected region. Replacing page reloads with tree mutation did more than make pages faster; it established the foundation for interactive Web applications.

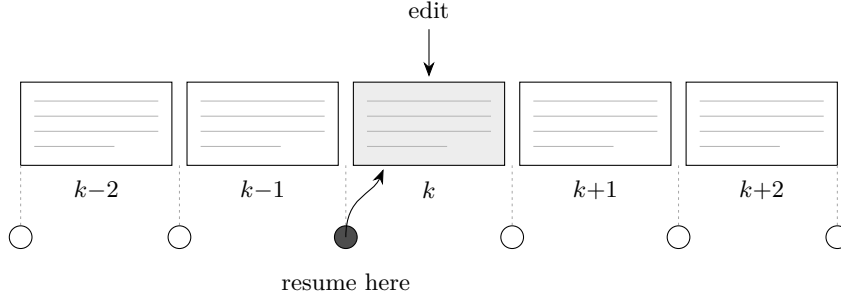


Figure 1: The document is divided into paragraph-sized blocks. At every boundary, a LuaTeX process that has finished typesetting up to that point is kept resident and halted (circles). The circle immediately before block k is a process that has finished blocks $1..k-1$. When block k is edited, that process resumes and reads only block k . Because it retains counters, macro definitions, and font state, its result agrees with a whole-document run.

Measurements also show that the typesetting algorithm itself is not the source of $\text{\TeX}$'s delay. Knuth–Plass line breaking[3] completes in about 1 ms per paragraph[1]. The expensive part is instead the disposable process model: read the input, process everything, write the output, and discard all internal state. As long as that model remains, neither caching nor parallelism removes the dependence of latency on document size. If the document could remain inside a running runtime and an edit could be treated as a mutation of retained state, editing latency could become interactive without changing typesetting quality.

This paper proposes TDOM (TeX Document Object Model), which realizes that architecture on LuaTeX. The Web DOM, however, cannot be transferred literally to $\text{\TeX}$. The role of every input character—escape character, group opener, math shift, or ordinary character—is determined by a value called its catcode, and these values can change during execution. After `\makeatletter`, `@` is read as a letter; inside a verbatim environment even `\` becomes an ordinary character. The same string can therefore mean different things depending on what has run before it, so a syntax tree cannot in general be obtained by statically parsing the source[4]. TDOM addresses this constraint by associating an execution state, rather than a parse result, with each tree node (Figure 1). The document is divided into paragraph-sized blocks, and the LuaTeX process that has finished typesetting through block k remains resident in a halted state. That process is the node. Its macros, counters, fonts, and catcodes are all still alive, eliminating any need to enumerate, serialize, and restore state. The replication mechanism is `fork(2)`, the Unix facility that duplicates a running process. The two copies share memory until either writes to it, so duplicating a large process has little immediate cost.

Independent work has recently begun to retypeset only edited material. Lode[1] observes that line breaking is independent of the context outside a paragraph and measures about 1 ms when a resident LuaTeX process retypesets only the edited paragraph. That result supports the premise of this work, but detaching the paragraph from its document context constrains the approach. Section numbers and cross-references depend on prior execution and do not take the fast path; updates to page breaks, floats, footnotes, and page numbers must await a periodic full compile. Nor does the system verify that its fast display agrees with genuine $\text{\LaTeX}$ output. This work addresses those three points: retaining context, incrementally converging global layout, and mechanically verifying output.

This paper makes four contributions. First, it presents an architecture that retains a document's execution state as a sequence of halted processes by loading only 71 lines of C into stock LuaTeX (Section 4). Unlike `TeXpresso`[2], which modifies the engine, it follows both existing packages and future `TeX Live` updates without an engine fork. Second, it presents

Table 1: Prior approaches to real-time L^AT_EX and TDOM

	Engine changes	Retypeset unit	Context	Immediate global layout	Verification
latexmk, etc.	no	whole document	—	full compile	none
TeXpresso[2]	yes	edit onward	yes	by re-execution	none
SwiftLaTeX/BusyTeX	no	whole document	—	full compile	none
Typst[6]	new engine	expression, $O(n)$	yes	full convergence	none
Lode[1]		paragraph, $O(1)$	no	waits for full compile	none
TDOM (this work)	no	bounded blocks	yes	yes	four procedures

incremental retypesetting that stops after verifying where the effects of an edit end (Section 4). Synchronous work checks a fixed number of blocks and then stops; any remainder converges after the author pauses, so cross-references, the table of contents, page breaks, floats, and footnotes can update without a full compile. Third, it makes display correctness itself an object of verification (Section 5). Only lines judged to agree are drawn as text; other lines are replaced with images from real output, and four independent procedures mechanically compare the display with actual L^AT_EX output. Fourth, measurements show that steady-state keystroke response is nearly independent of document size and stays below 100 ms at every tested size and edit location (Section 6).

2 Background

2.1 The disposable process model

A conventional L^AT_EX run is a one-shot batch process. `lualatex` reads the source from beginning to end, typesets it, emits a PDF, and exits without retaining internal state. Starting the process and loading the format—a snapshot with macro definitions already loaded—takes roughly 0.3 seconds; processing the preamble takes another 0.1–5 seconds, before body typesetting even begins. The first part is a fixed cost independent of the edit. Because all internal state is discarded on every run, the next run pays it again.

Multiple passes add another cost. Page numbers for cross-references and tables of contents are unknown until the document has been typeset. L^AT_EX therefore writes reference data to an auxiliary (aux) file on the first run and reads it back on the second. Correct output needs at least two whole-document compiles, and three if the table of contents changes pagination. Repeating this process after every edit leaves present-day L^AT_EX writing in the same kind of loop as the Web’s former page reload.

2.2 Related work

Table 1 summarizes the principal prior approaches. TeXpresso[2] modifies the XeTeX engine, uses process copies as checkpoints, and re-executes from the edit onward. It shares the idea of retaining execution state as a process, but requires modifications to the engine. SwiftLaTeX[8] and BusyTeX[9] port T_EX to WebAssembly and run it in the browser, changing distribution and startup while still compiling the whole document after every edit. Typst[6, 7] is a new typesetting system that incorporates incremental computation into the language design, but it is not L^AT_EX compatible and its incremental compile time has been reported to grow linearly with document size[1]. As described above, Lode[1] retypesets a paragraph at a time on stock LuaTeX, but does not retain context, incrementally update global layout, or verify output.

3 Design

The name TDOM comes from the Web DOM. Whereas the DOM stores the result of parsing HTML, TDOM stores execution state, a difference forced by the properties of $\text{\TeX}$ described above. Otherwise the editing flows are isomorphic: when part of the tree changes, only the changed nodes and the region they affect are recomputed, and only the difference is sent to the display. Table 2 gives the correspondence.

Table 2: Correspondence between the Web DOM and TDOM

	Web browser	TDOM
Internal representation	DOM tree (parse result)	block sequence plus halted $\text{\TeX}$ processes (execution state)
Unit of change	node or attribute	block (paragraph boundary; environments are not split)
Effect tracking	style invalidation and dirty flags	typeset-result hash, post-block value tuple, and dependency table
Relayout	recompute the affected region	bounded synchronous work plus convergence after idle
Repaint	repaint the difference	per-page display-list difference
Final output	rendering engine itself	ordinary LuaLaTeX output (canonical tier)

The design follows three principles.

First, retain context. A block is never detached from the document and typeset alone. Block k is typeset by a child copied from the process that finished block $k-1$, and it is typeset on the genuine main vertical list—the vertical sequence of completed lines before page breaking. Counters, macro definitions, font state, previous depth, and the rules for combining interparagraph glue are therefore the same as in a complete run. This is not an approximation: the process has actually executed the preceding document.

Second, verify the display. The fast display is used only where it can be judged to agree with real $\text{\LaTeX}$ output. A line that cannot be so judged is replaced with an image from the genuine output. Any region later found to disagree is automatically demoted. The discipline permits a display to be temporarily stale, but not to show something different from the real result.

Third, fall back progressively. No fast path can cover 40 years of packages. Unsupported constructs fall back at three levels: line (render a mathematical line as an image), block (typeset only that unit separately), or document (show only real output). At every level the author can continue editing.

4 Architecture

4.1 Checkpoint construction

Call a LuaTeX process that has finished processing through block k “checkpoint k .” When an edit arrives, the nearest checkpoint preceding the edit is copied. The child reads and typesets the edited block, then becomes the next checkpoint. The source process remains halted, so the following edit can reuse the same position.

This architecture requires only a way for LuaTeX to duplicate a process. The resident parent starts as an ordinary `lualatex` process. A `\directlua` command, which executes Lua during typesetting, loads 71 lines of C. That code exposes only a few system calls, including `fork`,

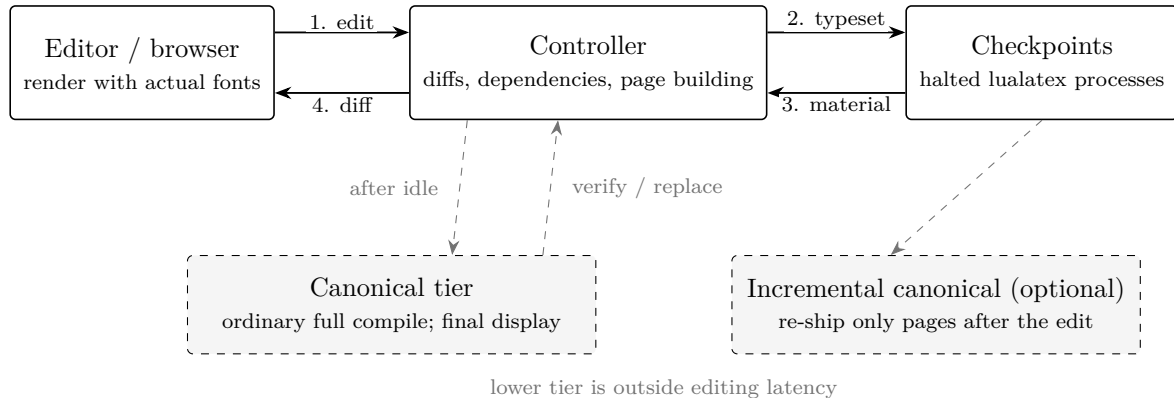


Figure 2: Overall flow. Only steps 1–4 run on each keystroke, and the round trip takes on the order of tens of milliseconds: (1) an edit arrives; (2) the preceding checkpoint is copied to typeset that block; (3) typeset material is returned; and (4) only the display lists of changed pages are sent back. The two components below are outside editing latency. They run after the author pauses, replace the display with genuine $\text{\LaTeX}$ output, and provide the reference for verification.

`waitpid`, and `_exit`; neither the engine executable nor the $\text{\LaTeX}$ kernel is modified. Loading shared libraries is normally disabled, so only the parent startup requires `--shell-escape`.

The number of checkpoints is capped because each is a real process and can occupy hundreds of megabytes. To remain under the cap, checkpoints are spaced uniformly through the document (64 by default, and eight in the measurements below), while those immediately around the active edit position are retained regardless of spacing. Coarser spacing is compensated by retypesetting the intervening blocks on resume, at a cost of about 3 ms per block.

4.2 Extracting typeset material

The result of a block is extracted without passing through PDF. Its source is $\text{\TeX}$ ’s main vertical list itself. Page building is suspended during this stage: `\vsize`, the target height of a page, is set to its maximum so that a page never fills, and `\holdinginserts=1` keeps footnotes in the document flow. The system then receives the completed line boxes, glue including stretch and shrink orders, penalties (the costs assigned to candidate page breaks), footnote bodies, and float markers. Lua $\text{\TeX}$ ’s `node.effective_glue` reports the actual amount by which glue has stretched, so the resulting character coordinates agree with those in the shipped PDF.

The critical point is that material is extracted after line breaking but before page breaking. Glue and penalties—the inputs to page-breaking decisions—are still intact, allowing the next stage to repeat page breaking under the same criteria as $\text{\TeX}$.

4.3 Building pages

For every edit, JavaScript applies the work corresponding to $\text{\TeX}$ ’s page builder and the $\text{\LaTeX}$ output routine—the macros invoked when a page fills to assemble body text, footnotes, and floats—to the extracted material. The score used to choose a page break is a direct port of Knuth’s implementation (tex.web §108). The implementation handles float placement, footnote assembly, top-of-page spacing, running heads and folios, parity adjustment for `\cleardoublepage`, and insertion of blank pages. Every parameter affecting placement is measured from the running $\text{\TeX}$ process; this layer introduces no independent dimensions.

Intermediate state is stored at every page boundary, so page recomputation is limited to

pages near the edit. Whereas Lode[1] delegates page-break updates to periodic full compiles, TDOM updates page breaks in the same response as the keystroke.

4.4 Determining the affected range

Each block stores a hash of its typeset result and a tuple of values after the block. The tuple contains every counter value, the depth of the preceding line, whether a page break was prohibited immediately before the boundary, and the amount of trailing glue. After an edit, following blocks are retypeset until both the result and tuple exactly match their previous values. Once they match, the edit cannot affect anything farther downstream.

The check is bounded: it stops after eight blocks when layout is coupled and after four when only values are propagating. At the limit, one of three outcomes applies. If equality has been established, the work ends. If only values such as counters are changing, number propagation is deferred. If downstream meaning may have changed—for example, after a macro definition changes—the remaining retypesetting is deferred. Deferred work begins after 300 ms without an edit; a new keystroke immediately interrupts it, and it later resumes from that point.

Cross-references do not pass through auxiliary files. A $\text{\LaTeX}$ cross-reference is implemented as a macro named `\r@label`; defining it directly in the running process eliminates the write-and-read cycle. A dependency table records which blocks refer to which labels, and when a label changes, only its referring blocks are retypeset. The table of contents is handled similarly: when page numbers move, the block that uses the table of contents is retypeset, converging in at most three iterations.

4.5 Display eligibility and fallback to real output

To draw text in the browser, the system serves the exact font files used by $\text{\TeX}$, disables browser kerning and ligatures, and reconstructs positions within a line from the font advances. Some lines still cannot be guaranteed to agree: lines containing mathematics and lines containing raw PDF operations, such as TikZ output. Mathematical fonts are commonly in legacy formats that a browser cannot read, and pieces of extensible symbols occupy private-use areas that cannot be addressed by character code. Mathematics is therefore detected not by font name but from boundaries recorded when $\text{\TeX}$ enters and leaves math mode. Every line derived from math is included, regardless of its font. Such a line is replaced by an image made from genuine output. A child copied from the checkpoint at that position ships just that region as a one-page PDF, which is then converted to an image.

Environments that intervene in page construction—including `multicols`, `longtable`, and breakable `tcolorbox`—and those that inspect the remaining page height to place themselves—including `wrapfigure` and rotated floats—fall back at block rather than line granularity. They are typeset independently in another process, with their starting position on the page reproduced to 0.25 bp (1 bp is 1/72 inch), while the genuine output routine decides placement and splitting. A construct that changes the whole page design, such as replacing the output routine or enabling two columns, causes document-level fallback to real output only. Editing remains uninterrupted at every fallback level.

4.6 Canonical tier and incremental canonical output

The final display is always the ordinary output of running `lualatex` until its auxiliary files stabilize; call this the canonical tier. A canonical compile is not merely excluded from editing latency—it does not run between every pair of edits. The first baseline is built after a short delay (2.5 seconds by default). Thereafter a compile runs only after both a sustained idle period (30 seconds by default) and an interval proportional to the previous compile time, or

Table 3: TDOM verification procedures. Each uses ordinary LuaLaTeX output as its reference.

Procedure	What is compared	Precision	When run
Layout comparison (verify-layout)	baselines of every line, read directly from genuine PDF content streams	0.1 bp	required in CI on a 12-document corpus diagnostic
Page-break comparison (compare-breaks)	material and dimensions emitted whenever the genuine output routine runs, against this page builder	0.02 bp	
Incremental-fresh equality (fuzz)	every block after random edits, incremental state against a freshly opened engine	exact	CI with fixed seeds
Incremental-canonical comparison (shipping)	text on every re-shipped page against a full compile	per page	CI

when PDF export is explicitly requested. Since the fast path provides immediate display, the canonical tier need only verify the document at a natural pause and provide the reference for the next section. On completion, unchanged pages are replaced by L^AT_EX’s own output and verification runs.

An optional second resident execution uses the genuine output routine and leaves a checkpoint whenever it ships a page. On an edit, it resumes from the last checkpoint before the first changed block and re-ships only the later pages. Updating the authoritative output then scales with the number of pages after the edit rather than with a full compile. We call this incremental canonical output.

5 Verification

To make “the display agrees with the real output” a testable condition rather than an assertion, TDOM provides four independent verification procedures (Table 3).

The central condition is equality between incrementally maintained state and a freshly opened state. After any edit sequence, every block’s typeset result and value tuple must match those obtained by opening the final source in a new engine. Random edits with fixed seeds check this condition after each group of edits. During development these tests found, among other bugs, incorrect recording of reference values and page-position-dependent blocks that had become stuck in stale states. A source that cannot compile, such as incomplete T_EX during typing, has no correct state to converge to. In that case the last valid state is frozen to prevent numbers from oscillating, and the system automatically returns when the source becomes valid. The same tests cover both freezing and recovery.

Verification also operates at runtime. After every canonical compile, the text on screen is compared with the text in the real output. Similarity is computed from character bigrams, so the same rule applies to Latin and Japanese text. Only blocks with a clear disagreement are automatically demoted; those blocks are thereafter displayed as real images or canonical output rather than drawn as text. A browser font-load failure is likewise reported back, causing lines using that font to switch to image display. Verification is not an external test around the system, but information used inside it to decide what and how to display.

Table 4: Document scale and edit response. A keystroke inserts one character; entries are from the worst of three locations with 30 keystrokes each. “Steady” excludes the first keystroke after choosing a location; “first” is that first keystroke. Full compile is two measured `lualatex` runs.

	S (4 pp.)	M (20 pp.)	L (30 pp.)
Full compile (2 runs)	1.1 s	1.1 s	1.0 s
TDOM startup (once)	2.0 s	3.1 s	3.1 s
Steady keystroke p50	16.6 ms	13.9 ms	18.2 ms
Steady keystroke p95	19.5 ms	73.6 ms	41.5 ms
First keystroke after moving	20.2 ms	92.6 ms	141 ms
Insert section (later numbers move)	43.1 ms	90.1 ms	98.5 ms
Change macro definition	20.6 ms	41.9 ms	35.5 ms

6 Evaluation

6.1 Setup

Measurements were taken on an Apple M5 with 16 GB of memory, using LuaHBTeX 1.18.0 (TeX Live 2024). We prepared English `article`-class documents at three scales: S with 4 pages and 49 blocks, M with 20 pages and 241 blocks, and L with 30 pages and 361 blocks. Each section contains four paragraphs, one numbered equation, a forward reference, and a backward reference. The preamble uses only `amsmath`, a light configuration near the lower bound for whole-document compile time; real documents will generally produce a larger difference than the one below.

The engine was started with a cap of eight checkpoints. This is much coarser than the default of 64 and saves memory at the cost of slowing the first keystroke after moving the edit position, as discussed below. It is therefore a configuration unfavorable to the proposed method. The canonical tier and image generation are outside editing latency by design (Section 4) and were disabled for this measurement. We inserted one character 30 times at each of three locations—25%, 60%, and 92% through the document—and recorded the distribution of synchronous response time. For comparison, we compiled the same document conventionally with two `lualatex` runs on the same machine. The measurement scripts and raw data are published with the implementation[14]. Here p50 is the median and p95 excludes the slowest 5% of observations.

6.2 Results

Table 4 gives the results. Steady-state keystroke latency was nearly independent of document scale. Although L is seven times the size of S, p50 ranges from 16.6 to 18.2 ms, while the worst p95 is 73.6 ms. Every scale and edit location remains below the 100 ms guideline for interactive response[5]. In contrast, even these deliberately light documents take about one second to compile in full (1.1–1.0 s). The ratio is approximately 50 even under this lower-bound condition. Since real documents can take 10–30 seconds[1], their gap is wider. The time previously paid on every edit becomes a one-time startup cost of 3.1 seconds for L.

The cause of the slow tail is identifiable. A single first keystroke after moving the edit position takes up to 141 ms (L, near the middle). That keystroke must retypeset intervening blocks from a coarsely spaced checkpoint; afterward the position remains fixed and latency returns to its steady value. The distance retypeset is inversely proportional to the checkpoint cap, so at the default of 64 the distance would be roughly one eighth of that in this measurement. The observed tail is not a fundamental limit, but a configurable tradeoff between memory and

first-keystroke response.

Edits with a large affected range also respond in bounded time. Inserting a section changes every following section number and reference, yet responds in 98.5 ms for L; number propagation is deferred and does not block the edit. Changing a macro definition is a worst case requiring downstream retypesetting, but its synchronous response is 35.5 ms and retypesetting converges in the background. Perceived responsiveness depends on more than the mean, making a bound on worst-case edits at least as important as average performance.

Finally, we ran the verification procedures under the same configuration. The invariant tests for the editing path passed 20/20. Across a 12-document layout corpus containing mathematics, floats, footnotes, references, mixed Japanese text, and environments typeset in a separate process, 318/331 baselines matched. Eleven documents matched on every line; the remaining document was added while this paper was being written and still has a discrepancy.

6.3 Relationship to prior systems

Lode[1] reports 0.1–2 ms to retypeset an isolated paragraph and Typst incremental compile times of 12.6, 76, and 206 ms for 10-, 100-, and 300-page documents, respectively, on that paper’s measurement system. TDOM’s keystroke response belongs to the same size-independent class as the paragraph result, but includes a different scope of work: context retention via process duplication, page breaking, float and footnote placement, and reference tracking. It handles the global layout excluded from Lode’s fast path while remaining in the same millisecond range. This is the principal measured result of the paper.

7 Applications

Managing a document as a live tree of execution states has consequences beyond reducing wait time. We identify three directions that build on this architecture.

First is direct editing. Every TDOM display command retains the source block and range from which it came. Any character on screen can therefore be mapped back to its source range. Combined with millisecond editing response, this supports word-processor-style editing directly on the typeset result, with typesetting following in place and $\text{\LaTeX}$ quality preserved. Mathematical lines likewise map images to line-level source ranges, allowing an author to select an equation, edit its $\text{\TeX}$ representation, and inspect verified typesetting through the same mechanism. The verification layer of Section 5 addresses the conventional weakness of WYSIWYG editing: a mismatch between the screen and final output.

Second, the architecture structures context for large language models. Today a model generally receives either a raw `.tex` string or a PDF, each of which loses part of the document’s structure. TDOM’s representation has block-level structure, relations describing what each block references and affects, an exact change set for every edit, and a bidirectional mapping between source and display coordinates. It can send only relevant blocks together with their typesetting state, apply a model’s proposal as an edit, and return typesetting and verification results in milliseconds. Batch compile latency has kept this kind of iteration impractical.

Third, display-list differences can be distributed to several viewers at once, providing a basis for collaborative editing. Incremental canonical output, whose print-identical pages follow edits, can also shorten proofreading loops in publishing workflows.

8 Limitations

The current implementation has six limitations.

First, it does not support `microtype`, which makes characters protrude slightly into the margin and subtly expands or contracts widths. In documents using it, coordinates of lines drawn as text can differ slightly. Detecting and falling back from this case, as well as full support, remain future work.

Second, two-column layout, packages that replace the output routine, and packages that draw over an entire page cause document-level fallback to genuine output only, while editing remains available.

Third, footnotes that split across pages are not supported. When a split would be required, the page builder approximates by breaking the page early; the canonical tier shows the final result.

Fourth, resident process groups depend on process duplication, so native Windows, which lacks this facility, is not supported.

Fifth, because checkpoints are real processes, memory use scales with document size and the configured cap. Under the evaluation configuration, the sum of apparent resident memory across the process group is about 748 MB. This figure double-counts memory shared with the source process, so effective use is lower.

Sixth, the depth of a single duplication lineage is limited. In a preliminary experiment with 120 sections, 721 blocks, and 60 pages, typesetting slowed as the lineage deepened and reached the default 12-second timeout. Startup took 331 seconds. The raw data are included in `measurements-1120.json`. The evaluated range of 361 blocks lies below this limit.

The second and third limitations fit within the existing progressive fallback framework. The first can be addressed by extending the text-rendering layer, the fourth by a degraded mode based on independent compiles, and the sixth by periodically rebuilding checkpoints to keep lineages shallow.

9 Conclusion

This paper has presented TDOM, a typesetting runtime that retains a document as a sequence of execution states, and has demonstrated in implementation and measurement that stock LuaTeX can provide millisecond editing latency nearly independent of document size. Edits are processed as block-level retypesetting; page breaks, floats, footnotes, cross-references, and the table of contents update without awaiting a full compile. Four verification procedures mechanically check agreement between the display and genuine output, while constructs that cannot be verified progressively fall back to real output. Even under a light-preamble condition close to a lower bound, evaluation measured a p95 steady keystroke of 41.5 ms against a 1.0-second full compile. For real documents whose full compiles take 10–30 seconds[1], the difference exceeds three orders of magnitude.

This reduction does not come from changing or approximating the typesetting algorithm. It follows from replacing the disposable process model with a state-retaining runtime. Just as making a Web document a live tree enabled interactive applications, the implementation and measurements in this paper show that the same transition is possible for typesetting.

References

- [1] C. Lode. Real-Time LuaTeX: Recompiling Large Documents in 1 ms. *TUGboat*, draft, 2026.
- [2] F. Bour. TeXpresso: Live rendering and error reporting for L^AT_EX. *TUGboat*, 44(2):185–192, 2023.
- [3] D. E. Knuth and M. F. Plass. Breaking Paragraphs into Lines. *Software—Practice and Experience*, 11(11):1119–1184, 1981.

- [4] D. E. Knuth. *The T_EXbook*. Addison-Wesley, 1986.
- [5] S. K. Card, T. P. Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.
- [6] M. Haug. Fast typesetting with incremental compilation. Master’s thesis, Technische Universität Berlin, 2022.
- [7] Typst. <https://typst.app/> (accessed August 2026).
- [8] SwiftLaTeX: The visual L^AT_EX editor. <https://github.com/SwiftLaTeX/SwiftLaTeX>.
- [9] TeXlyre. BusyTeX: T_EX live compiled to WebAssembly. <https://github.com/TeXlyre/texlyre-busytex>, 2026.
- [10] H. T. Thành. *Micro-typographic extensions to the T_EX typesetting system*. PhD thesis, Masaryk University Brno, 2000.
- [11] R. Schlicht. The `microtype` package: Subliminal refinements towards typographical perfection. <https://ctan.org/pkg/microtype>, 2024.
- [12] W3C. Document Object Model (DOM) Level 1 Specification. W3C Recommendation, 1998.
- [13] LuaTeX development team. *LuaTeX Reference Manual*. <https://www.luatex.org/>.
- [14] TDOM Engine (source code, verification suite, and measurement scripts). <https://github.com/Fermion-company/tdom-engine>, 2026.