Idle Time Spell Checking

https://goo.gl/zONC3v
PUBLIC (Google internal version is here)
Status: Current
Author: xiaochengh@chromium.org (Xiaocheng Hu)
Last Updated: 2017-06-11

Objective

We propose idle time spell checking to improve code health and also reduce latency in typing and other editing operations.

Currently, Blink's spell checker is triggered synchronously at every editing operation (text, selection and focus change), blocking the page for some time after every operation, figuring out which sentences and/or words need to be checked. Think about editing a very long article -- after typing every single character, the page freezes for a while before the character appears -- to which spell checking can contribute nontrivially (as shown in Figure 1).

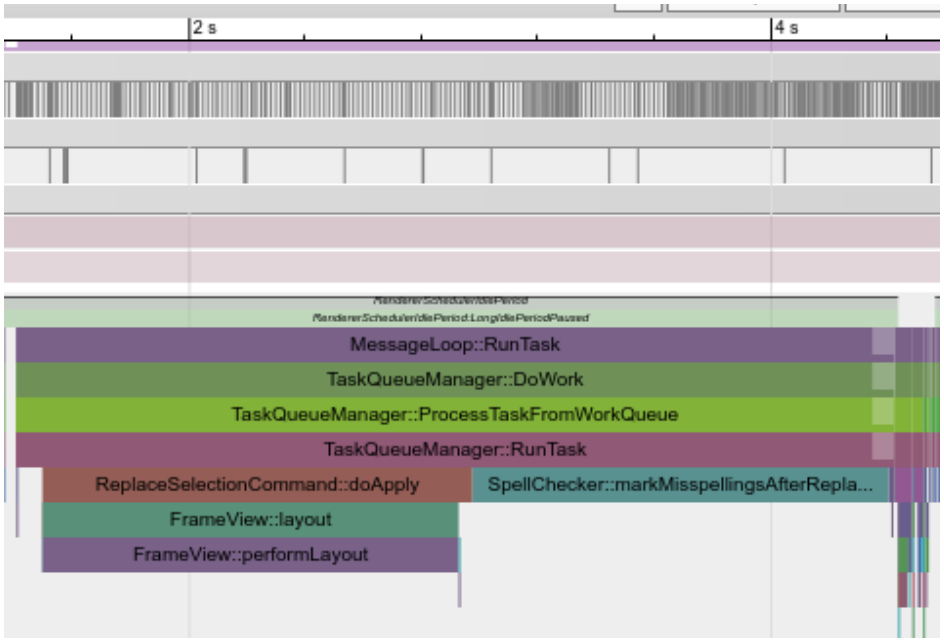

Figure 1. Trace of pasting a big text chunk into an empty <textarea>

Idle time spell checking, as suggested by its name, completely eliminates any synchronous spell

checking work after an editing operation. Instead, we utilize `requestIdleCallback` to invoke spell checking at the browser's idle time. As a result, the performance overhead of spell checking can be greatly reduced, and the possibility of decoupling editing and spell-checking code is open.

Background

WHATWG HTML Specification

The spec defines the usage of the `spellcheck` attribute, controlling whether the content of a editable or form control element should be spellchecked. When the attribute is missing (or has an invalid value), Chrome adopts the inherit-by-default behavior, namely, whether the content of an element should be checked is the same as its parent, which may ultimately trace back to the global default, which is true.

Spell checking can also be globally disabled via context menu or Chrome Settings. In this case, no element should be spell checked.

Workflow of Blink's Spell Checker

Figure 2 illustrates the current workflow of Blink's spell checking.

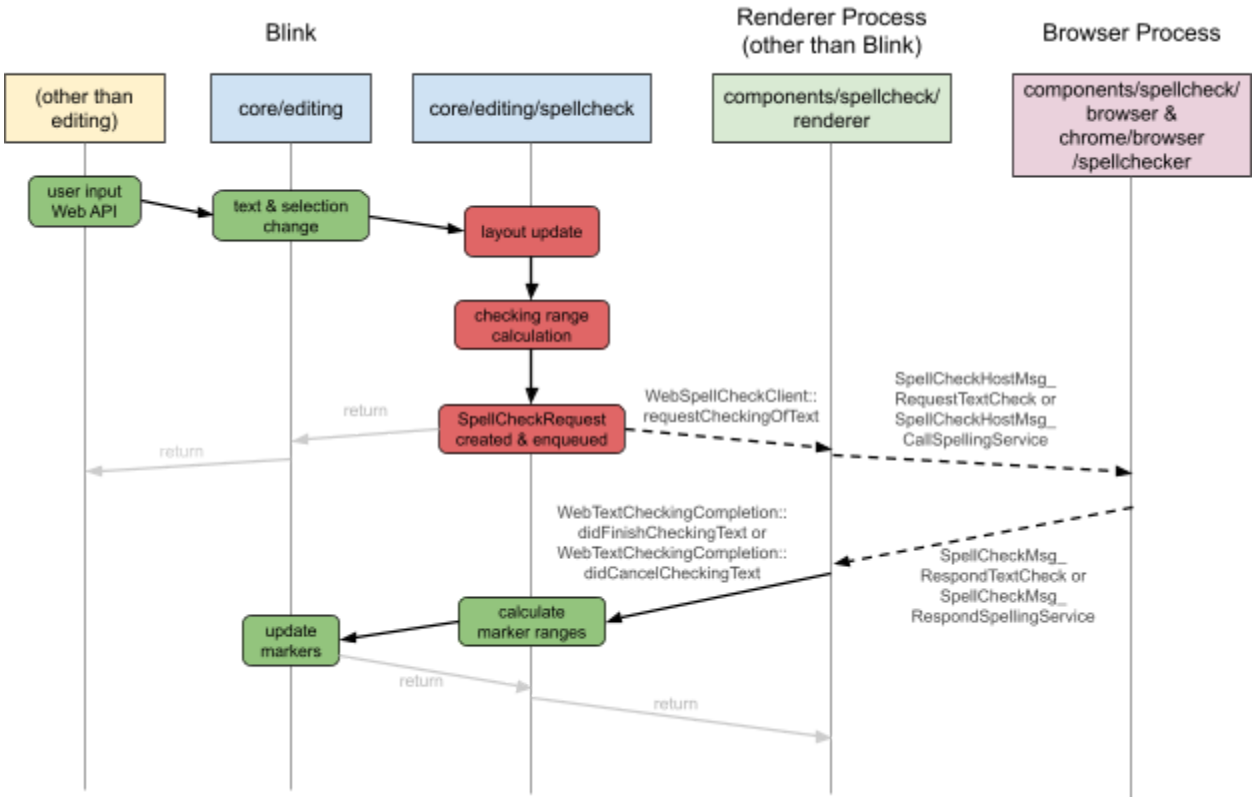

Figure 2. Workflow of Blink's spell checking

The spell checker does not always go through all the synchronous spell-checking step (marked in red) of the above diagram. At any step, it may decide that there is no need to check and abort there (e.g., after focusing an empty text field). Nevertheless, if it finally creates a new `SpellCheckRequest`[1], it has to go through all the steps.

Blink's spell checker can perform heavy synchronous computation because:

- If the operation ends up with dirty layout, a synchronous layout update is forced.
- Identifying the range of text to check can be expensive if the page is large.

`window.requestIdleCallback`

We assume that the reader has sufficient knowledge about the `requestIdleCallback` API.

Blink's undo and redo stacks

In order to support undo and redo on editing operations, Blink maintains an undo stack for each frame. Each entry of the stack is an `UndoStep`, which tracks an editing operation that has been done. More specifically, an `UndoStep` stores the starting and ending selection of an editing operation, the input type and a sequence of basic DOM changes (node insertion, node deletion, text node modification, etc.) that constitute the operation.

In other words, the undo stack provides sufficient information about past editing operations.

Overview

Idle time spell checker is not synchronously triggered by editing operations, but instead, invoked at the browser's idle time as a callback requested by `requestIdleCallback`. When invoked, spell checker should figure out by itself what to check, sticking to the following two principles:

1. Responsiveness: The part of page under active editing should be checked promptly, offering the user an interactive experience with spell checking during editing.
2. Completeness: Given a sufficient amount of idle time, all elements in the page that require spell checking (as described in section WHATWG HTML Specification) should be ultimately checked, guaranteeing a bottom line functionality of idle time spell checking.

Viewed differently, idle time spell checker generates `SpellCheckRequests` at idle time, which is asynchronous to the editing operations, while the existing spell checker does the same job synchronously with each editing operation.

---

[1] A `SpellcheckRequest` is basically a `Range` in the DOM tree to be checked, together with a `String` of the plain text extracted from the `Range` to be sent to the browser process.

The handling of each `SpellCheckRequest` is not affected by idle time spell checking, and hence, will not be discussed in the remaining sections of this design doc.

Detailed Design

Since this project is performance oriented, there is a lot of room for us to try different heuristics in order to get the best performance. Related discussion will be isolated out into section Heuristics. There will also be a number of parameters whose values can be tuned, which will be summarized in section Tunable Parameters.

We will introduce a new class `IdleSpellCheckCallback`, which is a subclass of `IdleRequestCallback`, containing most of the implementation of idle time spell checking. Each `LocalFrame` should own an `IdleSpellCheckCallback`, which is responsible for checking content inside the frame.

Lifecycle

Figure 3 shows a state machine for the lifecycle of an `IdleSpellCheckCallback`.

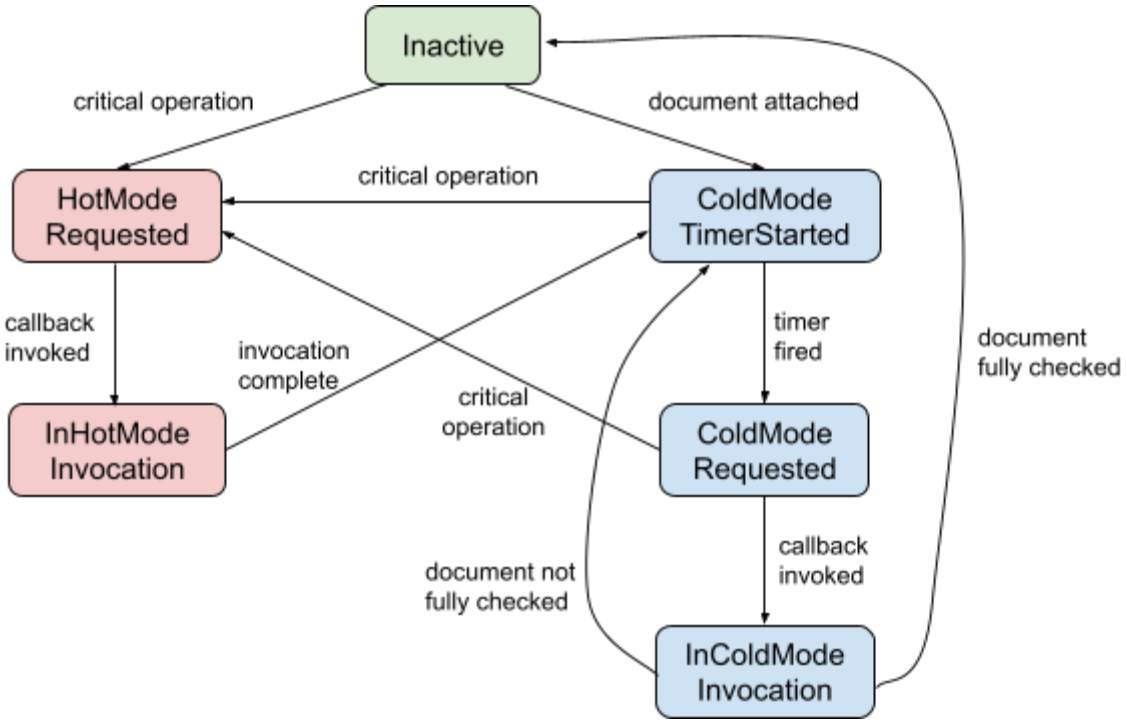

Figure 3. Lifecycle of an `IdleSpellCheckCallback`

Detailed description of the states is as follows.

Inactive

The callback has no activity at this state, and is waiting for external triggering:

- When a document is attached to the `LocalFrame`, proceed to **ColdModeTimerStarted**;
- If any critical operations occurs, proceed to **HotModeRequested**. See section Critical Operations for details of which operations are critical.

HotModeRequested

We enter “hot mode” only if some critical operation was recently performed, and for which spell checker must be responsive. We call `requestIdleCallback` to request invocation of the callback (unless we arrived here from **ColdModeRequested**), with the `timeout` parameter being a tunable parameter. We proceed to **InHotModeInvocation** when the callback is invoked.

InHotModeInvocation

The callback is invoked, and `SpellCheckRequests` may be generated to check affected content of some recent operation. See section Invocation in Hot Mode for details of the generation of the requests. At the end of the invocation, we move to **ColdModeTimerStarted**.

ColdModeTimerStarted

We would like to perform a complete check of the entire document, but we should do this only when the document is idle for a sufficiently long time. For this purpose, we set and start a timer, whose timeout is a tunable parameter. After that:

- If any critical operation occurs before the timer is fired, move to **HotModeRequested** and cancel the timer.
- If the timer is successfully fired (which means no critical operation happened when waiting for the timer), move to **ColdModeRequested**.

ColdModeRequested

We enter “cold mode” only if there is no critical operation for a sufficiently long time, in which case we would like to progressively check the full document, so that checking completeness is guaranteed. `requestIdleCallback` is called for the callback. We intentionally do not set any timeout for the request, since responsiveness is not the objectiveness of cold mode.

If any critical operation occurs before the invocation, we switch to **HotModeRequested** for responsiveness. If we remain at **ColdModeRequested** when the callback is invoked, we proceed to **InColdModeInvocation**.

InColdModeInvocation

The callback is invoked, and `SpellCheckRequests` may be generated to check some regions of the document. See section Invocation in Cold Mode for details. At the end of the invocation, if the document has been fully checked by this and previous cold mode invocations, we go back

to **Inactive** state. Otherwise, we move to **ColdModeTimerStarted** for checking other parts of the document.

Invocation in Hot Mode

Layout update

Spell checking requires clean layout because it needs to extract plain text content from the document. In most cases, `IdleSpellCheckCallback` should be invoked already with clean layout, because idle request callbacks are invoked at the end of the rendering pipeline where layout has already been updated. Nonetheless, it is possible that other idle request callbacks have made the layout dirty again by modifying the document[2]. In this case, we may:

- Abort, with the hope that next time the callback can be invoked with clean layout;
- Or force a synchronous layout and proceed to the remaining steps.

We need to balance between reducing the overhead of forced layout and ensuring the functioning of spell checking. Detailed strategies will be discussed in section Decision on Forcing Layout Update.

SpellCheckRequest generation

The first step is to identify the editing hosts under active editing. The followings are candidates:

1. The frame's selection currently exists and is in an editable element, then the editing host of the element may be under active editing.
2. For each `UndoStep`, the editing host of its ending selection, if any, may be under active editing.

We will discussion in section Filtering of Candidate Editing Hosts detailed strategies for deciding which candidates are under active editing. The filter also helps us prevent duplicate checking.

For each editing host surviving the filtering, if its content is short, then all of its content is requested for checking. if the content is long, we identify the part of content that is present in the viewport, extend bi-directionally by a certain length and request checking for the extended range [3]. The threshold between short and long, and the length for bi-directional extension are tunable parameters.

There is one exception: when in the middle of typing, we do not want markers to appear immediately under a partial word. For example, we do not want to see marker under "appl" if we are in the middle of typing "apple". In this case, "appl" should not be requested for checking. We will discussed in section Deciding Typing Progress on how to decide if we are in the middle of typing.

---

[2] It is allowed, but discouraged, to modify the DOM tree or style in an idle request callback. See this post.
[3] We will also ensure that final checking range is aligned to sentence boundaries to provide sufficient context for the spell checking work.

We may not have enough time for `IdleSpellCheckCallback` to check all the editing hosts or even decide if each of them is under active editing. In this case, candidate 1 has the highest priority to be processed; For the candidates in 2, a recent `UndoStep` has higher priority than an old `UndoStep`.

Invocation in Cold Mode

Cold mode invocation also requires clean layout. If the callback is invoked with dirty layout, we go through the same process as in hot mode.

The objective in cold mode is to progressively process the full page and check every element that requires spell checking. For this purpose, `IdleSpellCheckCallback` remembers the last processed element in the previous invocations in cold mode to serve as the starting point in the current invocation. The processing stops if the full page is checked or the time limit for the idle callback is exceeded.

When processing an editing host that has a lot of content, we should not generate a gigantic request for the full content, because handling of big requests is expensive -- the browser process needs a lot of time to process, and then the renderer may also need a lot of time processing the potentially big checking result. Instead, we should divide the content into multiple chunks of moderate lengths and request checking of each chunk separately[4], where the chunk length is a tunable parameter.

Heuristics

All strategies in this section are tentative and may be improved in the future.

Critical operations

Critical operations are considered to have occurred only if the following functions are called:

- `Editor::respondToChangedContent`
- `Editor::respondToChangedSelection`

Decision on forcing layout update

At the current stage, we always force a layout update if the layout is dirty.

Filtering of candidate editing hosts

Each candidate is filtered if and only if either of the followings hold:

- The candidate's editing host is not in the viewport
- The candidate's editing host is the same as an unfiltered candidate of higher priority

[4] Similar to the hot mode, chunks will be further extended to sentence boundaries.

Deciding typing progress

We decide that we are in the middle of typing if all of the following hold:

- The current selection is caret in an editing host, adjacent to or inside a word;
- `Editor::lastEditCommand` is an open `TypingCommand`, and its ending selection is the same as the current selection.

Tunable parameters (a.k.a. magic numbers)

We tentatively set the values of all tunable parameters as follows:

- Each hot mode `IdleSpellCheckCallback` is requested with 200ms timeout.
- For the timer set in **ColdModeTimerStarted**, its timeout is 1000ms if we entered the state from **InHotModeInvocation**, and 200ms otherwise.
- An editing host whose content is no more than 1024 characters is deemed short.
- The visible part of an editing host is extended bidirectionally to 1024 characters.
- The chunk length for cold-mode checking is 16384 characters.

All these parameters are hard-coded and will be hand-tuned if the performance needs improvement. We do not have plan to introduce mechanisms for auto-adjustment at this stage.

Caveats

Fixed: Duplicate checking of typing progress (crbug.com/635504)

Blink's `SpellCheckRequest` is handled by the `SpellcheckProvider` class (in `components/spellcheck/renderer`) before being sent to browser. `SpellcheckProvider` may decide that we are in the middle of typing a word, and thus, rejects the request directly. While the rationale is good, the checking should be done only in Blink, not in `components/`. It sometimes even over-rejects requests that should be processed, for example, when pasting "wellcome. a" (without quotes) into an empty text field, "wellcome" is not marked, because the request is rejected.

We should remove such duplicate checking from `SpellCheckProvider`.

Cold mode power regression

We should improve/justify the extra power consumption due to code mode's scanning the full document in the background.

Performance Measurement

New dynamic benchmark

Since the motivation of this design doc is most about reducing jank in editing, we consider the scenario of performing a sequence of editing operations in a page, and measure:

- **Response latency**. For each editing operation that is expected to change (add/remove) any spelling marker in the viewport, we measure the latency between the operation start and the expected marker update is observed. If the expected marker update cannot be observed, the latency of the operation is infinity -- this can happen, and is sometimes expected, if the page is too busy.
- **Overall processing time**. We measure the wall clock time from the starting of the first editing operation to the last marker change.

New static benchmark

Given a static page without any editing operation, we measure:

- **Complete checking time**: The wall clock time from page load to the moment when all misspellings are marked.

Note that this metric applies to idle time spell checker only. The existing spell checker cannot be triggered without editing operations.

Existing benchmarks

TODO: Convert layout test editing/spelling/spellcheck-huge-text.html into a performance test

We expect to see improvement in basically all tests in `PerformanceTests/Editing`.

UMA

We will watch the following UMAs in finch trials. The feature can be shipped as long as it does not introduce regression to any of the metrics.

- **WebCore.ScriptedIdleTaskController.IdleCallbackOverrun:** the number of overrun idle tasks. Optimizations are needed if we significantly increase the percentage of overrun tasks.
- **SpellCheck.api.async**: the number of characters sent by each request. We should not send out significantly more text for checking, otherwise optimization is needed.
- **Event.Latency.EndToEnd.KeyPress**: the time from key press (or some equivalence on Android) to resulting frame. The UMA was added recently (crbug.com/660833). We expect reducing the end-to-end input latency with idle time spell checking.

- WebCore.SpellChecker.RequestInterval: the timer interval between consecutive requests.
- SpellCheck.ShownSuggestions and SpellCheck.ReplacedWords: these two metrics together measures the effectiveness of the spell checker.

Release Plan

We will first land the implementation behind runtime enabled feature `IdleTimeSpellChecking`. In addition, due to the different nature of hot and cold modes, we will ship hot and cold mode separately, for which purpose cold mode will be implemented behind a dependent flag `IdleTimeColdModeSpellChecking`.

We will first start a finch trial in M60 for hot mode only, and ship it if there is no regression to the watched metrics.

Cold mode needs substantial improvement. Hence, its release will not be covered by this doc.

Document History

Date	Author	Description	Reviewed by	Signed off by
2016/11/17	xiaochengh	Draft round 1	yosin	yosin
2016/12/14	xiaochengh	Major revision on lifecycle		
2017/03/14	xiaochengh	Match the current implementation		

Appendix (personal notes, random thoughts, ...)

Another approach

Instead of looking at the undo stack, we may also let the `IdleSpellCheckCallback` log all the changes for which spell checking is required. Specifically, these are the changes that calls method of current `SpellChecker` class. In this way, we are essentially moving the synchronous work of `SpellChecker` to idle time. This approach leads to the smallest code change and basically keeps the current spell checking behavior, at the cost of requiring extra memory to maintain the change log.

Stale marker removal (crbug.com/705180)

Stale marker removal (`SpellChecker::updateMarkersForWordsAffectedByEditing`) is also a synchronous time-consuming operation. We may also try to move it to somewhere asynchronous, or even try to remove it with idle time spell checker.

Besides, the function seems over complicated, and the code paths entering it is not clear -- it can be called before editing, or after, or even both. Proper refactoring is needed. Note: the function was introduced back in 2010 by this patch (with a different name `Editor::removeSpellAndCorrectionMarkersFromWordsToBeEdited`), at which time the function is already made to be called both before and after editing.

Solution:
https://docs.google.com/document/d/1RVnBQxwIe_Tfx0bVHrM0bdUeRSJiG2OGuPgsKIO2gOI

Decision on whether a range should be checked or not

Currently, Blink decides whether a range should be checked by simply looking at whether spell checking is enabled at the starting position of the range. While being a reasonable economic heuristic, it leaves room for both overchecking and underchecking. See crbug.com/371567.

While overchecking is merely a performance issue (as long as we don't overmark), underchecking clearly breaks the expected behavior. Anyway, we might want to fix it.

One possible solution is to propagate the spellcheck attribute in the DOM tree, so that the decision can be made faster without much performance overhead.

DONE: Validity check in `SpellChecker::markAndReplaceFor`

On completion of a `SpellCheckRequest`, `SpellChecker::markAndReplaceFor` is called to add markers to the document. However, the function assumes that there is no document change since the creation of the request, or at least no change in the checking range. If change did occur, markers can be added to incorrect parts of the document. This might be one of the sources of crbug.com/179639

The function itself is mostly legacy code inherited from the age of fully synchronous spell checker. Necessary refactoring has to be done.

DONE: Performance optimization of `SpellChecker::markAndReplaceFor` (crbug.com/668408)

The function is called on completion of a spell check request, blocking the main thread for:

- Updating document markers
- Triggering a new frame since the original frame is invalidated

We were observing long running time of this function, but the observation is gone after an optimization in `DocumentMarkerController::addMarker`.

Anyway, the function still has quadratic running time, so we might still observe slowness in the future. If that really happens, we need to do further optimization. We might also try moving it to idle time too, but we need to be very cautious about invalidating the current frame in an idle request callback. Or we may want better scheduling of this function so that it's done in a better place in the pipeline before paint, or maybe the scheduler may put such paint-invalidation tasks together somewhere in/out of the pipeline.

Marker removal when switching `spellcheck` or `contenteditable` attribute value to `false`

Consider we already have a <div contenteditable spellcheck> with misspellings marked. If we change its `contenteditable` or `spellcheck` value to false, the markers are still there, which is not desired. See crbug.com/155781

We may

- Remove the markers when those two values are changed
- Change the markers to some invisible type. This helps restoring the markers when the relevant attribute values are changed back
- Do nothing to the markers. Instead, let painting code decide whether a misspelling marker should be painted. This has minimum performance impact, but introduces extra code dependency

DONE: Signal for spell checking complete (crbug.com/674819)

We may want to expose a signal, at least to layout tests, of the complete of spell checking. Currently, all layout tests simply repeatedly wait and check of marker existence, which is inefficient and error-prone.