Spineless Traversal for Layout Invalidation

ANONYMOUS AUTHOR(S)

Latency is a major concern for web rendering engines like those in Chrome, Safari, and Firefox. These engines reduce latency by using an *incremental layout algorithm* to redraw the page when the user interacts with it. In such an algorithm, elements that change frame-to-frame are marked dirty; only the dirty elements need be processed to draw the next frame, dramatically reducing latency. However, the standard incremental layout algorithm must search the page for dirty elements, accessing a number of auxiliary elements in the process. These auxiliary elements add cache misses and stalled cycles, and are responsible for a sizable fraction of all layout latency.

We introduce a new, faster incremental layout algorithm called Spineless Traversal. Spineless Traversal uses a more computationally demanding priority queue algorithm to avoid the need to access auxiliary nodes and thus reduces cache traffic and stalls. This leads to dramatic speedups on the most latency-critical interactions such as hovering, typing, or animations. Moreover, thanks to numerous low-level optimizations, we are able to make Spineless Traversal competitive across the whole spectrum of incremental layout workloads. As a result, across 2216 benchmarks, Spineless Traversal is faster on 78.2% of the benchmark, with a mean speedup of 3.23× concentrated in the most latency-critical interactions such as hovering, typing, and animations.

Additional Key Words and Phrases: Web browsers, incremental, order maintenance, latency

ACM Reference Format:
Anonymous Author(s). 2024. Spineless Traversal for Layout Invalidation. 1, 1 (November 2024), 20 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Latency is a major concern for modern web rendering engines such as those used in Chrome, Safari, and Firefox. A rendering engine must redraw the page 60 times per second to guarantee smooth animations, fluid interactions, and prompt responses as users browse and interact with web pages. When this frame rate cannot be met, the user experiences lag and may be forced to use another web application, browser, or device. Moreover, demand for low-latency rendering is only increasing with modern 120 Hz displays.

Layout is a key driver of web rendering latency. Layout means calculating the size and position of each element of the web page, after which the page can be rendered into pixels on the screen. Every time the user interacts with the web page by hovering over an element, receiving updated data, or even observing an animation, the web page, and thus the tree of HTML elements that represent it in memory, changes. To show the updated page to the user, the page must then be re-laid-out. Since the browser has many tasks besides layout, such as JavaScript execution, this re-lay-out step must be completed in a sub-millisecond budget in order to meet the 60 frame-per-second goal. On such a tight budget, every cycle counts!

Incrementalization. The key optimization that makes this possible is *incrementalization*. When an element on the page changes, the browser *marks* all dependency of that element as dirty. Then, when the next frame must be drawn, the layout algorithm traverses the node tree to find and

XXXX-XXXX/2024/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

re-lay-out only the dirty nodes. In the process it might need to mark additional nodes dirty, but typically—in, say, animations or interactions—few layout nodes are ultimately affected. In these cases, searching the tree for dirty elements is the bottleneck, especially since every node access is likely to incur a cache miss.

To address this issue, the state-of-the-art "Double Dirty Bit" algorithm adds summary bits to each node to avoid searching subtrees without any dirty nodes. While this reduces the search time, this algorithm still has to traverse the tree, starting from the root, to find dirty nodes. This means it must access not only the actually-dirty nodes but many extra "auxiliary nodes": the root node, every node between it and a dirty node, and any node adjacent to that path. On large pages, there can be far more auxiliary nodes than actually-dirty nodes, especially for the most latency-sensitive interactions. Since each node access introduces its own cache miss, simply traversing these auxiliary nodes (just checking their summary bits) stall the layout algorithm for hundreds of microseconds. Indeed, this problem has been widely observed in practice. For example, Google's widely-used web performance debugging tool, Lighthouse, uses tree depth and maximum children count as performance metrics, precisely because these parameters determine the number of auxiliary nodes.

Spineless Traversal. We introduce *Spineless Traversal*: a new, faster algorithm for incremental layout. Unlike the standard Double Dirty Bit algorithm, Spineless Traversal accesses only dirty nodes, not auxiliary nodes, and therefore reduces cache misses. To do so, Spineless Traversal stores the set of dirtied nodes in a priority queue and jumps directly from one dirty node to the next, with no auxiliary nodes in between.

The key to making this work is maintaining the correct traversal order. Recomputing a single field on a single node mark all fields that depend on it, and the set of transitive dependencies is complex. Fields must therefore be recomputed in a specific order, and Spineless Traversal must respect that order as it jumps from node to node. Spineless Traversal enforces this using an *order maintenance* data structure. Order maintenance assigns a logical time to each field on each node, and provides an efficient way to check which logical time comes first. The priority queue then uses this order to recompute node fields in logical time order.

Evaluation. We implement Spineless Traversal in a new DSL for browser layout algorithms, Megatron. To match the extremely tight latency budgets of modern browsers, Megatron uses a variety of low-level optimizations, including unboxing, a custom allocator, pointer compression, and an optimized branchless implementation of order-maintenance comparison. We implement a fragment of web layout in Megatron, including line breaking, flex-box, intrinsic sizes, and other complex features. We then evaluate both the double dirty bit and spineless traversal algorithms, using this implementation, on 2216 incremental layouts of 50 real-world web pages, including Twitter, Discord, Github, and Lichess.

Spineless Traversal is 3.23× times faster on average. Moreover, these speedups are concentrated in the most latency-critical changes with the fewest dirty nodes: on the 60.7% of benchmarks where fewer than 1% of fields are recomputed, spineless traversal achieves a speedup of 5.85× or more. Spineless Traversal is only slower than the double dirty bit algorithm on 21.8% of benchmarks, with most of these representing less-latency-critical events like page loads or navigations.

2 WEB LAYOUT

Web pages are written in HTML, which is a tree-structured markup language containing text and elements that wrap it. The web browser's parsed representation of this tree is called the DOM or HTML tree. To draw the page to the screen, the browser applies a sequence of transformations—called rendering phases—to this DOM tree: matching, styling, layout, paint, and so on. The focus of this paper, the layout phase, applies to an intermediate tree structure called a "layout tree", whose

```
# real web layout much more complex
def layout_simple(self):
  self.width <-
    if parent? then max(0, parent.width - 10)
    else 50 # screen size
  children.forEach(layout_simple)
  self.height <-
    if last? then last.height_acc + 10 # padding
    else self.attribute[height]
  self.height_acc <-
    if prev?
    then prev.height_acc + self.height + 10 # margin
    else 0
```

Fig. 1. A layout algorithm that computes width and height of each dom node. The above program is a tree traversal: it walks down the tree then walks up the tree, computing values for each node during the walk.

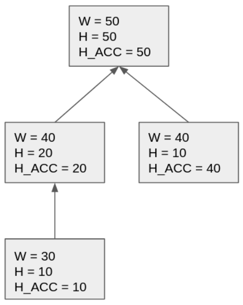

Fig. 2. The layout algorithm running on a layout tree of size 4. All nodes have an height attribute of 10.

shape largely matches the DOM tree (though with some deviations for "generated content" like bullets for list items). The layout phase reads layout node properties (which reflect HTML attributes, CSS properties, and other data) and computes layout fields like width and height for those nodes. Later phases like painting then read those layout fields and use them to draw the page to the screen. The DOM and layout trees are typically poorly-balanced, with both lots of "wrapper" elements (with a single child) and also many "list" elements with many children; an example DOM tree for a small web page is shown in Figure 6. In memory, the layout tree is stored as a pointer tree, with the children stored as a doubly linked list, to allow for fast insertions and deletions. This pointer-heavy structure means that layout nodes are spread throughout memory, with every access typically generating a cache miss.

2.1 The Layout Phase

The layout phase computes a number of layout fields for each node in the layout tree. Computing layout fields is a recursive process because each node's layout depends on the layout of its neighbors. For example, the height of a node is (typically) the sum of the heights of all its children, while a node's x position depends on the x position of its previous sibling, plus that previous sibling's width.[1] Moreover, visible properties like width and height in turn depend on intermediate properties such as intrinsic size, current line ascent/descent, and even more obscure properties like the sum of its siblings' `flex-grow` values. A complete layout phase must visit each node multiple times, in a well-defined order, in order to compute each field on each node before any others that depend on it.

We remark on a couple of key properties of layout that influence our approach:

(1) Bounded work per node. There are no data-dependent loops, recursions, or data structures except the layout tree itself.
(2) Immutable shape. The layout tree's structure is not modified during layout; only the layout fields on each node are. Moreover, these computed fields are only written once (per frame) and then become read-only (for that frame).
(3) Static control flow. The fields are computed in a fixed order dependent only on the layout tree shape, not the values of other computed fields.

[1]In reality, these rules are quite a bit more complex, with various exceptions to the simplified sketch given here.

(4) Local data flow. To compute a particular node's fields, only that node's neighbors in the layout tree and their fields are accessed.

These properties are discussed in the wealth of prior work on formalizing layout [? ? ? ? ? ? ? ?].

In other words, layout can be expressed in the DSL shown in Figure 3. In this DSL, a layout is defined by a set of passes Pass_n performed in a certain order (the schedule). Each pass performs a recursive, in-order traversal of the tree, computing some fields pre-order and some fields post-order. Each field computation is a simple assignment $\text{self}.V \leftarrow T$, where V is a field of the current node self and T is an expression that can refer to fields of self or its neighbors parent, prev, next, first child, and last child. Computations can also refer to HTML attributes or CSS properties of the current node using $\text{attribute}[x]$ or $\text{property}[x]$.[2] Expressions can also use conditionals to test whether a given neighbor exists (N?). Notably, our DSL enforces the key properties of layout described above: there is no functionality to mutate the tree shape or to reorder control flow according to field values. There are also no loops or data structures, and the only field access allowed is to a node's neighbors.

Many compound operations can be compiled into this DSL; for example, to access parent.parent.x, one can instead define $\text{self.parent\_x} \leftarrow \text{parent}.x$ to store each node's parent's x field. Then $\text{parent.parent}.x$ is simply parent.parent_x. In any case, prior work in similar DSLs [? ? ? ? ? ? ?] has already shown that layout features like the CSS box model, intrinsic widths, absolute positioning, flexible box layout, and line breaking are expressible in this DSL.

A key property of a layout L is the order in which it computes different fields on different nodes in the tree. Specifically, let the *trace* of a layout on a tree T be sequence of (n, v) pairs where n is a node in T and v is a field name in L. Because our DSL only performs in-order traversals, the trace is structurally related to the tree shape: local changes to the tree (node insertions or deletions) imply a local change to the trace (subtrace insertion or deletion). This structural relationship means that the ordering of any two (n, v) pairs is fixed: if (n, v) appears before (n', v'), this appear-before relationship will be maintained even as new nodes are added or removed.[3]

In the remainder of this paper, we will assume a correct implementation of web page layout exists in this DSL, with correctness meaning both the correct set of rules that compute the fields required in the CSS standard [?] and also a correct schedule for computing those fields while preserving dependencies. Our own implementation (Section 5.1) implements a subset of CSS covering a variety of complex and widely-used web layout features; however, spineless traversal is applicable to any layout expressed in this DSL, and we do not focus on the details of the rules or schedule further. Moreover, while this paper focuses on web layout, we expect spineless traversal to be applicable to a number of other incremental computations, differential dataflow, or semi-naive evaluation problems.

2.2 Incremental layout

Layout needs to be performed any time the DOM tree changes, typically in response to a user interaction like clicks, hovers, drags, animations, or typing, possibly via the execution of page-specific JavaScript code. The change might be to an HTML attribute or CSS property (as happens when the user selects a drop-down item or types into a text box) or might be the insertion or deletion nodes (as might happen when a page is loaded or new content is inserted from the network), but most of these modifications, and especially the most latency-critical interactions like hovers, drags, animations, and text editing, modify only a small portion of the DOM tree at a time. (By contrast,

[2]We use two different namespaces for HTML attributes and CSS properties because some names, like `height`, appear in both sets. There is no other semantic difference between them, and other accessible properties, such as the tag name or image width and height do not affect invalidation traversal (but are modeled in our system as special properties).

[3]We assume here that only brand-new nodes, not previously-removed ones, are inserted into the tree. To our knowledge, this is true of all major web rendering engines.

$$
\begin{aligned}
\text{Layout} &:= \text{Rule}^+; \textbf{schedule}\ \text{Pass}_n^+ \\
\text{Rule} &:= \textbf{def}\ \text{Pass}_n()\ \{\ A^+;\ \text{children.forEach}(\text{Pass}_n);\ A^+;\ \} \\
A \in \text{Assignment} &:= \text{self}.V \leftarrow T \\
T \in \text{Term} &:= \text{if } T \text{ then } T \text{ else } T \mid F(T^+) \mid N? \mid N.V \mid \text{attribute}[V] \mid \text{property}[V] \\
N \in \text{Neighbor} &:= \text{self} \mid \text{prev} \mid \text{next} \mid \text{parent} \mid \text{first} \mid \text{last} \\
V \in \text{Variable} &:= \text{unique symbols} \qquad F \in \text{Function} := \text{primitive functions}
\end{aligned}
$$

Fig. 3. A minimal DSL for defining web layout as a set (rules) of passes performed in a specific order (schedule). The syntax P^+ represents a sequence of non-terminal P. Passes are in-order traversals of the layout tree performing a sequence of assignments to local fields while accessing fields of the current node or its neighbors.

navigating to a new page might change a large portion of the tree but likely isn't latency-critical since it follows a high-latency network request.) In any case, once the DOM tree changes, the layout tree must change as well, and then the computed value of various layout fields typically need to be recomputed.

Each layout node thus maintains a *dirty* bit for each field, which defines whether that field value needs to be recomputed.[4] The dirty bit is set when a layout node is added to the tree and is cleared when its associated field is computed. A field is also dirtied when a value it depends on changes. For example, in the simple layout of Figure 1, when a node's `height` attribute is changed, its `height` field is marked dirty. Then, when that `height` field is recomputed, its `height_acc` field is in turn marked dirty. When the `height_acc` field is recomputed, its next sibling's `height_acc` field is then marked dirty in turn. Alternatively, if a node is deleted, the `height` of its parent (if it was the last child) or the `height_acc` of its next sibling must be marked dirty. In other words, fields are marked dirty when HTML attributes or CSS properties change, or when nodes are added and removed from the DOM; then these dirty bits propagate through the layout tree until all affected nodes are marked and recomputed.[5]

One critical optimization is that recomputing a field will not mark any dependent fields if the field's recomputed value matches the its previous value. For example, imagine typing into a multi-line text box; this moves around the text within the text box but, if the text box has a fixed height, doesn't affect the size or position of anything outside it. More broadly, the web layout rules have many conditionals, `min`, and `max` operations; as a result, recomputing a field may not change its value even if the fields it depends on have changed. This same-value optimization is critical: it means that many changes affect only a small fraction of the layout tree, which is what makes incremental layout so efficient.

Fields are marked dirty both *before* layout starts, when an attribute changes or a node is added or removed, and also *during* layout, when fields are recomputed. As a result, the set of nodes *currently marked dirty* is distinct from the set of *transitively dirty nodes*, and this second set must be *discovered* in the process of performing layout. Whether a field will be marked dirty during layout can be hard to predict: it depends on which fields are dirty before layout starts, which other fields on which nodes depend on those fields, and also whether recomputing those fields changes their value or leaves them unchanged. In this paper, when we talk about dirty nodes, we typically mean

[4]In a real browser, often one boolean bit summarizes whether any in a set of fields need to be recomputed, but we describe a bit per field here for simplicity.
[5]As described this is a flow-insensitive dependency analysis; it is possible to do a finer-grained flow-sensitive analysis instead. This design decision is orthogonal to the algorithms presented in this paper.

the logical set of transitively dirty nodes, but note that this set is conceptual: it cannot be easily determined before layout is done.

2.3 Dirty bit propagation

Dirty bit propagation code can be synthesized from the layout algorithm. To do so, one analyzes every assignment $\mathsf{self}.V \leftarrow T$ in the layout program. For each field access $N.U$ in the expression T, we know that $\mathsf{self}.V$ depends on $N.U$, meaning that any changes to $\mathsf{self}.U$ must mark $N^{-1}.V$ as dirty, where N^{-1} is the inverse of the relation N; flipping next and previous, mapping first and last to parent, and mapping parent to all children. Note that, if the original algorithm respects the dependency order, then a field is only computed after all fields that it depends on. This in turn means that, after a field is computed, it can no longer be marked dirty. This guarantees termination: computing a field clears its dirty bit, so by the end of an incremental layout every field is clean.

Insertion and deletion require particular care in propagating dirty bits. An inserted node has all of its new fields marked dirty; likewise, deleting a node marks all dependents of its fields. Inserting or deleting nodes can also change N? expressions; any fields that use such expressions must then also be marked dirty.[6] Importantly, an entire subtree can be inserted or deleted at once; this is quite common in real-world web pages where pop-ups or menus can appear as a result of a hover or click. Since all data accesses in our model are local, only the root of the subtree actually needs to be considered, meaning that deleting or inserting a subtree takes constant time, regardless of the size of the subtree.

With dirty bit propagation implemented, an incremental layout phase must simply find dirty fields (including transitively dirty fields), and recomputing them in the scheduled order. We call a method for doing this an incremental traversal algorithm. A naive incremental traversal algorithm could simply traverse the entire layout tree, as if rerunning from scratch, only re-computed dirtied fields. While this trivial invalidation algorithm respects the dependency order, clears all dirtied fields, and performs the minimal number of field computations, it also accesses every node in the tree. As a result, it incurs just as many cache misses as a non-incremental layout and therefore does not lead to a significant speed-up.

Driven by this observation, we classify the nodes accessed by an incremental traversal algorithm into dirty nodes, which have dirtied fields and must be accessed to recompute those fields, and auxiliary nodes, which are accessed only to find dirtied nodes. The number of dirtied nodes is set by the dependency structure of web layout and the changes performed to the DOM tree. However, the number of auxiliary nodes accessed is dependent on the invalidation traversal algorithm and can be reduced through better algorithms for finding dirtied nodes. While the problem of auxiliary nodes in incremental traversals may seem quite specific to readers, we emphasize that it is a major source of latency in existing web browsers, which in turn are both critical application platforms and also already highly optimized.

2.4 Double Dirty Bit

The SOTA algorithm, which we dub the Double Dirty Bit algorithm and which is used (with variations) in all major rendering engines, reduces auxiliary nodes by adding a summary dirty bit per pass to every node. This summary bit indicates whether any field assigned during the pass is dirty in the *subtree* rooted at a node. The summary bit is set every time a field set in that pass is marked, and since the summary bit is recursive over the subtree, setting a summary bit must also set the parent's summary bit, and its parent, and so on, stopping at the root node or at a node with

[6]Once again, this could use a flow-sensitive analysis or a simpler flow-insensitive one. The invalidation challenges are similar in either case.

```
def mark_dirty(self):
    self.dirty = True
    self.set_summary_bit()

def set_summary_bit(self):
    if self.summary_bit: return
    self.summary_bit = True
    if self.parent:
        self.parent.set_summary_bit()
```

Fig. 4. Setting the summary bit for a node.

```
def find_dirty_nodes(self):
    if self.dirty:
        yield self
    if self.summary_bit:
        # Access auxiliary nodes
        for child in self.children:
            find_dirty_nodes(child)
        self.summary_bit = False
```

Fig. 5. Finding the dirty nodes in a tree.

the summary bit already on. The summary bit then allows an incremental layout to skip recursing into subtrees whose summary bit is off. Figure 5 shows an example implementation of Double Dirty Bit as a Python iterator yielding dirty nodes.

The Double Dirty Bit algorithm reduces the number of auxiliary nodes, greatly improving performance, and is considered the state of the art for layout invalidation traversal [? ?]. The number of auxiliary nodes is still large. To reach any dirty node, the Double Dirty Bit traversal must must traverse the path from the root of the tree to the dirtied node—the "spine" of that node. Moreover, each node along that spine will have its summary bit set, meaning that the Double Dirty Bit algorithm must recurse into that node's children. In other words, Double Dirty Bit's auxiliary nodes are the spine of each dirtied node plus also all children of nodes in that spine. Since web layout trees have both very wide and very deep nodes, this set of auxiliary nodes can be very large. For example,the `lobste.rs` page of Figure 6 has an `<input>` element representing a dropdown that a user might interact with. Opening and closing the dropdown transitively dirties seven elements (the `<input>`, its sibling `<label>` and all its descendants, and its parent `<span>`), but its spine contains eight nodes and adding the children of the spine nodes brings the total number of auxiliary nodes to 25, three times the number of dirty bits.

More generally, the number of auxiliary nodes can be vastly larger than the number of dirty nodes, especially for latency-sensitive interactions where web developers take pains to dirty as few nodes as possible. Unsurprisingly, these auxiliary nodes are a significant source of latency, affecting both browser and web developers. Google Chrome's Lighthouse performance monitoring tool [?], for example, warns if tree depth or fanout is high. This implicitly means that achieving the expected performance, especially on a highly interactive web application such as Figma, VS Code, Office 365, or Photoshop, may require changing the global shape of the layout tree, which (in modern frameworks like React) requires refactoring the application as a whole. Naturally, an invalidation algorithm that simply did not require so many auxiliary nodes would be a superior solution.

3 SPINELESS TRAVERSAL

Spineless Traversal is such an algorithm. Unlike Double Dirty Bit, Spineless Traversal can jump directly to the next dirty node without accessing any auxiliary nodes; as a result, it suffers dramatically fewer cache misses than Double Dirty Bit. Achieving this requires a more computationally-heavy approach, storing all dirty nodes in a priority queue and maintaining the correct traversal order using an order maintenance data structure. Spineless Traversal's savings in cache misses outweigh the greater computational requirements of these data structures.

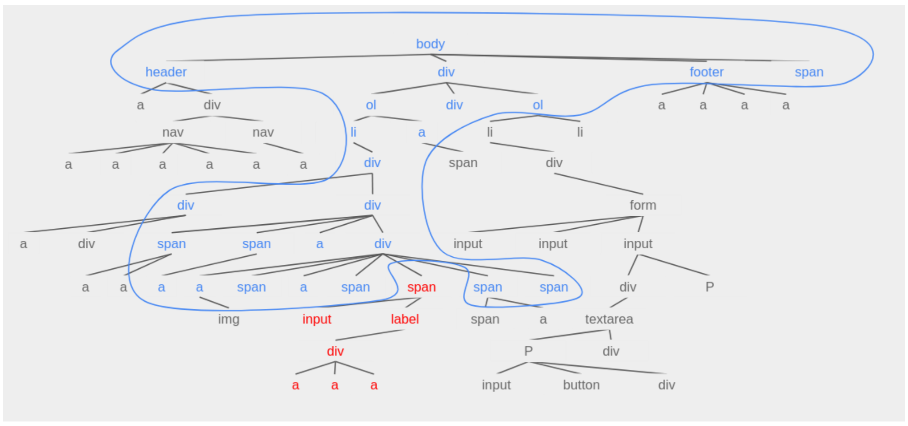

Fig. 6. The Double Dirty Bit algorithm in action on a `lobste.rs` page. The red `input` node is being dirtied. However, to reach the node, Double Dirty Bit have to traverse the 25 blue nodes. This is typical, especially for the most latency-critical user interactions.

3.1 The Priority Queue

Spineless traversal is conceptually simple. Each node field in the layout tree is assigned a label, indicating its position in a layout trace. Node fields are placed in a priority queue when dirtied, with the label used as their priority. To perform an incremental layout, node fields are popped from the priority queue and recomputed, with dependent fields added back to the priority queue as they get dirtied, until the priority queue is empty. Because the priority queue pops node fields with smaller labels first, Spineless Traversal respects the dependency order of layout. Since only dirty nodes are ever pushed or popped from the priority queue, no auxiliary nodes are accessed.

We use a min-heap for our priority queue, which is cache-friendly and requires relatively few operations for each push and pop. Moreover, the queue is typically small: while there are typically thousands of nodes, with each node having approximately 50 fields, the priority queue typically contains less than 1000 fields, and for the most latency-critical interactions, like hovers or drags, it can contain 100 or fewer. With such a small size, a priority queue push/pop requires 5–10 label comparisons, which can be performed in roughly the time for one or two L2 cache misses in our optimized implementation. Since there are typically *far* more auxiliary than dirty nodes, this means Spineless Traversal performs much faster than Double Dirty Bit.

3.2 Order Maintenance

The key to Spineless Traversal is maintaining the labels. The issue is that layout nodes are added and removed over time; labels need to be comparable, but it also needs to be possible to add new labels between existing ones, arbitrarily. Following SAC [**?**], spineless traversal achieves this using an *order maintenance* data structure. First introduced by **?**], order maintenance is a data structure that maintains a totally ordered set of objects while allowing objects to be added and removed from the order arbitrarily. Crucially, both adding/removing and comparing nodes takes $O(1)$ time. Abstractly, order maintenance provides the following API:

Compare(p, q)	Decides whether p or q comes first in the order (or are equal).
Head$()$	Returns the first object of the order.
Create(p)	Creates and returns a new object right after p.

Deleting OM objects is also possible; however, our implementation does not.

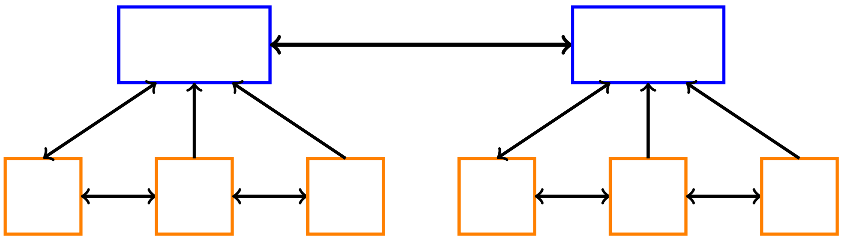

Fig. 7. An Order Maintenance data structure. The blue node represent the higher level doubly linked list, and each node store a lower level doubly linked list, denoted by the red node. Lower level node also store a pointer to the higher level node. Each node additionally hold an unsigned integer, label, such that inside a single list the node earlier have a strictly smaller label then the node later.

Our implementation is based on that by **?**], which uses a two-level structure with a double-linked list of double-linked lists. Objects are represented by nodes in the lower-level lists. Both levels are ordered; the total order traverses lower-level lists in order, in the order dictated by the higher-level list. Each object (node in the lower-level list) maintains a pointer to its higher-level list cell; two objects are in the same low-level list if they have the same higher-level pointer. To allow fast comparisons between nodes, both low-level and high-level list cells store an unsigned integer of fixed size (in our implementation, 32 bits) called labels. Within both lists, node labels are strictly increasing; this makes comparisons fast. Specifically, comparison has two cases: the two objects can be in the same low-level list, in which case their labels can be compared, or in different ones, in which case their parents' labels can be compared. This comparison operation is the bulk of the Spineless Traversal time so its simplicity is essential; Section 4 discusses how we micro-optimize it in our implementation.

To create an object inside an order maintenance structure, a new lower-level list cell is created whose label is the average of the two neighboring labels.[7] If the two labels differ by exactly 1, however, this becomes impossible (a label would be repeated). In this case, the data structure re-balances itself, evenly reassigning labels to existing objects. This process might create a new higher-level list cell to split a lower-level list in two, ensuring a sufficiently large gap between its cells. Rebalancing is algorithmically tricky but is not a significant time sink in our use case, so we do not detail re-balancing here; details can be found in **?**].

3.3 Bulk insertion and deletion

Bulk insertions into the layout tree are common, often due to a "lazy loading" pattern: a "shell" web page loads first and shows a loading indicator; then the "content" loads and is inserted into the page as a single large subtree. This pattern is encouraged by frameworks like React, and can occur in several stages, with a "shell" first inserting "subshells" which themselves load subcomponents in turn. Efficiently inserting the large subtree requires special care in Spineless Traversal.

The basic issue is that all fields of all nodes in the subtree need to have them and their OM objects initialized, yet we also want subtree insertion to be fast. Our solution adds these "initialization passes" as special elements in the priority queue.[8] In other words, the priority queue can contain not only (n, v) pairs for a dirtied field but also (r, p) fields pointing to the root r of an entire subtree

[7]When creating a node after the last node, the maximum representable number is used as the larger number.

[8]Just enqueueing all fields of all nodes in the subtree is not a good idea—it will create a large queue with slow queue operations. In our experiments, this often took the queue from tens to thousands of members, leading to a small integer slowdown.

that needs to be initialized for pass p. When a subtree T is inserted, each pass for it is enqueued; the Order Maintenance object for that is created after the OM of the last field initialized by P in T's previous sibling or parent. In this way, we maintain the consistency of incremental layout with from-scratch layout. [9]

When one of these (r, p) elements is popped from the queue, the pass p is performed on the whole subtree under r, creating all necessary order maintenance nodes. Since all fields in newly-created nodes are initially dirty, performing this pass will not enqueue any fields in the priority queue, meaning that no priority queue operations need to be performed. Likewise, since all data accesses are local, no existing nodes will refer to any newly-inserted node except the root node, so no nodes need to be dirtied when running p on the subtree. This means that, when initializing a subtree, no priority queue operations or dirty bit propagations need to be performed, which makes subtree insertion faster. (However, order maintenance objects still need to be created, meaning that, despite these optimizations, Spineless Traversal is typically slower than Double Dirty Bit for subtree insertion.)

When deleting a subtree, some nodes in the subtree may already be in the priority queue; for example, this can happen if a subtree is inserted and then later deleted. To avoid unnecessary recomputation, we add a "deleted" bit to each node and defer actual deallocation until after incremental layout is performed and the priority queue is empty.[10]

4 OPTIMIZATION

By avoiding auxiliary node accesses, Spineless Traversal can be dramatically faster than Double Dirty Bit in some cases. However, to make Spineless Traversal competitive on arbitrary real-world web pages, the computationally expensive steps in spineless traversal have to be meticulously optimized to avoid allocation, reduce memory traffic, and optimize branch mispredictions.

4.1 Pointer Compression and Custom Allocator

Order maintenance objects have to be allocated every time new layout nodes are inserted; optimizing that allocation is essential. We use a hand-written pool allocator for order maintenance objects; in fact, we use separate pools for high- and low-level list cells to enhance locality. Since allocations are always for the same size, our custom allocator is significantly simpler than the system allocator.

Our pooled allocator, `OMPool`, is shown in Figure 8. It is paramaterized by *two* types: the type of allocated object `T` and the index type `P` for pointers to allocated objects. Crucially, `malloc` and `free` return and consume the pointer type `P` instead of raw pointers. (And `addressof` function converts the pointer type `P` to a raw pointer.) By making `P` a smaller integer type, like `uint32` or `uint16`, order maintenance objects become smaller and more of them fit per cache line, which in turn improves throughput. In our implementation, we use 32-bit pointers; this conveniently makes the total size of an order maintenance object 128 bits, meaning that it exactly fits in each cache line, with no order maintenance objects split across two. By contrast, an experiment with 16-bit pointers actually increased runtime—because, we suspect, some order maintenance objects split across two cache lines, which introduced additional stalls.

The actual implementation of `OMPool` is standard; it stores a `pool` of memory as a `std::vector`, in which we ensure sufficient capacity at startup. Freed elements are placed in a separate `freed` vector, which is preferentially drawn from by `malloc`. Because the objects are all the same size, there is no fragmentation and `malloc`/`free` are nearly instantaneous. Moreover, since spineless

[9]An edge case is inserting a subtree into a subtree that itself has not yet been laid out. In this case no further actions need to be taken, since both subtrees will be visited together.

[10]Some existing browsers choose to retain layout nodes even longer, or even garbage-collect them.

```
template<
  typename T,            // Type of allocated object
  typename P=uint32_t    // Integer type for "pointers"
> struct OMPool {
  std::vector<T> pool;   // Fast allocation, maximize cache usage

  std::vector<P> freed;  // Rapid reuse minimizes cache churn

  // Implementation is straightforward
  T* addressof(P p) { return &(pool[p]); }
  P malloc();
  void free(P p) { freed.push_back(p); }
}
```

Fig. 8. A pooling allocator that focuses on reducing cache misses.

traversal creates order maintenance objects in order, allocation patterns are extremely favorable, with temporally-close nodes placed nearby in memory.

4.2 Branchless Order Maintenance Comparison

Priority queue pushes and pops spend basically all of their time comparing order maintenance objects. Order maintenance objects are small and, thanks to our allocator, typically stay in cache. However, order maintenance object comparison has two cases (same or different second-level lists) and the bottleneck ends up being the pipeline stall induced by the conditional. The branch predictor does not help much, because (thanks to the heap) the comparison is unpredictable. We therefore implemented a branchless comparison function, relying on conditional move instructions instead; in a microbenchmark, this reduces comparison time to 5 cycles; Figure 9 shows the assembly implementation.

4.3 Dependency Deduplication

When synthesizing the dirty bit propagation code, we make sure to only dirty any given field once per field computation. For example, if a field $N.V$ is used twice in an expression, or if two different paths first.V and last.V reverse to the same field, the field is only dirtied once. This deduplication is especially challenging in the case of N? expressions, since whether or not a field is dirtied can depend on whether an element is the first/last/other child of its parent. Moreover, the dirty bit is checked before pushing to the priority queue; this means any node field appears in the priority queue at most once, which keeps the queue small.

4.4 Field packing

To further shrink the size of the priority queue, we use a single dirty bit to cover multiple co-computed fields. Instead of a dirty bit per field, we use two dirty bits per pass—one for the pre-order-computed fields and one for post-order-computed fields—and place these pass references, not individual fields, in the priority queue. This further reduces the number of unique dirty bits set during dirty bit propagation and reduces the size of the priority queue and the number of order maintenance objects needed. Field packing in this way is a common optimization in real browsers; the trade-off is that it reduces the complexity (and thus cache misses in) invalidation traversal at the cost of possibly more field recomputations (since the browser must now recompute

```
bool operator<(const _l2_node &l, const _l2_node &r) {
  Label lpl = l.parent->label, rpl = r.parent->label;
  Label ll = l.label, rl = r.label;
  uint64_t result;
  asm volatile(

    "xor   %%rbx,  %%rbx    \n"
    "cmp   %1,     %2       \n"
    "seta  %%bl             \n"
    "xor   %%rax,  %%rax    \n"
    "cmp   %3,     %4       \n"
    "seta  %%al             \n"
    "cmove %%rbx,  %%rax    \n"

    : "=&a"(result) : "r"(ll), "r"(rl), "r"(lpl), "r"(rpl));
  return result;
}
```

Fig. 9. The branchless comparison assembly code, with the conditional move instruction highlighted. Note that, while the assembly snippet is seven instructions long, the two `xor` instructions clear a register and are thus handled by the register renamer, and also the `cmp`/`seta` combination is fused on recent Intel/AMD CPUs. This means the assembly snippet itself typically takes three cycles to execute; even adding the label loads, comparison typically takes only five cycles.

all covered fields when a dirty bit is set); the trade-off appears to be worth it. Spineless Traversal can be applied to any field packing, whether finer- or coarser-grained, though coarser-grained packing makes the priority queue smaller, meaning that, with Spineless Traversal, browsers may opt for coarser-grained field packing then they do today.

4.5 Field representation

Recall that priority queue entries refer to either a subtree or a node, and also name a phase plus pre/post location. We combine the subtree-or-node bit with the phase and pre/post location into a simple enum, which we store in a byte. We use the enum as an index into a jump table, which allows us to efficiently execute the field recomputation that corresponds to a single priority queue entry.

4.6 Attempted, Failed Optimizations

A number of attempted optimizations failed to improve performance:

- A hybrid of Double Dirty Bit and Spineless Traversal, using a summary bit for subtree dirty bit propagation but the priority queue for more distant jumps. We were unable to make switching between the two modes efficient enough to be competitive.
- A splay tree instead of a min-heap to improve priority queue locality. Resulted in a slowdown.
- A red-black tree instead of a min-heap for the priority queue. Resulted in a slowdown.
- 1-based array indices in the min-heap, which need fewer addressing instruction. Resulted in a slowdown.
- 16-bit `OMPool` pointers, and 16-bit OM labels. No faster than 32 bits, and more rebalancings needed.

- Pointer tagging—reduce priority queue node size by moving boolean bits into pointers. No faster.
- Splitting OM `left`/`right` pointers from the `label` and `parent`/`children` pointers, to improve cache locality. Did not see improvement.
- Deallocating OM objects when the corresponding tree node is deleted, to increase cache locality. Did not see improvement.

5 MEGATRON

To evaluate Spineless Traversal, we implement it for a simplified web layout algorithm implemented in the DSL of Figure 3 and compiled by a compiler we call Megatron. In this section, we first describe the layout modes implemented, then how we compile the DSL to a layout engine, and finally compare Spinless Traversal to Double Dirty Bit on a collection of real-world web page interactions.

5.1 Web Layout Algorithm

To compare Spineless Traversal and Double Dirty Bit, we needed a web layout algorithm that was decoupled from its invalidation strategy. Unfortunately, existing browser rendering engines are too complex and too tightly coupled to their current invalidation strategy, making it impossible to reuse them. We instead implemented our own web layout algorithm based on the Cassius and MEDEA formalizations [**? ? ?**]. Naturally, this implementation is simplified compared to a real-world web browser and only implements a subset of features. However, we took care to implement several features with complex invalidation behavior, including intrinsic width, flex-box layout, and positioning. In total, our implementation is 700 lines of code in the custom DSL, which computes approximately 50 fields for each layout node. The remainder of this section describes several key layout features implemented in our layout algorithm.

Box model. Each layout node has an x, y, `width` and `height` field. Typically the rectangle this represents contains all of the node's children, while sibling elements don't overlap. In general, width is computed top-down (children's width depends on parent's width), height is computed bottom-up (parent's height is the sum of children's heights) and x and y are computed in-order. This forms long dependency chains between elements, where updating one element can cause many others to be transitively dirtied. However, the `width`, `min-width`, and `max-width` CSS properties, and similar for `height`, can directly set width and height, breaking these dependency chains. Our layout algorithm's support for these properties means this early-stopping behavior is tested. That said, even the explicit `width` and `height` properties allow "percentage values" like `50%`, which are resolved relative to the parent and thus still create inter-node dependencies.

Line Breaking. Line breaking lays out inline layout nodes (text) horizontally from left to right, until the edge of the parent box is reached, and which point the next layout node is placed at the left edge of its parent, one line below its previous sibling. Line breaking requires handling control dependencies, where one layout nodes' width may move layout nodes from one line to another by changing line breaking. Additionally, our layout algorithm allows different lines to have different height (based on the height of the largest layout node in the line), which introduces a field (line height) that is dependent on many different nodes (each word in the line). This has a number of interesting effects for invalidation. For example, adding a node to a line may cause no change if it is not the tallest node in the line. However, if the new node is now the tallest node, that may cause the line to become taller, which in turn can cause all other text on the page to move down, causing a lot of invalidation.

Display. Nodes have a `display` property that changes whether they act line words (`inline`) or paragraphs (`block`). The `display` property can also be set to `none`, in which case the layout nodes are not shown on the page and have almost no effect on layout. Changing a layout node's `display` property between `block` and `none` is a common way to implement drop-down menus, pop-ups, and other elements that appear and disappear, such as the "preview" hover effects on Wikipedia. Importantly, inserting a `<script>` or `<style>`, which are `display: none`, should not invalidate any other nodes and should be fast.

Position. An element with `position: absolute` is manually assigned a position on the page by the web developer. This is commonly used for popups, tooltips, and other effects that appear over other page content. It is also common to change the manually-assigned x and y positions from JavaScript, such as to move a tooltip away from the cursor. Layout nodes with absolute positioning do not affect the position of other layout nodes outside their subtree, and handling changes to x and y positions quickly is essential.

Intrinsic sizes. In a number of cases, a layout node's width or height is computed from its "intrinsic" size, which effectively measures how large it would need to be to contain all its text without line breaks. For example, absolutely-positioned elements compute their width and height this way by default. Importantly, intrinsic widths are computed bottom-up, but then used in the top-down width computation and then the bottom-up height computation. This means intrinsic sizes require the use of multiple layout phases and require Spineless Traversal to support such.

Flex-box. Flex-box layout is the most complex feature our layout algorithm supports. In flex-box layout there is flex container element whose children are flex items. The width/height of flex items depends on the intrinsic sizes of the other flex items and the actual size of the flex container. Properties like `flex-grow` and `flex-shrink` determine how the intrinsic sizes of the flex items are adjusted to compensate for either extra or not enough available space in the flex container. The `max-` and `min-width/height` properties can also cap the growth/shrinkage of individual flex items. In all, our implementation of flex-box layout uses 9 of intermediate fields and requires 2 passes to compute all of them.

Miscellaneous. We also implemented a variety of miscellaneous features, including automatic sizing of images and video, manual line breaks with the `<br>` element, and conditional elements like `<noscript>` (which are only rendered if JavaScript is disabled, not the case in our tests). We also had to add a special case for `<svg>` elements, whose children describe drawing commands that do not participate in layout. Finally, a variety of minor tweaks like the `width` and `height` HTML attributes (which behave slightly differently from the CSS properties) were also implemented.

5.2 Compiler

This layout algorithm is implemented in the DSL and then compiled to C++ to maximize performance.[11] Passes are compiled to three C++ functions: the pass itself, a function that performs just the pre-order assignments and a function that performs just the post-order assignments. The compiler type-checks all fields, attributes, and properties using Hindley-Milner type inference [?] and uses appropriate unboxed C++ member variables to store the relevant values. Importantly, this means a single node and all its fields are contiguous in memory (as it would be a real web browser) with a minimum of pointers (beyond the standard parent/first/last/next/previous pointers to other layout nodes) and that field access is compiled to a memory offset. All string values (like the keyword values for `display`) are interned and represented in C++ as a single `enum` type, meaning

[11] It was initially an interpreter. Thanks to Final Tagless[?], the transition from interpreter to compiler was smooth.

that no string allocation or comparison is performed at runtime. Discriminated unions are used for fields with units. The bottom line is that the compiled code is long but readable, idiomatic, and performant C++ code with no allocation, hash, or string operations. This is critical because it means that, like in a real web browser, computing fields is very fast and the cache misses from finding dirty fields are a measurable fraction of the runtime.

The compiler also synthesizes all dirty bit propagation code. Specifically, the compiler examines each assignment in each pass, determines which fields on which neighbor nodes are read to compute each field, and computes the inverse neighbor relation. Every field assignment compares the new and old values in the field and sets dirty bits only if the new and old values differ. The actual invalidation traversal algorithm (Double Dirty Bit or Spineless) is encapsulated in a global value that provides a method to dirty a node/field and an iterator over all dirty nodes. The incremental layout entry point function iterates over all dirty nodes and calls the relevant pass functions. The compiler also generates functions to modify the tree, either by inserting and deleting tree nodes or by changing an attribute or property. These generated functions dirty the appropriate nodes and fields, including using the special case for subtree insertion for Spineless Traversal.

To ensure our implementation is correct, the compiler can also generate a from-scratch layout function which does not use dirty bits at all and instead recomputes the entire layout from scratch. This was extremely valuable during development and gives us confidence that our invalidation algorithm is correct.

6 EVALUATION

We use this web layout algorithm to compare Spineless Traversal against the Double Dirty Bit algorithm on 50 real-world websites.

6.1 Benchmarks

To evaluate web layout for these websites, we modified the Ladybird web browser to dump the layout tree at every rendered frame; we then use a separate program to “diff” successive frames, outputting a list of insertions, deletions, and attribute/property changes. A benchmark program then reads the diff and performs each modification in the list, then invokes the incremental layout entry point. Both invalidation and recomputation time are measured using the `rdtsc` instruction, allowing a granular comparison of both Spineless Traversal and Double Dirty Bit. In total, the 50 websites generate traces with 2216 frames in total. Each frame leads to a single incremental layout call, so our experiment has 2216 individual data points. Note that this large number of frames, covering gigabytes of layout tree data, nonetheless represents only a few minutes of web browsing activity. All experiments are run on a machine with an Intel i7-8700K CPU (8th generation) clocked at the standard 3.70 GHz with 64 KB L1 cache, 256 KB L2 cache (both per core), and 12 MB L3 cache, plus 32 GB of DDR4 memory across 4 DIMMs, each configured to 3000 MT/s.

The 50 real-world websites include Amazon, Wikipedia, Github, Google, as well as a number of other popular websites drawn from the Alexa ranking of top websites and focusing on large web pages and complex web applications. It also includes a number of personal favorites of the authors, including Github and Lichess. We focus on latency-sensitive interactions like hovering, typing, dragging, and animations. These interactions typically do not require loading data over the network and invalidation time is thus a big determinant of their latency. By contrast, interactions like loading a whole web page or moving between portions of a website typically have network latency which far overshadows any possible invalidation latency. Even though the interactions may seem minor, it is important to note that the browser is nonetheless performing a significant amount of work to render them. For example, on Wikipedia, hovering over a link shows a “preview” window, and Wikipedia code must track and respond to mouse movements to hide and show the

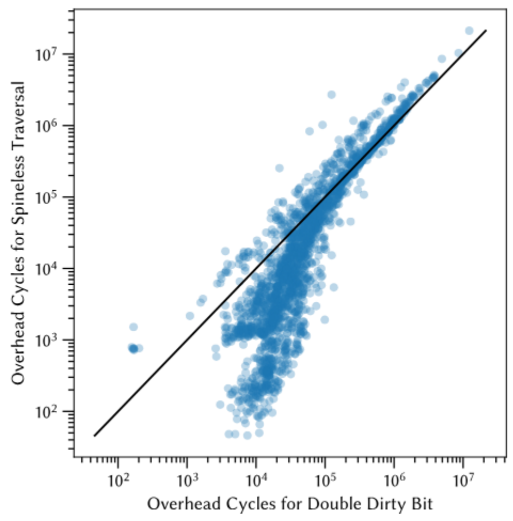

Fig. 10. The invalidation traversal time for all 2216 frames, with Double Dirty Bit time on the x axis and Spineless Traversal time on y axis. The diagonal line shows the $x = y$ equal time line; points below the line are faster with Spineless Traversal while points above the line are faster with Double Dirty Bit. Both axes are in log scale, meaning Spineless Traversal is often tens or hundreds of times faster than Double Dirty Bit.

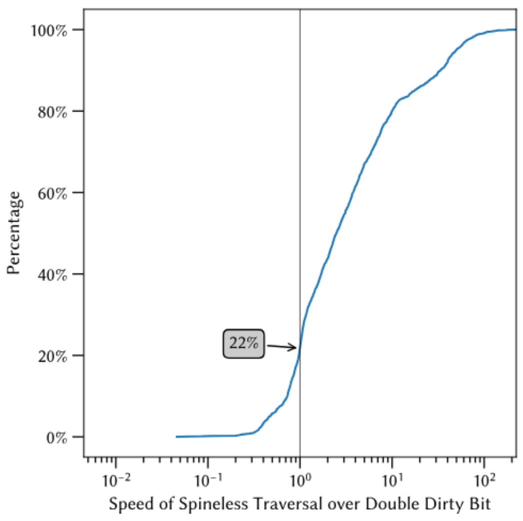

Fig. 11. A CDF of the ratio between Double Dirty Bit time and Spineless Traversal time for each frame. The vertical line, at 10^0, marks where both invalidation algorithms take equal time. To the left of the line, 21.8% of frames are slower with Spineless Traversal. To the right of the line, 78.2% of frames are faster with Spineless Traversal. The geometric mean speedup with Spineless Traversal is 3.23×.

preview window at the correct time. Moreover, there is a short, nearly-imperceptible animation by which the preview window slides and fades in and out of view. Similarly, on the Lichess web page, our trace captures one of the authors stepping through a chess opening using the website's chess commentary tools. The Lichess website renders the chess board using HTML elements and each move animates visual aids like arrows. Text editing, an especially latency-sensitive interaction, was also tested. For example, on the Google website we tested typing a search term letter by letter, with the Google website changing autocomplete suggestions as we typed.

6.2 Results

Figure 10 shows our results. In Figure 10, points below the diagonal line are faster with Spineless Traversal, while points above the diagonal line are faster with Double Dirty Bit. Most points are below the line, often with enormous speedups: 10–100×. These points are often interactions where only a few deeply-nested nodes are dirtied, where Double Dirty Bit accesses a huge number of auxiliary nodes. By contrast, while some points are above the line, meaning they are slower with Spineless Traversal, they are typically much closer to the line, indicating that slowdowns are not as severe as the speedups are beneficial. The geometric mean is a 3.23× speedup from Spineless Traversal, with only 21.8% of frames rendered slower. Figure 11 shows CDF of the speedup of Spineless Traversal over Double Dirty Bit.

Figures 10 and 11 show only invalidation time, including traversal, dirty bit propagation, enqueuing nodes in the priority queue (for Spineless Traversal) and setting summary bits (for Double Dirty

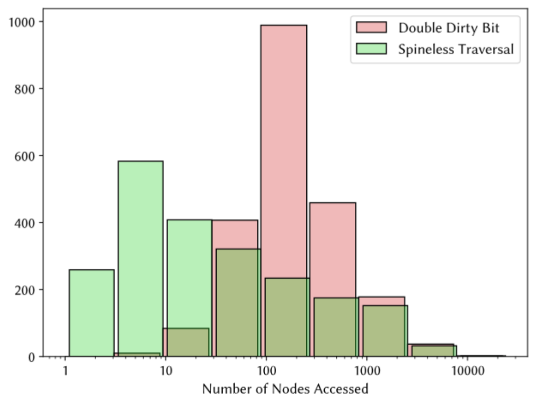

Fig. 12. Histograms of Number of Nodes Accessed by Double Dirty Bit and Spineless Traversal. Double Dirty Bit access much more nodes compare to Spineless Traversal, so the latter cause much fewer cache misses.

Bit). We measured field computation time separately, and confirmed that it was nearly identical between Spineless Traversal and Double Dirty Bit, which is as expected since both algorithms will recompute the exact same fields in the exact same order. In total, invalidation time was roughly one third of total runtime, while field computation time was roughly two thirds, showing that invalidation is a determinant of latency.

A more careful inspection of Figure 10 shows several additional features. The slowest frames all feature slowdowns. This is expected: the slowest frames likely represent the initial page load or other “loading” frames, which we necessarily capture in our traces. While speedups are always better than slow-downs, these loading frames likely follow network latency, so invalidation time for these frames is less important. Meanwhile, frames where fewer nodes are invalidated are typically those triggered in response to an animation or user interaction, where latency is most critical. Figure 13 shows an analog of Figures 10 and 11 but restricted to points where fewer than 1% of fields are recomputed; the intention is that a loading frame typically changes large portions of the web page while interactions typically change much less. On this latency-critical subset, which contains more than half of frames, the geometric mean speedup is larger, at 5.85×, and a smaller fraction of frames suffer slowdowns.

Spineless Traversal is also faster on average outside this subset, likely because the 1% threshold is an imprecise heuristic, but the dramatically larger speedup in this latency-critical subset supports the idea that Spineless Traversal helps most precisely on those points where latency is most important.

6.3 Case Study: Twitter

To gain a deeper understanding of Spineless Traversal, we focus on Twitter (now X), a social media platform. More specifically, we focus on the 125 incremental layouts performed to opened the Twitter news feed, load the default number of tweets, and scroll down repeatedly to load more tweets. Twitter is a large web page, with the tree growing to 3700 DOM nodes. This tree has a depth of 53, a maximum child count of 128, and some nodes having as many as 134 auxiliary nodes. Considering all the frames in aggregate, Twitter sees a mean speedup of 4.36× over the Double Dirty Bit algorithm.

Most of the 125 incremental layouts are small, dirtying no more then 20 nodes (Figure 14a). For these frames, incremental layout spends most of its time accessing auxiliary nodes. However, the

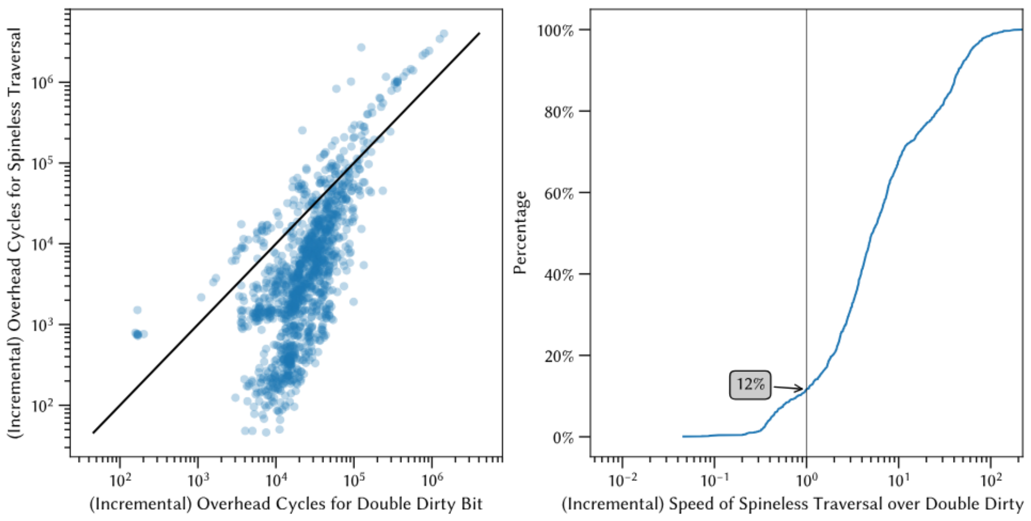

Fig. 13. The overhead scatter plot and the speedup CDF, restricted to frames where fewer than 1% of fields are recomputed. These frames typically represent the latency-sensitive interactions like hovering, animations or editing.

(a) The numbers of nodes changed externally for each frame, for Twitter. Most frames are very incremental, with 79 frames changing no more than 20 nodes in the tree. However there are some very large changes caused by content insertion/removal, with the largest change being 787 nodes in a single frame.

(b) The overhead scatter plot of Spineless Traversal compared to Double Dirty Bit for Twitter. Most points are below the $y = x$ line, indicating a speedup. Moreover, there are many points providing large speedup of over an order of magnitude.

largest incremental layout dirties several hundred nodes. The 125 incremental layouts are triggered by a variety of causes.

Linked Files. Many layouts are triggered when linked files—JavaScript, CSS, images, and videos—finish loading. Loading JavaScript might add new `<script>` and `<style>` elements to the page, while loading CSS can affect the CSS properties of existing elements. Loading images and videos, meanwhile, changes inherent width and height attributes (from 0 to the actual image width/height) used to compute a layout node's intrinsic size. Typically only one or a few nodes are dirtied, but these nodes have many auxiliary nodes. Most image and video nodes, for example are located deep in the layout tree, while `<script>` and `<style>` nodes are close to the root but have a huge number of siblings. Spineless Traversal thus dramatically speeds up these incremental layouts, by up to 100× in extreme cases.

Lazy Loading. Twitter uses a lazy-loading technique which first loads a "shell" page and then gradually adds more and more elements to the shell as more content is loaded over the network. For example, the header bar, side bar, ads, and tweets all load in separate frames and thus require separate incremental layouts. Scrolling causes yet more content (tweets and ads) to load. These frames, typically insert a single large subtree. Despite the bulk insertion optimizations described in Section 3.3, these are still difficult for Spineless Traversal because it must create a large number of new OM objects and then rebalance the OM data structure. Spineless Traversal is thus slower than Double Dirty Bit on these frames, typically by about 2×. However, note that the latency in this case is partially hidden by the network latency of loading the content in the first place, so the slowdown here is likely less critical than for other frames.

Removal. The Twitter application also occasionally removes remove parts of the page that are no longer visible to the user. For example, offscreen content is sometimes removed; Spineless Traversal handles these removals much faster than Double Dirty Bit, often by 10× or more, since Twitter is removing exactly those elements that do not affect what the user sees on the screen. Twitter also sometimes removes individual `<script>` and `<style>` elements that don't affect the page; here Spineless Traversal's speedup is smaller, approximately 2×, as multiple `<script>`/`<style>` are removed at once, amortizing cost of accessing auxiliary nodes in Double Dirty Bit.

Moreover, some frames mix file loading, lazy loading, and removals, simply due to whether these occur in the same or different frames; in this case the time taken for Spineless Traversal basically sums over the time for each individual change, while Double Dirty Bit can amortize the cost of traversing auxiliary nodes. As a result, these mixed frames typically see smaller speedups; however, these frames are still typically small (often many images loading at once) so Spineless Traversal still sees a significant speedup.

7 RELATED WORK

Incremental Computation. Incremental computation—speeding up computations by reusing previously-computed results—is a long-studied topic in computer science broadly [?] and programming language theory in particular; **?**] and **?**] give thorough surveys of the fielld. The recent Self-Adjusting Computation (SAC) [?] framework proposes incrementalizing arbitrary computations, including a cost semantics [?], optimizations for data structure operations [?], and opportunities for parallelization [?]. The Adapton framework [?] aims at *demand-driven* incremental computation, and allows manually-specified annotations **?**] for greater reuse. While this prior work focuses on general-purpose computations, Spineless Traversal is focused on a particularly-critical application of incremental computation: web browsers. This application-specific focus has precedent: **?**] speed up memoization for functional programs over lists using "chunky decomposition", while differential dataflow databases [?] incrementalize Relational Algebra computations and have achieved some prominence in industry.

?] is the earliest work on incremental evaluation of attribute grammar, motivated by syntax-directed editors; Later work [?] allows reference to non-neighbor attributes. However, these early papers require recomputation immediately after every tree change, whereas web browsers, our target application, batch updates to perform incremental computation only once per frame. The standard in web browsers is instead the Double Dirty Bit algorithm, implemented in all existing web rendering engines. **?**] describes the Double Dirty Bit algorithm in industry publication `web.dev`, and it also features in textbooks [?].

The formal methods community has put significant effort into formalizing web page layout. **?**] proposed using attribute grammars, similar to the DSL in Figure 3, for formalizing web-like layout rules. Later work [?] proposes synthesizing schedules from the attribute grammar rules, including proposals [? ?] to use parallel schedules to further improve layout performance. Parallel schedules could be combined with Spineless Traversal to reduce latency even further. The Cassius project [?] later formalized a significant fragment of CSS 2.1 using an attribute-grammar-like formalism. Our layout implementation is based on Cassius. Later work [?] also proposed using the Cassius formalism to verify web page layouts, including in a modular way [?]. However, none of these works investigated incremental layout. By contrast, the MEDEA project [?] proposed synthesizing incremental layout algorithms by automatically synthesizing dirty bit propogation code. Our work extends MEDEA by exploring optimized incremental traversal algorithms.