Spineless Traversal for Web Layout

MARISA KIRISAME, College of William and Mary, USA
PAVEL PANCHEKHA, College of William and Mary, USA

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: Wireless sensor networks, media access control, multi-channel, radio interference, time synchronization

ACM Reference Format:
Marisa Kirisame and Pavel Panchekha. 2010. Spineless Traversal for Web Layout. *ACM Trans. Web* 9, 4, Article 39 (March 2010), 7 pages. https://doi.org/0000001.0000001

1 Introduction

1.1 Problem

Latency is a major key concern in modern web browser such as chromium, firefox, and safari. Ideally, a 60fps frame rate should be maintained to present smooth user experience as user browse and interact with web pages. When the frame rate objective is not achieved, the user will experience the lag from the web browser and might use another one. (should we hype up the consequence? e.g. maybe a online stock trading app was lagging which cause a trader to miss the optimal time to buy stock)

A significant bottleneck to having good web latency is web page layout, which calculate position and size for each dom node, which then can be rendered into pixels on the screen. When the user interact with the web page, the dom tree will change, and will have to be re-layouted then re-rendered. As the rendering function, the javascript mutator and garbage collector, miscellaneous cost such as inter-process communication, all take significant time, layout got a budget of sub-millisecond. On such tight budget, every cycle count!

1.2 Incremental

To reduce latency introduced by web page layout, browser employ incremental layout algorithm. That is, when the dom tree change, the browser mark down all dependency(field in node) depending on the changed portion. Then, the incremental algorithm traverse the tree to find and fix the

This work is supported by the National Science Foundation, under grant CNS-0435060, grant CCR-0325197 and grant EN-CS-0329609.
Author's addresses: G. Zhou, Computer Science Department, College of William and Mary; Y. Wu and J. A. Stankovic, Computer Science Department, University of Virginia; T. Yan, Eaton Innovation Center; T. He, Computer Science Department, University of Minnesota; C. Huang, Google; T. F. Abdelzaher, (Current address) NASA Ames Research Center, Moffett Field, California 94035.
Authors' Contact Information: Marisa Kirisame, College of William and Mary, Williamsburg, VA, USA; Pavel Panchekha, College of William and Mary, Williamsburg, VA, USA.

ACM 1559-114X/2010/3-ART39
https://doi.org/0000001.0000001

dependency. (maybe: emphasize batchness here. is it important? batchenss mean reps's work is non-applicable, and it also explain why we have separate mark and dirty instead of fixing immediately). Traversing the tree is the bottleneck in incremental layout, as accessing node that does not need fixing incur a large amount of l2 cache miss.

Modern web browsers use summary bits on each node to denote whether the subtree is dirtied(need fixing) or cleaned. With the summary bits, incremental layout need not traverse the whole tree, but still need to traverse the ancestor of dirtied node. This is the spine of the dirtied node.

The algorithm not only have to traverse the spine of the tree, but also the children of the spine, as it have to read the summary node to decide whether it need traversing.

This means that a deep tree or a tree with nodes that have many children is have high latency. Indeed, this problem had been widely observed. For example, google's web performance analytic tool, lighthouse, which have more then 1 million users, use dom tree depth and maximum children node count as performance metrics.

Such metric imply that dom tree are unbalanced, and traversing the spine and it's children (spine+1) is a bottleneck of web page layout, as it access unnecessary node and cause l2 misses.

1.3 Spineless Traversal

Facing this issue, we developed a incremental layout algorithm that only access necessary node. That is, the algorithm only access node where values had been invalidated and need recomputation. Our algorithm(should we name it) achieve such goal by storing all invalidated nodes in a data structure, and jump to the node directly during recomputation. By accessing less node, the algorithm have fewer l2 cache miss and hence better latency then the baseline, the algorithm implemented by major web browsers.

One subtlety of our approach is ordering. More specifically, we want to recompute the dependency of a value, before recomputing the value itself. Otherwise, the value will be immediately invalidated again, and need another recomputation (otherwise the algorithm under-invalidate and is incorrect).

We solved the ordering problem via order maintenance. Order maintenance enable us to assign logical time to each value, such that value initialized later have a larger logical time. Conversely, a larger logical time imply the value was initialized later then a value with a smaller logical time, and cannot depend on it. It is thus safe to process invalidated node in the order indexed by the logical time, and we can store the invalidated node in a priority queue.

Another challenge our approach face is tight computation budget. As web layout have to run in a few milliseconds, every cycle matter. We had meticulously implemented all data structures we deploy, such that the layout algorithm does not allocate, suffer low l2 miss, and branch only when absolutely necessary.

To validate that l2 access on irrelevant node is indeed a latency concern, we had constructed a benchmark by capturing the dom tree generated by interacting with 50 web site. We then run both the baseline and our invalidation algorithm, and our algorithm is 135% faster.

To sum up, our contributions are:

- A spineless traversal algorithm that reduce latency by skipping unnecessary node.
- An efficient implementation of the algorithm such that it run in the computation budget.
- A benchmark comprised of 50 real-world popular website alongside a nontrivial web layout algorithm, complex enough that it handle multiple classic web workload, including linebreaking, flex, display, intrinsic size, min/max width/height, and absolute position. The algorithm amount for over 700 line of code, with about 50 computed property for each node.

2 Web layout

2.1 Tree Traversal

Inside web browser, web page is presented as a dom tree. Inside the dom tree, each node store some intermediate property, such as the type, the html attribute, and css property of the value. It additionally store a doubly linked list of its children, and a parent/two sibling pointer, previous/next.

Web layout convert this tree into the layout tree, a tree of boxes, where parent boxes typically contain children boxes. Later pass such as rendering convert the layout tree into pixels on the screen.

The layout of a node depend on the layout of it's neighbors: it's parent, children, and close sibling(previous/next). For example, the width of the box is the sum of the width of all children, where each children shift the x position by the width of it's previous node (how do we make this accurate?). Each node might also compute some value, such as intrinsic size, to aid computing the position and size of the box. To aid recomputation, we store both the intermediate value and the output of layout in-place, in the tree.

```
def layout_simple(self): # real layout much more complex
  self.x <-
    if has_path(self.previous) then self.previous.x + self.previous.width
    elif has_path(self.parent) then self.parent.x
    else 0
  for c in self.children:
    layout_simple(c)
  self.width <-
    if has_path(self.last) then self.last.width_acc
    else self.basic_width # readonly variable supplied by the dom tree
  self.width_acc <-
    if has_path(self.last) then self.last.width_acc + self.width
    else 0
```

A layout algorithm that compute x position and width of each dom node. The above program is a tree traversal: it walk down the tree then walk up the tree, computing values for each node during the walk.

The above algorithm, while simple, already illuminate multiple key property of layout.

Functional: the tree shape is not modified during layout. The computed fields are also written only once, at initialization, then become readonly.

Local dataflow: only local data and their neighbor is accessed. For example, parent.parent is not accessed, and there are no pointers except the 5 specified by the dom tree.

Fixed controlflow: the control flow is decided by the tree shape and nothing else. In particular, while there are conditional in the expression used to initialized variables, said conditional cannot call id/size().

Driven by our observation, we had designed a DSL suitable for web layout. It capture the 3 facts above to restrict language expressivity, allowing a efficient incremental implementation, while still being expressive enough to implement multiple web layout feature such as the box model, hiding elements, absolute positioning, flex display, and linebreaking.

```
  M(ain) := P_N()...
  P(roc)_N := self.BB_X(); children.P_N(); self.BB_Y()
  BB(BasicBlock)_X := self.V <- T...
  V(ar) := unique symbols
```

```
F(unction) := a predefined set of primitive functions
P(ath) := self | prev | next | parent | first | last
T(erm) := if T then T else T | F(T...) | HasPath(P) | P.V
```

The language describe a list of procedure, executing one after another. Each procedure call itself recursively on all children, initializing variable before and after. The variables can only be computed using limited control flow (branch that merge control flow immediately (formalize?)), primitive functions, haspath query, and variable access. The initialization are grouped into basicblock, so we can talk about multiple initialization as once.

write operational semantic? or is this too trivial so it doesnt need one?

2.2 Incremental layout

The above subsection specified a language for describing a from scratch tree traversal, which can be used to implement web layout. However, when user interact with the web browser, the dom tree will be changed in 3 ways:

(1) Change of value. The html attribute or the css property of a dom node might change. For example, when the user click on a drop down bar, the display property of the drop down bar is set from "none" to "block", indicating that it is now visible.
(2) Dom tree insertion. A dom tree is inserted as a sub tree. This typically occur during loading of a web page, where a temporary dom element is replaced with the final dom element.
(3) Dom tree deletion. A sub tree is deleted from the dom tree. After initial loading, the temporary elements might be deleted.

When the dom tree change, the layout computed will be broken: a value might be dirtied, in that recomputing the value will give another result(todo: too strong, see other formalization). Likewise, a value might be uninitialized after dom tree insertion. The job of the incremental layout algorithm is to find and fix the problems(dirtiness and uninitialization). A key subtlety is that fixing problems yield more problems - when a dirtied value is recomputed, recomputing a value that depend on it will give another result, thus that value had become dirtied. This subtlety dictate that the incremental algorithm should fix a value before fixing its (transitive) dependents. An incremental algorithm that respect such dependency is called regret-free, as it need not fix the same field twice.

We have to detect and mark dirtiness/uninitialization before fixing it.

2.2.1 Marking changed field. When a initialized field F changed it's value, all field depending on it is dirtied. To locate all the field at runtime, we analyze all assignment statement in the program of form "self.X <- Y". For each occurrence of P.F in Y, we construct the reverse-path of P, mapping prev to next/next to prev, first/last to parent, and parent to all children, marking all X on the reverse-path. For all node on the reverse-path, P.F evaluate to the changed field, and thus need to be mark dirtied. (Note: it is possible to use a more precise dependency analysis, such that expression on the branch currently not evaluated is not considered dependency. However this is an orthogonal design decision to the central thesis of the paper, and add too much tracking overhead.)

2.2.2 Marking tree deletion. When a subtree is deleted, all field depending on any field in the root of the tree is marked dirtied, by following reverse-path like the above subsubsection. Note that we do not have to follow the reverse-path of the children, as all access in our language are local. Thus, the only field of a tree that can be directly observed by a node outside of the tree is the root node of the subtree(too long, dont like). Moreover, subtree deletion might change the result of haspath query. For example, deleting the last children of a node cause HasPath(First) and HasPath(Last) query to return false instead of true. Thus, when deleting the first/last/only children

of a tree, reverse-path to the corresponding HasPath query is calculated and field using such query is marked dirtied.

2.2.3 Marking tree insertion. Field are mark dirtied using the same method that mark field for tree deletion. Additionally, the inserted tree is marked uninitialized.

2.2.4 Marking and Fixing. To support marking, each node have a dirtied bit for each field, and a single initialized bit, all initialized to false. Marking dirtiness turn the corresponding bit on. Re-layout then have to find all corresponding dirtied field and uninitialized node, to execute the corresponding assignment to fix the tree. Once a field is fixed, or a node is initialized, the corresponding bit is changed as well to reflect the status of the tree.

A naive incremental implementation can recursively traverse the tree as if rerunning from scratch, but only re-executing the dirtied assignment. While trivial, the algorithm is regret-free: as it execute all dirtied assignment in a from-scratch order, the dependency must be fixed before the dependents. We will be exploiting this insight in our subsequent incremental layout implementation. However, it is still unsatisfactory: multiple traversal on the complete tree will cause too much memory access, and the subsequent l2 miss will be unacceptable.

2.2.5 Spine+1 Traversal. The common approach employed by all major web browsers such as chrome, firefox, safari is the Spine+1 Traversal algorithm.

For each procedure in the layout algorithm, each node maintain a summary bit that dictate whether that procedure need to access the node(and its children) to fix dirtiness/uninitialization. The summary bit is initially off, and when a node is marked, the summary bit is turned on. The parent's summary bit also have to be turned on, and so on, until a node with the bit turn on have been found (when such a node is found, the node and its ancestor already have the bit on, so there is no need to continue traversing up), or until it reached the root node. Each procedural traversal can then skip all node where the corresponding summary bit is off.

```
def set_summary_bit_x(self):
if self.summary_bit_x:
pass # no need to call parent, as the spine must already be set
else:
self.summary_bit_x <- true
if self.parent:
self.parent.set_summary_bit_x()
```

Using this approach, access to non-essential node (nodes that does not need fixing) is greatly reduced. If a node is dirtied, it's ancestor will have the summary bit turn on, leaving a breadcrumb which allow the node to be found.

This breadcrumb is the spine of the node, which is non-essential: the spine does not need fixing, but must be accessed in order to find the essential node that need fixing. Not only does this algorithm have to access the spine, it also have to access all children of the spine, as it must read the summary bit of a node to be able to skip it. The spine alongside its children is named spine+1.

This mean that a dom tree with a deep depth, or a dom node with lots of children, will incur a large spine+1 and kill layout performance. The google chrome team had long observed this problem and correspondingly advice frontend programmer to optimize tree depth and max children size. This advice had been codified into the performance monitoring tool lighthouse.

3 Spineless Traversal

3.1 Order Maintenance

We implemented order maintenance in <two simplified algorithm for linked list>.

Order maintenance is implemented by a doubly linked list of separated doubly linked list. Each node in the linked list contain an unsigned integer, called label for comparison, and each node in the lower-level list contain a pointer to the upper-level list that contain it.

A order maintenance node is a node of the lower linked list.

Two order maintenance node can be compared by comparing the upper level label and (if equal) the lower level label.

3.2 Priority Queue

Our priority queue is implemented via a binary heap where each element is implicit. That is, all elemented are stored in an array, and node at index x have children at node at index 2x+1 and 2x+2. The root node live at index 0.

Huh this is pretty vanilla, maybe should not even be a section

3.3 Putting it together

With order maintenance and priority queue, we can implement a spineless traversal algorithm.

The algorithm utilize order maintenance as logical time. The algorithm maintain a global OM as the current time, and each field in each node have a OM to denote when is it initialized. More precisely, everytime a field is initialized, the counter advance, and the old counter is assigned to the OM of that field.

Fields in nodes also have dirty bit but not recursive dirty bit. When a node is freshly dirty (the bit is set from false to true), the node and the field name is inserted into the priority queue, indexed by the OM node. should we talk about dirty bit as central or as an optimization?

During recomputation, node/field pair are popped from the priority queue and repaired 1-by-1. Just like dirty bit, reevaluation will also dirty more fields, and such fields are also pushed into the queue.

4 Implementation

4.1 HTML features

- visibility (display)
- position (static vs absolute)
- line breaking
- flex
- box model
- intrinsic width/height
- fixed width/height
- min/max width/height

We also have some features that is too small for a bullet point (e.g. image with width and not height/vice versa), which should not be talked about in detail, but i still think we should brief over. how should i structure it?

4.2 Compiler optimization

4.2.1 destringification.

4.2.2 defunctionalization.

4.2.3 field packing.

4.3 Micro Optimization

branchless compare
custom 32bit allocator for order maintainence

5 Eval

how we gathered the 50 website
tree diffing algorithm
measurement (readtsc libperfm)
machine
numbers
argue geomean is correct
talk about good/bad example

6 RW

thomas reps
sac
yu feng

7 Conclusion

Received February 2007; revised March 2009; accepted June 2009