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), 6 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' addresses: Marisa Kirisame, College of William and Mary, 104 Jamestown Rd, Williamsburg, VA, 23185, USA; Pavel Panchekha, College of William and Mary, 104 Jamestown Rd, Williamsburg, VA, 23185, USA.

1559-1131/2010/3-ART39 $15.00
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 Attribute Grammar

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(self.previous) then self.previous.x + self.previous.width
    elif has(self.parent) then self.parent.x else 0
  for c in self.children:
    layout-simple(c)
  self.width <- if has(self.last) then self.last.width-acc else 0
 self.width-acc <- if has(self.last) then self.last.width-acc + self.width else 0
```

A layout algorithm that compute x position and width of each dom node.

As compute is local, that is, as value of a node depend on value of its neighbor, previous work such as (cite) model web layout as attribute grammar. (how detailed should i be in at explaining AG?) However, as we are uninterested in the scheduling aspect of attribute grammar, we assumed the user had already schedule computation into recursive traversal of the tree, there-and-back-again style.

2.2 There And Back Again

The layout algorithm execute a sequence of taba-function on the root of the dom tree. Each taba-function can initialize some values using it's field or that of it's 5 references (or the nullness of the 5 references). It then will recursively invoke itself on each of the children, in the list order, and can initialize values with the same restriction as before.

def id/size(self): self.id <- if has(prev) then prev.id+1 else if has(parent) then parent.id+1 else 0; for c in self.children: c.id/size() self.size <- if has(last) then last.sizeacc+1 else 1 self.sizeacc <- if has(prev) then prev.sizeacc + self.size else self.size

The above figure is a simple taba program that assign a unique id and calculate the size of each node.

Our formalization had deliberately introduced three key constraint.

Immutability constraint: the tree shape can not be modified during relayout. The computed fields can likewise only be written once at initialization. Such constraint is typical for incremental computation (cite adapton/memoization/sac/differential dataflow) as incremental computation have to replay old computation.

Data constraint: only local data and their neighbor can be accessed. For example, parent.parent is an illegal path and cannot be read/written to. This constraint simplify dependency tracking as it limit the amount of direct dependency of a field to the node's neighbor. Note that this constraint can be circumvented by introducing auxiliary variable.

Control constraint: 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(), or other taba-function. This try to limit the work done in each expression, as recomputation work at the granularity of each variable initialization and not (sub)expression calculation.

The figure presented above is a simplification of the web layout process.

web layout compute position and size for each dom node.

Below we give a syntax and semantics of the DSL megatron, and explain the design rational.

M(ain) := P_N()...
P(roc)_N := self.BB_X(); children.P_N(); self.BB_Y()
BB(BasicBlock)_X := self.N <- T...
V(ar) := unique symbols
F(unction) := a predefined set of primitive functions
P(ath) := self | prev | next | parent | first | last
T(erm) := V | if T then T else T | F(T...) | P.V

We assumed that web page layout can be implemented in attribute grammar, and the attribute grammar have been scheduled into multiple 'there and back again' pass.

That is, web page layout call a sequence of mutating function, where each function:

- compute value for field x0, x1, x2...
- recursively invoke itself for each of the children, left to right.
- compute value for field y0, y1, y2...

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

2.3 Dirtying

For each field in each node: keep a dirty bit, and a recursive dirty bit.

Dirty bit is set iff the field is dirtied and need recomputing.

Recursive dirty bit is set iff any of such field in descendent is dirtied.

A field is dirtied if any of its dependency is modified. Inversely, when a field is modified, it's dependent is dirtied. This can be compute rather efficiently, and the dirty bit is setted. The recursive dirty bit is setted via a recursive traversal up the spine, stopping until the said recursive bit is already set.

def set-recursive-dirty-bit-x(self): if self.recursive-dirty-bit-x: pass no need to call parent, as the spine must already be set else: self.recursive-dirty-bit-x <- true if self.parent: self.parent.set-recursive-dirty-bit-x()

Re-layout is done by calling the incremental version of each taba function. Like the non-incremental version, the taba function recursively invoke itself on each of the children, executing a tree-traversal. Unlike the non-incremental version, however, only dirtied fields is recomputed (then the dirty bit is set back to false), and node where no recursive bit(that correspond to fields initialiized during the function) is on will skip all computation, as the node and it's subtree is clean. Before the function exit, the recursive dirtied bits are toggled off, as they and the subtree have been cleaned.

During the recomputation, more field are modified, and their dependent are dirtied. The corresponding dirty bit and recursive dirty bit is then likewise toggled.

In order to clean a node deep down in the tree, the above algorithm have to traverse the spine, using the recursive dirty bit as breadcrumb, to reach the node.

Furthermore, all children of the spine have to be traversed, as the dirty bit and the recursive dirty bit have to be read to determine if further processing is needed.

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.

2.4 Correctness condition

A field is inconsistent if re-computing it yield a different value.

For the incremental layout algorithm to be correct, all field have to be consistence once the evaluation end. Inconsistency arise from doing too little work.

2.5 Optimality condition

An incremental layout algorithm is sub-optimal if it re-evaluate the same expression twice during a single execution. Suboptimality arise from doing work in the wrong order. Specifically, when a field and its (recursive)dependent is inconsistent, the field must be reevaluated before its dependent. Otherwise, reevaluation of the field might cause the dependent to be inconsistent again, causing extra reevaluation.

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