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), 5 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Major web browser including Chromium, Firefox, and Safari use dirty bit algorithm to implement incremental web layout. Such algorithm maintain boolean bits on each dom node to track whether the node's property or its children's property need recomputation.

To recompute a node's property, dirty bit algorithm need to traverse the spine+1 of the node - that is, the node's recursive parent, and children of said node.

This traversal is the bottleneck of web layout, as it causes frequent l2 cache miss. This is further exacerbated by the fact that dom tree are often imbalanced: they are both deep and wide, causing spine+1 to be greatly larger than that of a balanced binary tree.

We designed an algorithm that need not traverse the spine+1 of the dom node, an implemented it in a DSL, megatron. The algorithm employ order maintenance to track dependency between property, and maintain dirtied element that need recomputing in a priority queue, indexed by order maintenance. This remove the need to traverse spine+1 while still only execute the minimal amount of recomputation necessary.

Our web layout algorithm is 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.

Megatron compile the web layout algorithm into incremental, native and machine-efficient program, using the dirtybit and the spineless algorithm. It is then benchmarked under 50 website,

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

including multiple popular websites such as Google, Twitter, and Amazon. The spineless algorithm execute 135% faster then the dirty bit one.

2 BACKGROUND

2.1 Web page layout/rendering

A web page is structured into a dom tree. Then, properties on the dom tree is computed according to html attribute and css property of the tree.

Web browser then layout the web page, by computing data for each node, such as x position, y position, width, height, and value. After such computation, it is then rendered into pixels on the screen.

As user interact with the web page, the dom tree will be modified to reflect the interaction. When this happens, the dom-tree needed to be re-layouted and re-rendered. Major web browser employ incremental algorithm to avoid recomputing from scratch everytime interaction happens.

Since web page layout is only a small part of web browser, and since web browser strive to achieve 60fps(cite idle time gc), web page layout should take a few ms to incrementally re-layout a page.

2.2 There and Back again

Web page layout had been modeled via attribute grammar. (cite that UCB guy). (why is it enough? sounds pretty weak argument.). We had further assumed that the scheduling problem of attribute grammar had been solved, either by a program or by a programmer.

This mean that we model web page layout as multiple there-and-back-again (taba) pass. More specifically, each element in the dom tree contain:

5 possibly null references - the parent, previous/next sibling, first/last child.

A record of read only information passed in as input, that are of primitive values (int, string, float). These include the dom node type as a string, html attribute, and css property.

A record of uninitialized primitive values. They cannot be read before initializing, and no modification can occur beside from an initialization.

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.

2.3 Dirty Bit Algorithm

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.

2.4 Spine+1

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.

2.5 Example

Wikipedia

Hover

I'm confused. Why do we actually need recomputation at this place? display is set during hovering but what depend on display? Isnt it render-only?

3 LANGUAGE(4)

3.1 Attribute Grammar

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?

3.2 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.

3.3 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.

4 SPINELESS TRAVERSAL

4.1 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.

4.2 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

4.3 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.

5 IMPLEMENTATION

5.1 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?

5.2 Compiler optimization

5.2.1 destringification.

5.2.2 defunctionalization.

5.2.3 field packing.

5.3 Micro Optimization

branchless compare

custom 32bit allocator for order maintainence

6 EVAL

how we gathered the 50 website

tree diffing algorithm

measurement (readtsc libperfm)

machine

numbers

argue geomean is correct

talk about good/bad example

7 RW

thomas reps

sac

yu feng

8 CONCLUSION

Received February 2007; revised March 2009; accepted June 2009