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), 10 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

To fix the problem of traversing unnecessary node, we developed a Spineless Traversal algorithm. The algorithm store all dirtied node into a priority queue, indexed by the time order, to ensure the

regret-free property. During incremental layout, fields are popped one by one from the priority queue, and their values recomputed. The incremental layout is finished when all elements in the queue is popped and processed.

3.1 Time Order

We implemented Time Order using Order Maintenance in <two simplified algorithm for linked list>.

Order Maintenance is a data structure that maintain total order between node of the data structure. It is implemented with the three following API:

(1) Compare: OM -> OM -> Ordering, which establish a total order between all OM
(2) Head: OM, which return the smallest OM
(3) Insert: OM -> OM, this is the heart of order maintenance, which create a new OM node right next to an already existing one. This mean that, the returned node is only larger then the original node and every value smaller then the original node, but smaller then every node.

Compare to the typical use of integer or floating point as time, order maintenance allow creation of any new timepoint in the middle of two other timepoint, by calling insert. On the other hand, one might attempt to do this on integer by taking the average of two timepoint, but will quickly run into precision issue.

The order maintenance datastructure is a doubly-linked-list of doubly-linked-list structure, where a order maintenance node(OM) is a node of the inner list. Additionally, the inner list hold a pointer to the outer list node which contain it, and each list node (inner or outer) hold an unsigned integer.

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. Comparison is implemented as lexical ordering on the outer level list then the lower level list. On insertion, a new node is inserted after the input node, and the label is the average of the neighboring label. If the two neighboring label only differ by 1, the gap is not big enough to fit a new label, and the datastructure rebalance itself, reassigning new label to existing values to make space. During relabeling, new outer-level-node might be created to ensure there is enough gap between two element. As rebalancing is not a bottleneck, we will not focus on that.

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.

While our priority queue is very standard, one noteworthy point is that the priority queue hold no duplicate element. That is, each dirtied field will only be inserted once.

This is because a field is only pushed onto the queue when a node is initially dirtied - that is, the dirty bit is set from false to true. When there are diamond dependency, there will be subsequent assignment of the same dirty bit to true, but as the bit is already on, it will not be re-enqueued.

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.

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.

3.4 Handling Insertion

4 Implementation

One big challenge faced by our work is, can data structure manipulation compete on performance with setting summary bit?

While the summary bit algorithm does incur extra L2 cache miss, it only use does rudimentary operation on bit. On the other hand, our spineless algorithm have to maintain nontrivial data structure. As incremental layout often complete in thousands or even hundreds of cycle, every cycle count. To fit into the computation budget, we implemented the two data structure, order maintenance and priority queue meticulously(word too big?), avoiding allocation, cache miss and branch misprediction as much as possible.

4.1 Optimizing Order Maintenance

4.1.1 Pointer Compression and Custom Allocator. For each level of the list, we used a hand-written, single-threaded pooling allocator that only support malloc/free of the fixed size. Further more, as we know the number of elements are smaller then $2\hat{3}2$, a 32 bit pointer is used (todo: 16 bit?). Implementation wise speaking, the allocator is made from a dynamic array of block size, holding all values, and a dynamic array of 32 bit index, pointing to free space in the previous array. This design pack nodes of the list tightly together, eliminating fragmentation completely so malloc/free does no search at all, and compress pointer to 32 bits, so more node fit in a cache line and incur less cache miss. The allocator had pre-allocate typically enough space to avoid resizing at runtime.

```
template<typename T, typename P=uint32_t>
struct Allocator {
  std::vector<T> pool;
  std::vector<P> freed;
  T* addressof(P p) { return freed[p]; }
  P malloc() {
   if (freed.empty()) {
     pool.push_back(T());
     return pool.size() - 1;
   } else {
     P p = freed.back();
     freed.pop_back();
     return p;
   }
  }
  void free(P p) {
   freed.push_back(p);
  }
}
```

4.1.2 Branchless Comparison. As order maintenance is used to index value in a priority queue, the comparison is executed extremely frequently. The comparison will fetch memory mostly from l2 (todo: measure?) as most nodes are on the lower level vs higher level. Moreover the comparison is done using quite few instruction, so the bottleneck is the pipeline stall induced by the conditional. Moreover, branch predictor can not help here, as the comparison result might goes both way (I want to say 50 50 but not true, goes both way seems a bit weak). To fix this, we implemented a comparison function that is completely branchless, relying on conditional move or bit operation depending on the architecture to do the trick. On our x86 machine which have conditional move available, it is compiled to conditional move. (maybe remove the bit operation branch?) The comparison take 5 cycles sync with lazer on his microbenchmark.

```
SignedLabel operator<=>(const _l2_node &l, const _l2_node &r) {
  Label lpl = l.parent->label;
  Label rpl = r.parent->label;

  // A readable but inefficient implementation
  // if (lpl != rpl) {
  //   return lpl - rpl;
  // } else {
  //   return l.label - r,label;
  // }

  // Comparing the result of the lower level label
  SignedLabel result1 = static_cast<SignedLabel>(l.label - r.label);
  // Comparing the result of the higher level label
  SignedLabel result2 = static_cast<SignedLabel>(lpl - rpl);

#ifndef CMOV_SUPPORTED
  // A 64 bit integer where every bit is 1 (if lpl != rpl) or 0 (if lpl == rpl)
  SignedLabel mask1 = static_cast<SignedLabel>(lpl == rpl) - 1;
  // selecting result1 or result2 base on mask
  return ((result1 & ~mask1) | (result2 & mask1));
#else
  SignedLabel result;

  asm volatile("cmp %1, %2\n" \\ test for equalness of lpl and rpl
               "cmove %3, %0\n" \\ if equal move result1 into result
               "cmovne %4, %0\n" \\ if notequal move result2 into result
               : "=&r"(result)
               : "r"(lpl), "r"(rpl), "r"(result1), "r"(result2));

  return result;
#endif
}
```

4.1.3 Optimizing priority queue. I think the code there could likewise be made more branchless. However I need to do it.

4.2 Web layout implementation

To evaluate our spineless traversal against the standard approach, we had implemented a web layout engine in our tree traversal language. While supporting all html features will be way overscoped for our work, we had implemented a wide variety of html features, which are core to web browsing experience, and many of them have non-subtle incremental behavior. All in all, the non-incremental layout algorithm consists of 750 lines of (non-incremental) code, and compute 40 fields for each node. Below is a list of major features we had implemented. subsubsection or itemize?

- box model Each dom node is represented as box with x, y, width and height. The box of a dom node typically contain all the boxes of its children. Likewise, two dom node with no ancestor relation will typically have a separated dom node.
  The width and height of the box is typically computed from the width and height of its children, but css property can set a fixed width/height, or a min/max width/height. should we implement margin?
  When this occur, the fixed box act as a 'stopping point' for recomputation: the change in the box's children's layout will not effect the box's layout, as it is dictated by an external source. talk more about stopping recomputation in technical section web layout
- line breaking Without line breaking, the children of a dom node is layout as a single line, with each subsequent box having a larger x value, while the y value stay the same.
  Some node might cause line break, introducing a new line and continuing the layout process there.
  The height of each line is the max height of a box of the line, and the width of each line is the sum of width of box.
  The width of the parent box is max of width of each line and the height of the parent box is the sum of height of each line.
  Incrementally speaking, when a new node is added, but it does not have the highest height, nor adding it make the line the longest line, the box consume existing space only, without any need for other boxes to make way. Thus other layout boxes need no change.
- Display A node might have the css property display set to "none". This indicate that the node and all its subchildren will have a box of size 0 and is thus invisible. Such node are typically metadata node such as javascript or css file.
  Incrementally speaking, such node are typically lazily loaded: the website will display a preview version of the web page, loading such extra file in the background. Once the loading is finished, it is added to the dom tree. When loaded, nodes with display="none" should not change the layout in anyway, thus little recomputation is needed.
- position (static vs absolute) Another usage of display is to setup hover boxes. While on wikipedia, when hovering above a hyperlink to another wikipedia article, the website will pop up a preview of the article, shadowing part of the existing document. This is implemented with a box of absolute positioning, such that it is freed from typical box model relationship and is displayed directly on the screen. The box already existed when the page complete loading, only with display="none" set to hide it; hovering set it to some visible value temporarily, and setting it back to none once done.
  Incrementally speaking, absolute positioning does not effect other layout, so change in node of absolute positioning should only recompute up to said box.
- Flex Flex is the most complex feature we supported. To compute flex, some auxiliary values, shrink-to-fit width/height needed to be compute as well. shrink-to-fit width/height is like typical width/height computation, only that it is calculating the minimum dimension needed to fit the node, instead of a default one, which (does what?). Once shrink-to-fit dimensions

for all children is computed, we can compute the 'extra space' for all children, by subtracting box dimension with sum of shrink-to-fit dimensions of children. The children additionally have a css property 'flex-grow' which dictate the portion of the extra space it should inherit. The node finally distribute its extra space, and each children recieved a portion according to its flex-grow divided by the sum of all flex-grow. Additionally, children that have reached its max-width/max-height, should be excluded from the extra-space splitting calculation. check this part

Flex is one of the most complex feature of html, requiring computing multiple auxiliary value, and is full of corner cases. Yet, we have less trouble then we thought implementing the feature. Moreover, while our tree traversal language seems highly restrictive in first sight, it did not provide much hurdle in our flex implementation, and most trouble of flex come from the feature itself. This serve as a stress testing, demonstrating that our modeling stand up to complex feature of the html. what is the incremental implication of flex?

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?

4.3 Compiler optimization

4.3.1 destringification.

4.3.2 defunctionalization.

4.3.3 field packing.

4.4 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