Parallelizing the Browser: Synthesis and Optimization of Parallel Tree Traversals

by

Leo A. Meyerovich

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Rastislav Bodik, Chair
Professor George Necula
Professor Krste Asanovic
Professor David Wessel

Fall 2013

The dissertation of Leo A. Meyerovich, titled Parallelizing the Browser: Synthesis and Optimization of Parallel Tree Traversals, is approved:

Chair _____ Date _____

_____ Date _____

_____ Date _____

_____ Date _____

University of California, Berkeley

Parallelizing the Browser: Synthesis and Optimization of Parallel Tree Traversals

Abstract

Parallelizing the Browser: Synthesis and Optimization of Parallel Tree Traversals

by

Leo A. Meyerovich

Doctor of Philosophy in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Rastislav Bodik, Chair

From low-power phones to speed-hungry data visualizations, web browsers need a performance boost. Parallelization is an attractive opportunity because commodity client devices already feature multicore, subword-SIMD, and GPU hardware. However, a typical webpage will not strongly benefit from modern hardware because browsers were only designed for sequential execution. We therefore need to redesign browsers to be parallel. This thesis focuses on a browser component that we found to be particularly challenging to implement: the layout engine.

We address layout engine implementation by identifying its surprising connection with attribute grammars and then solving key ensuing challenges:

1. We show how layout engines, both for documents and data visualization, can often be functionally specified in our extended form of attribute grammars.

2. We introduce a synthesizer that automatically schedules an attribute grammar as a composition of parallel tree traversals. Notably, our synthesizer is fast, simple to extend, and finds schedules that assist aggressive code generation.

3. We make editing parallel code safe by introducing a simple programming construct for partial behavioral specification: schedule sketching.

4. We optimize tree traversals for SIMD, MIMD, and GPU architectures at tree load time through novel optimizations for data representation and task scheduling.

Put together, we generated a parallel CSS document layout engine that can mostly render complex sites such as Wikipedia. Furthermore, we scripted data visualizations that support interacting with over 100,000 data points in real time.

To You

Hey you! out there in the cold Getting lonely, getting old, can you feel me Hey you! Standing in the aisles With itchy feet and fading smiles, can you feel me Hey you! don't help them to bury the live Don't give in without a fight.

Contents

List of Figures

List of Tables

Acknowledgments

I want to thank my advisor for advising me.

Chapter 1

Introduction

Why Parallel Computing

Why Mechanize Layout

Approach

1.1 Mechanizing Layout Languages with Sugared Attribute Grammars

1.2 A Scheduling Language for Structuring and Verifying Parallel Traversals

1.3 Controlling Automatic Parallelization through Schedule Sketches

1.4 The Design of a Parallel Schedule Synthesizer

1.5 Optimizing Parallel Tree Traversals for Commodity Architectures

1.6 Collaborators and Publications

Chapter 2

Layout Languages as Sugared Attribute Grammars

2.1 Motivation and Approach

Important properties for layout languages and others

- Verified semantics: total definition, linear complexity, change-impact analysis, ...
- Verified implementations
- Implementation complexity of layout lang: program analysis and code generation simplify optimization, tooling, debugging, ...
- Implementation complexity of spec lang: desugaring eliminates costs

Approach

- language for defining tree evaluator that is restricted enough for automation support
- push language expressiveness where needed
- reduce language complexity via desugaring semantics

2.2 The HBox Language as a Classical Attribute Grammar

Example tree with dynamic dependencies

Example static grammar instance

Dynamic evaluator

2.3 Desugaring Modern Constructs

Motivation: Productive Features with Simple Implementations

Interfaces: Lightweight and Reusable Input/Output Specifications

Traits: Reusing Cross-cutting Code

Foreign Functions: Embedded Domain Specific Language

Loops

2.4 Evaluation: Mechanized Layout Features

Rendering: Immediate Mode and Beyond

Non-euclidean: Sunburst Diagram

Charts: Line graphs

Animation and Interaction: Treemap

Flow-based: CSS Box Model

Grid-based: HTML Tables

2.5 Related Work

- loose formalisms: browser impl (C++), d3 (JavaScript), latex formulas (ML)
- restricted formalisms: cassowary and hp, UREs
- AGs: html tables

Chapter 3

A Safe Scheduling Language for Structured Parallel Traversals

3.1 Motivation and Approach

- structure is good for parallelization
- parallelization needs checking
- structured parallelism in layout

3.2 Background: Static Sequential and Task Parallel Visitors

Sequential Visitors

- Knuth: synth and inh
- OAG

Task Parallel Visitors

- FNC-2 / Work stealing

3.3 Structured Parallelism in Visitors

td, bu, in order

(related to distributed?)

concurrent

(old paper: unstructured within visit)

multipass

(any old paper? unstructured within visit)

nested

3.4 A Behavioral Specification Language

Formalism

3.5 Schedule Compilation

Phrase as rewrites working in an EDSL w/ templates

Rewrite rules

3.6 Schedule Verification

Overview

- properties to prove: schedule followed (and complete), dependencies realizable
- structure of proof

Axioms

- axioms
- examples from each

Proof

3.7 Case Studies: Layout as Structured Parallel Visits

Box model

Nested text

Grids

3.8 Related Work

Lang of schedules

- background
- stencils and skeletons: wavefront, ...
- polyhedra

Schedule verification

- compare to OAG etc., looser dataflow/functional langs

Chapter 4

Interacting with Automatic Parallelizers through Schedule Sketching

4.1 Automatic Parallelization: The Good, the Bad, and the Ugly

The Good: Automating Dependency Management

The Bad: Guiding Parallelization

The Ugly: Preventing Serialization

4.2 Holes

4.3 Generalizing Holes to Unification

4.4 Case Studies: Sketching in Action

Show use in CSS and data viz:

- when automatic is fine
- when sketch needed for checking/debugging
- when sketch needed for sharing

4.5 Related Work

- sketch, sketch for concurrent structures
- oopsla paper for individual traversals

Chapter 5

Parallel Schedule Synthesis

5.1 Motivation: Fast and Parameterized Algorithm Design

5.2 Optimized Algorithm: Finding One Schedule

5.3 Optimized Algorithm: Autotuning Over Many Schedules

Alternation Heuristic: Off-by-one Optimality

Enumeration via Incrementalization

5.4 Complexity Analysis and the Power of Sketching

5.5 Evaluation

Speed of synthesis

Success, fail, enumerate

Line counts of extensions

Loss from greedy heuristic

Benefit from autotuning

Chapter 6

MIMD and SIMD Tree Traversals

6.1 Overview

For a full language, statically identified parallelization opportunities still require an efficient runtime implementation that exploits them. In this section, we show how to exploit the logical concurrency identified within a tree traversal to optimize for the architectural properties of two types of hardware platforms: MIMD (e.g., multicore) and SIMD (e.g., sub-word SIMD and GPU) hardware. For both types of platforms, we optimize the schedule within a traversal and the data representation. We innovate upon known techniques in several ways:

1. **Semi-static work stealing for MIMD:** MIMD traversals should be optimized for low overheads, load balancing, and locality. Existing techniques such as work stealing provide spatial locality and, with tiling, low overheads. However, dynamic load balancing within a traversal leads to poor temporal locality across traversals. The processor a node is assigned to in one traversal may not be the same one in a subsequent traversal, and as the number of processors increases, the probability of assigning to a different one increases. Our solution dynamically load balances one traversal and, due to similarities across traversals, successfully reuses it.

2. **Clustering traversals for SIMD:** SIMD evaluation is sensitive to divergence across parallel tasks in instruction selection. Visits to different types of tree nodes yield different instruction streams, so naıve vectorization fails for webpages due to their visual variety. Our insight is that similar nodes can be semi-statically identified. Thus *clustered* nodes will be grouped in the data representation and run in SIMD at runtime.

3. **Automatically staged parallel memory allocation to efficiently combine SIMD layout and SIMD rendering:** We optimized the schedule of memory allocations in the layout computation into an efficient parallel prefix sum. Otherwise, parallel dynamic memory allocation requests would contend over the free memory buffer and void GPU performance benefits. We automated the optimization by reducing the scheduling

problem to attribute grammar scheduling and automatically performing the reduction through macro expansion.

Our techniques are important and general. They overcame bottlenecks preventing seeing any speedup from parallel evaluation for webpage layout and data visualization. Notably, they are generic to computations over trees, not just layout. An important question going forward is how to combine them as, in principle, they are complementary.

6.2 MIMD: Semi-static work stealing

Scheduling

Data representation

Evaluation

6.3 SIMD Background: Level-Synchronous Breadth-First Tree Traversal

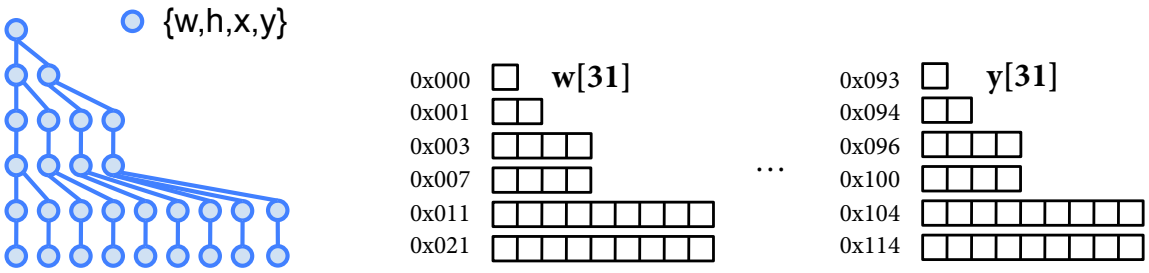
The common baseline for our two SIMD optimizations is to implement parallel preorder and postorder tree traversals as level-synchronous breadth-first parallel tree traversals. Reps first suggested such an approach to parallel attribute grammar evaluation [[CITE]], but did not implement it. Performance bottlenecks led to us deviate from the core representation used by more recent data parallel languages such as NESL [[CITE]] and Data Parallel Haskell [[CITE]]. We discuss our two innovations in the next subsections, but first overview the baseline technique established by existing work.

The naıve tree traversal schedule is to sequentially iterate one level of the tree at a time and traverse the nodes of a level in parallel. A parallel preorder traversal starts on the root node's level and then proceeds downwards, while a postorder traversal starts on the tree fringe and moves upwards (Figure 6.1 6.1a). Our MIMD implementation, in contrast, allows one processor to compute on a different tree level than another active processor. In data visualizations, we empirically observed that most of the nodes on a level will dispatch to the same layout instructions, so our naıve traversal schedule avoids instruction divergence.

The level-synchronous traversal pattern eliminates many divergent memory accesses by using a corresponding data representation. Adjacent nodes in the schedule are collocated in in memory. Furthermore, individual node attributes are stored in *column* order through a array-of-structure to structure-of-array conversion. The conversion collocates individual attributes, such as the width attribute of one node being stored next to the width attribute of the node's sibling (Figure 6.1c). The index of a node in a breadth-first traversal of the tree is used to perform a lookup in any of the attribute arrays. The benefit this encoding is that, during SIMD layout of several adjacent nodes, reads and writes are coalesced into

```
void parPre(void (*visit)(Prod &), List<List<Prod>> &levels) {
  for (List<Prod> level in levels)
        parallel_for (Prod p in level)
              visit(p)
}
void parPost(void (*visit)(Prod &), List<List<Prod>> &levels) {
  for (Array<Prod> level in levels.reverse())
        parallel_for (Prod p in level)
              visit(p)
}
```

(a) **Level-synchronous Breadth-First Traversal**



(b) **Logical Tree** (c) **Tree Representation**

Figure 6.1: **SIMD tree traversal as level-synchronous breadth-first iteration with corresponding structure-split data representation.**

bulk reads and writes. For example, if a layout pass adds a node's padding to its width, several contiguous paddings and several contiguous widths will be read, and the sum will be stored with a contiguous write. Such optimizations are crucial because the penalty of non-coalesced access is high and, for layout, relatively few computations occur between the reads and writes.

Full implementation of the data representation poses several subtleties.

- **Level representation.** To eliminate traversal overhead, a summary provides the index of the first and last node on each level of a tree. Such a summary provides data range information for launching the parallel kernels that evaluate the nodes of a level as well as the information for how to proceed to the next level.

- **Edge representation.** A node may need multiple named lists of children, such as an HTML table with a header, footer, and an arbitrary number of rows. We encode the table's edges as 3 global arrays of offsets: header, footer, and first-row. To support iterating across rows, we also introduce a 4th array to encode whether a node is the last sibling. Thus, any named edge introduces a global array for the offset of the pointed-to node, and for iteration, a shared global array reporting whether a node at a particular index is the end of a list.

- **Memory compression.** Allocating an array the size of the tree for every type of node attribute wastes memory. We instead statically compute the maximum number

of attributes required for any type of node, allocate an array for each one, and map the attributes of different types of nodes into different arrays. For example, in a language of HBox nodes as Circle nodes who have attributes 'r' and 'angle', 4 arrays will be allocated. The HBox requires an array for each of the attributes 'w', 'h', 'x', and 'y' while the Circle nodes only require two arrays. Each node has one type, and if that that type is HBox, the node's entry in the first array will contain the 'w' attribute. If the node has type Circle, the node's entry in the first entry will contain the 'r' attribute.

- **Tiling.** Local structural mutations to a tree such as adding or removing nodes should not force global modifications. As most SIMD hardware has limited vector lengths (e.g., 32 elements wide), we split our representation into blocks. Adding nodes may require allocation of a new block and reorganization of the old and new block. Likewise, after successive additions or deletions, the overall structure may need to be compacted. Such techniques are standard for file systems, garbage collectors, and databases.

In summary, our basic SIMD tree traversal schedule and data representation descend from the approach of NESL [[CITE]] and Data Parallel Haskell [[CITE]]. Previous work shows how to generically convert a tree of structures into a structure of arrays. Those approaches do not support statically unbounded nesting depth (i.e., tree depth), but our system supports arbitrary tree depth because our transformation is not as generic.

A key property of all of our systems, however, is that the structure of the tree is fixed prior to the traversals. In contrast, for example, parallel breadth-first traversals of graphs will dynamically find a minimum spanning tree [[CITE]]. Such dynamic alternatives incur unnecessary overheads when performing a sequence of traversals and sacrifice memory coalescing opportunities. Layout is often a repetitive process, whether due to multiple tree traversals for one invocation or an animation incurring multiple invocations, so costs in creating an optimized data representation and schedule are worth paying.

6.4 Input-dependent Clustering for SIMD Evaluation

Once the tree is available, we automatically optimize the schedule for traversing a tree level to avoid instruction divergence. **TODO overview subsection**

The Problem

The problem we address stems from layout being a computation where the instructions for each node are heavily input dependent. The intuition can be seen in contrasting the visual appearance of a webpage vs. a data visualization. Different parts of a webpage look quite different from one another, which suggests sensitivity to values in the input tree, while a visualization looks self-similar and thus does not use widely different instructions for different nodes. For an example of divergence, an HBox's width is the sum of its children

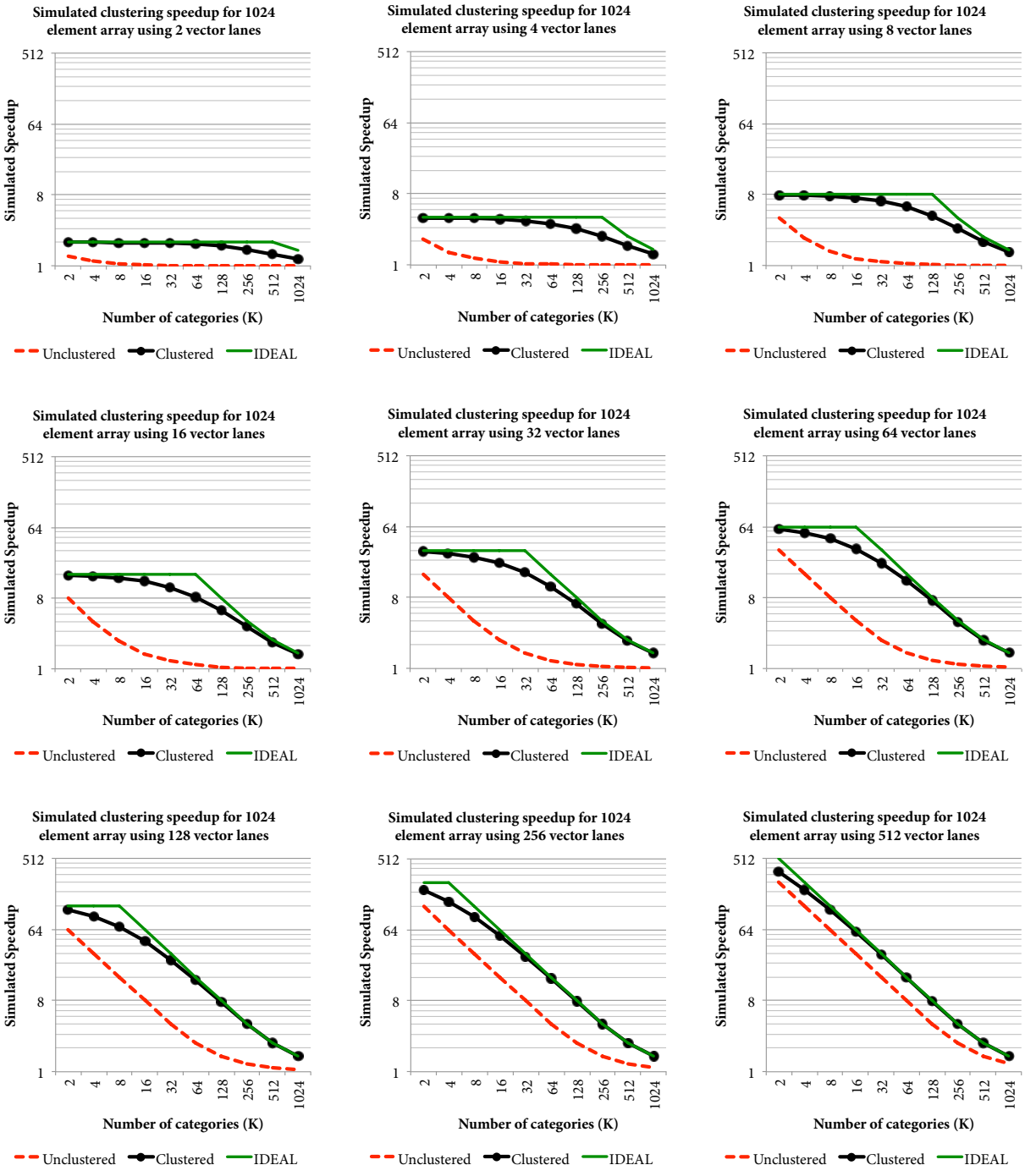


Figure 6.2: **blah**

```
void parPreClustered(void (*visit)(Prod &), List<List<Array<Prod>>> &levels) {
  for (List<Prod> level in levels)
        for (Array<Prod> cluster in level)
              parallel_for (Prod p in cluster)
                    visit(p)
}
```

Figure 6.3: **ASDF.**

widths, while a VBox's is their maximum. The visit to a node (Figure 7.3c) will diverge in instruction selection based on the node type.

We ran a simulation to measure the performance cost of the divergence. Assuming a uniform distribution of types of nodes in a level, as the number of types of nodes go up (K), the probability that all of the nodes in a group share the same instructions drops exponentially. Figure 6.2 shows the simulated speedup for SIMD evaluation over a tree level of 1024 nodes on computer architectures with varying SIMD lengths. The x axis of each chart represents the number of types and the y axis is the speedup. As the number of choices increase, the benefit of the naıve breadth-first schedule (red line) decreases. It is far from the ideal speedup, which we estimated as a function of the SIMD length of the architecture (maximal parallel speedup, contributing the horizontal portion of the green lines) and the expected number of different categories (mandatory divergences, contributing the diagonal portion).

Code Clustering

Our solution is to cluster nodes of a level based on the values of attributes that influence the flow of control. SIMD evaluation of the nodes in a cluster will be free of instruction divergence. Furthermore, by changing the data representation to match the clustered schedule, memory accesses will also be coalesced. We first focus on applying the clustering transformation to the code.

Figure 6.3 shows the clustered evaluation variant of the MIMD $parPre$ traversal of Figure 7.3. The traversal schedule is different because the order is based on the clustering rather than breadth-first index. Changing the order is safe because the original loop was parallel with no dependencies between elements. Computing over clusters guarantees that all calls to a visit dispatch function in the parallel inner loop (e.g., of $visit1$) will branch to the same switch statement case. This modified schedule avoids instruction divergence.

Our loop transformation can be understood as a use of loop unswitching, which is a common transformation for improving parallelization. Loop unswitching lifts a conditional out of a loop by duplicating the loop inside of both cases of the conditional. Clustering establishes the invariant of being able to inspect the first item of a collection sufficing for performing unswitching for a loop over all of the items. Figure 6.4 separates our transformation of $visit1$ (Figure 7.3) into using the same exemplar for the dispatch and then loop unswitching.

```
Prod firstProd = cluster[0]
parallel_for (prod in Cluster) {
  switch (firstProd.type) {
    case S → HBOX: break;
    case HBOX → ϵ:
      HBOX.w = input(); HBOX.h = input(); break;
    case HBOX → HBOX1 HBOX2:
      HBOX0.w = HBOX1.w + HBOX2.w;
      HBOX0.h = MAX(HBOX1.h, HBOX2.h);
      break;
  }
}
```

(a) **Clustered dispatch.**

```
Prod firstProd = cluster[0]
  switch (firstProd.type) {
    case S → HBOX: break;
    case HBOX → ϵ:
      parallel_for (prod in Cluster) {
          HBOX.w = input(); HBOX.h = input();
      }
      break;
    case HBOX → HBOX1 HBOX2:
      parallel_for (prod in Cluster) {
        HBOX0.w = HBOX1.w + HBOX2.w;
        HBOX0.h = MAX(HBOX1.h, HBOX2.h);
      }
      break;
  }
}
```

(b) **Unswitched dispatch**.

Figure 6.4: **Loop transformations to exploit clustering for vectorization.**

Clustering is with respect to input attributes that influence control flow, which may be more than the node type. For example, in our parallelization of the C3 layout engine, we found that the engine author combined the logic of multiple box types into one visit function because the variants shared a lot of code. He instead used multiple node flags to guide instruction selection. Both the node type and various other node attributes influenced control flow, and therefore our clustering condition was on whether they were all equal. Using all of the attributes led to too granular of a clustering condition, so we manually tuned the choice of attributes.

Data Clustering

The data representation should be modified to match the clustering order. The benefit is coalesced memory accesses, but overhead costs in performing the clustering should be considered.

Our algorithm matches the data representation order to the schedule by placing nodes of a cluster into the same contiguous array. Parallel reads and are coalesced, such as the inspection of the node type for the visit dispatch. Parallel writes are likewise coalesced.

Reordering data is expensive as all of the data is moved. In the case of our data visualization system, we can avoid the cost because the data is preprocessed on our server. For webpage layout, the client performs clustering, which we optimize enough such that the cost is outweighed by the subsequent performance improvements.

We optimize reordering with a simple parallel two-pass technique. The first pass traverses each level in parallel to compute the cluster for each node and tabulate the cluster sizes for each tree level. The second pass again traverses each level in parallel, and as each node is traversed, copies it into the next free slot of the appropriate cluster. Even finer-grained

parallelization is possible, but this algorithm was sufficient for lowering reordering costs enough to be amortized.

Evaluation

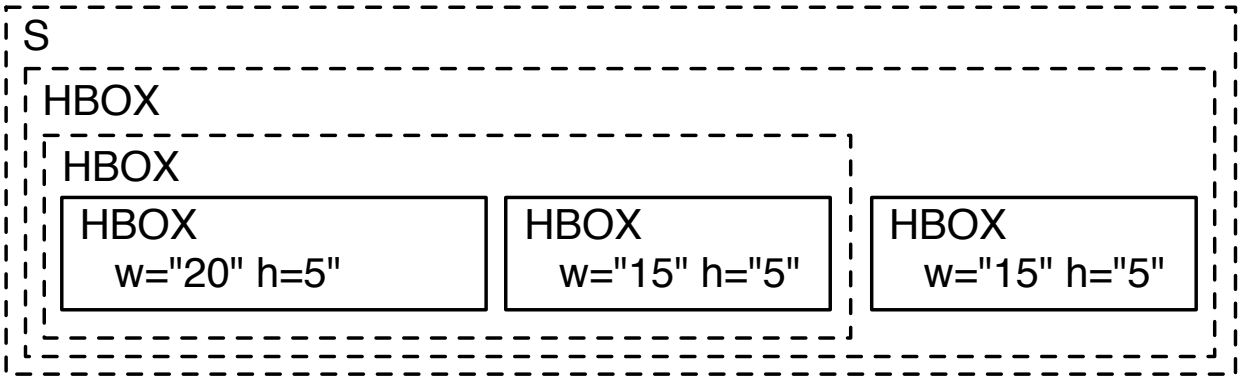
1. structure split BFS benefit for data viz and webpages
2. clustering limit study and benchmarks
3. CPU vs GPU for layout, then factor in rendering loop

6.5 Related Work

1. representation The representation might be further compacted. For example, the last two arrays will have null values for Circle nodes. Even in the case of full utilization, space can be traded for time for even more aggressive compression [[CITE rinard]]
2. duane
3. trishul
4. gnu irregular array stuff

Chapter 7

Conclusion



(a) **Input tree.** Only some of the x, y, w, and h attributes are specified.

$S \rightarrow HBOX$
{ HBOX.x = 0; HBOX.y = 0 }

$HBOX \rightarrow \epsilon$
{ HBOX.w = input$_w$(); HBOX.h = input$_h$() }

$HBOX_0 \rightarrow HBOX_1\ HBOX_2$
{ HBOX_1.x = HBOX_0.x;
HBOX_2.x = HBOX_0.x + HBOX_1.w;
HBOX_1.y = HBOX_0.y;
HBOX_2.y = HBOX_0.y;
HBOX_0.h = max(HBOX_1.h, HBOX_2.h);
HBOX_0.w = HBOX_1.w + HBOX_2.w }

(b) **Attribute grammar for a language of horizontal boxes.**

$AG \rightarrow (Prod\ \{\ Stmnt?\ \})^*$

$Prod \rightarrow V \rightarrow V^*$

$Stmnt \rightarrow Attrib = id(Attrib^*) \mid Attrib = n \mid Stmnt\ ;\ Stmnt$

$Attrib \rightarrow id.id$

(c) **Language of attribute grammars.**

Figure 7.1: For a language of horizontal boxes: (a) input tree to solve and (b) attribute grammar specifying the layout language. Specification language of attribute grammars shown in (c).

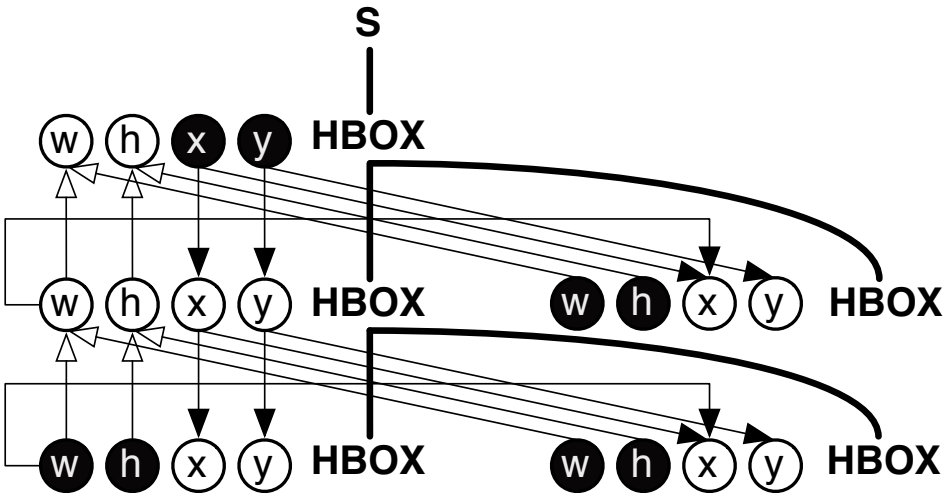


Figure 7.2: **Data dependencies.** Shown for constraint tree in Figure 2 (a). Circles denote attributes, with black circles being input() sources. Thin lines show data dependencies and thick lines show production derivations.

```
parPost
  HBOX0 → HBOX1 HBOX2 { HBOX0.w HBOX0.h }
  HBOX → ϵ { HBOX.w HBOX.h }
;
parPre
  S → HBOX { HBOX.x HBOX.y }
  HBOX0 → HBOX1 HBOX2
    { HBOX1.x HBOX2.x HBOX1.y HBOX2.y }
```

(a) **One explicit parallel schedule for** `H-AG` **.**

```
void parPre(void (*visit)(Prod &), Prod &p) {
  visit(p);
  for (Prod rhs in p)
    spawn parPre(visit, rhs);
  join;
}
void parPost(void (*visit)(Prod &), Prod &p) {
  for (Prod rhs in p)
    spawn parPost(visit, rhs);
  join;
  visit(p);
}
```

(b) **Naïve traversal implementations** with Cilk's [**cilk**] **spawn** and **join**.

```
void visit1 (Prod &p) {
  switch (p.type) {
    case S → HBOX: break;
    case HBOX → ϵ:
      HBOX.w = input(); HBOX.h = input(); break;
    case HBOX → HBOX1 HBOX2:
      HBOX0.w = HBOX1.w + HBOX2.w;
      HBOX0.h = MAX(HBOX1.h, HBOX2.h);
      break;
  }
}
void visit2 (Prod &p) {
  switch (p.type) {
    case S → HBOX:
      HBOX.x = input(); HBOX.y = input(); break;
    case HBOX → ϵ: break;
    case HBOX → HBOX1 HBOX2:
      HBOX1.x = HBOX0.x
      HBOX2.x = HBOX0.x + HBOX1.w;
      HBOX1.y = HBOX0.y
      HBOX2.y = HBOX0.y
      break;
  }
}
parPost(visit1, start); parPre(visit2, start);
```

(c) **Scheduled and compiled layout engine for** `H-AG` **.**

Sched → *Sched* ; *Sched* | *Sched* || *Sched* | *Trav*

Trav → *TravAtomic Visit**{(*TravAtomic* ↦ *Visit**)*}?

TravAtomic → **parPre** | **parPost** | **recursive**

Visit → *Prod* { *Step** }

Step → *Attrib* | **recur** *V*

(d) **Language of schedules** (without holes)

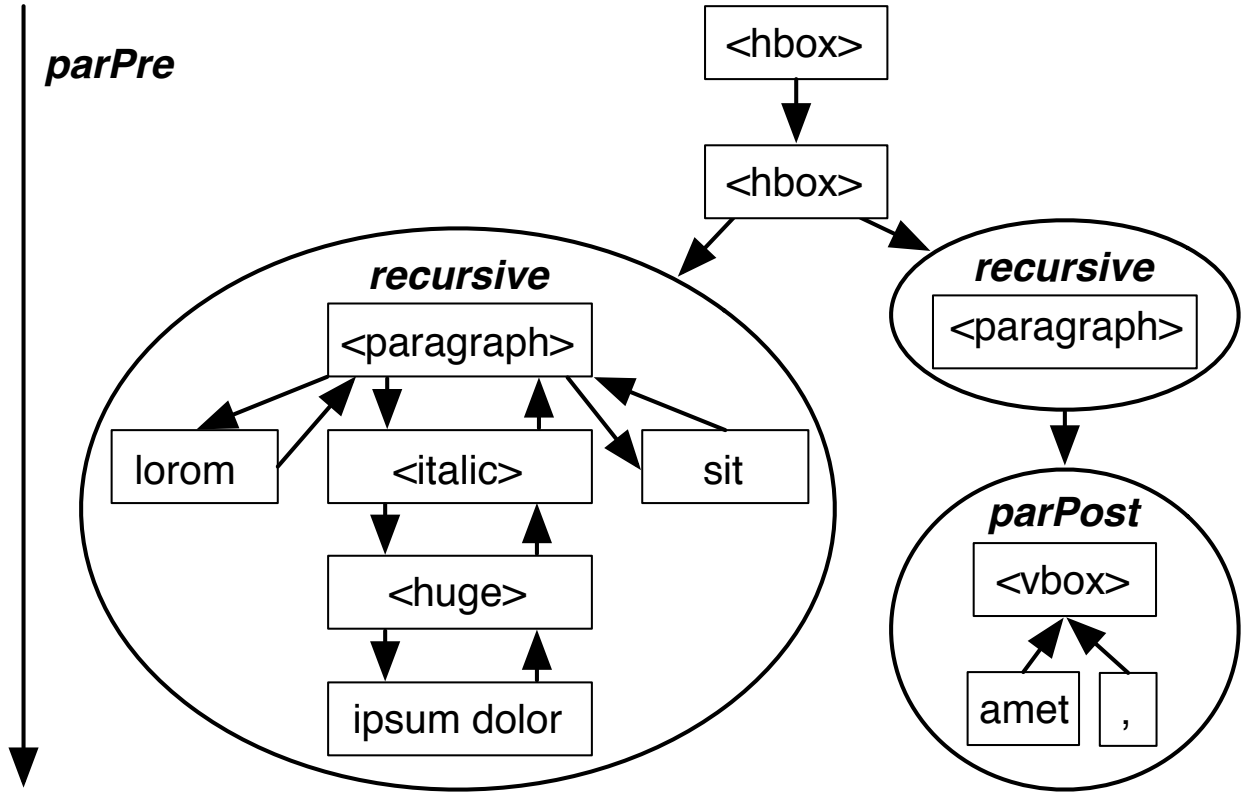


Figure 7.4: **Nested traversal for line breaking**. The two paragraph are traversed in parallel as part of a preorder traversal and a sequential recursive traversal is used for words within a paragraph.

Appendix A

Layout Grammars