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 MIMD: Semi-static work stealing

Scheduling

Data representation

Evaluation

6.2 GPU: Staying on-device

Level-synchronous breadth-first traversal

Rendering

Evaluation

6.3 Sub-word SIMD: Clustering

The problem

Instruction clustering

Data clustering

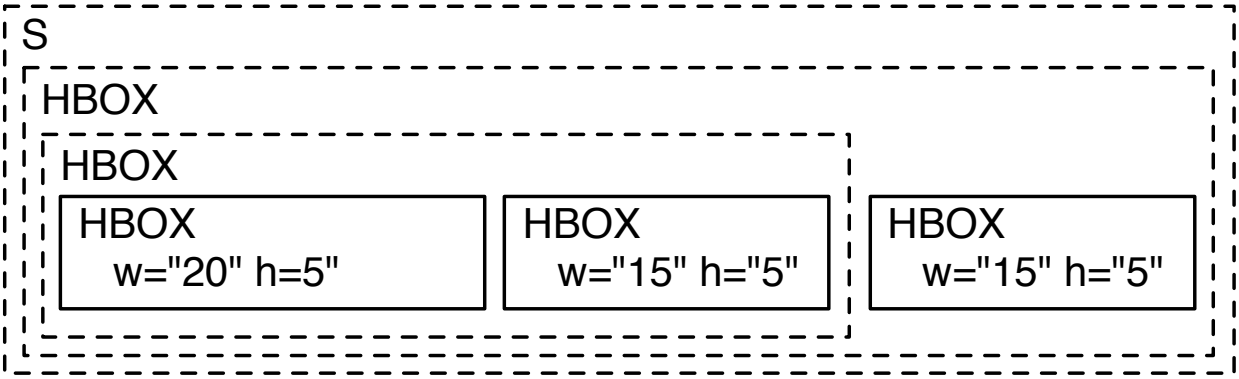
Evaluation

Limit study and benchmarks

Chapter 7

Conclusion

[**Input tree.** Only some of the x, y, w, and h attributes are specified.]



[**Attribute grammar for a language of horizontal boxes.**]

S → *HBOX*
{ HBOX.x = 0; HBOX.y = 0 }

HBOX → ϵ
{ HBOX.w = input$_w$(); HBOX.h = input$_h$() }

$HBOX_0$ → $HBOX_1$ $HBOX_2$
{ $HBOX_1$.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 }

[**Language of attribute grammars.**]

AG → (*Prod* **{** *Stmnt*? **}**)*

Prod → *V* → *V**

Stmnt → *Attrib* = *id*(*Attrib**) | *Attrib* = *n* | *Stmnt* ; *St*

Attrib → *id*.*id*

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).

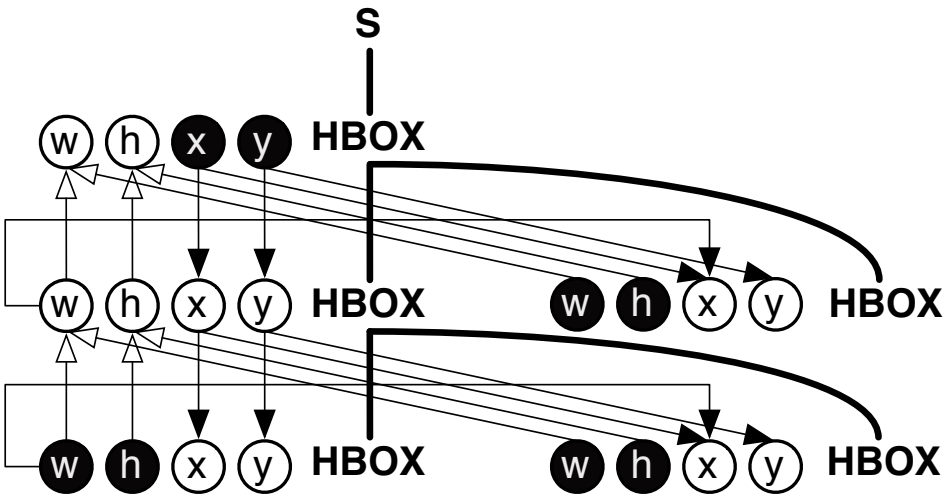


Figure 7.2: **Data dependencies.** Shown for constraint tree in Figure 2 (a). Circles denote attributes, with black circles being input() sources. Thin lines show data dependencies and thick lines show production derivations.

[**One explicit parallel schedule for** `H-AG` **.**]

```
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 }
```

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

```
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);
}
```

[**Scheduled and compiled layout engine for** `H-AG` **.**]

```
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;
  }
```

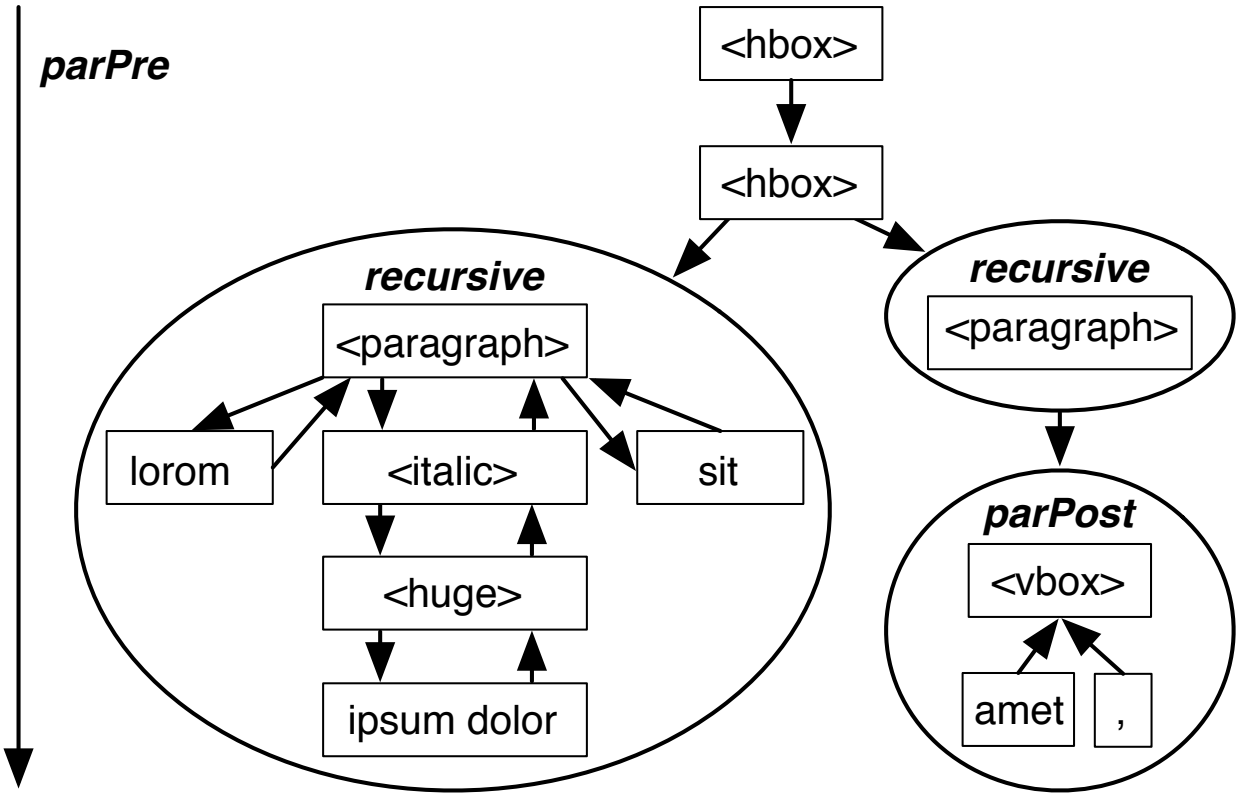


Figure 7.4: **Nested traversal for line breaking**. The two paragraph are traversed in parallel as part of a preorder traversal and a sequential recursive traversal is used for words within a paragraph.

Appendix A

Layout Grammars